Embedded Coder®

User's Guide

MATLAB®&SIMULINK®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Embedded Coder® User's Guide

© COPYRIGHT 2011–2023 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (Release 2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)
September 2017	Online only	Revised for Version 6.13 (Release 2017b)
March 2018	Online only	Revised for Version 7.0 (Release 2018a)
September 2018	Online only	Revised for Version 7.1 (Release 2018b)
March 2019	Online only	Revised for Version 7.2 (Release 2019a)
September 2019	Online only	Revised for Version 7.3 (Release 2019b)
March 2020	Online only	Revised for Version 7.4 (Release 2020a)
September 2020	Online only	Revised for Version 7.5 (Release 2020b)
March 2021	Online only	Revised for Version 7.6 (Release 2021a)
September 2021	Online only	Revised for Version 7.7 (Release 2021b)
March 2022	Online only	Revised for Version 7.8 (Release 2022a)
September 2022	Online only	Revised for Version 7.9 (Release 2022b)
March 2023	Online only	Revised for Version 7.10 (Release 2023a)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

Embedded Coder Fundamentals

1 Embedded Coder Fundamentals

Software Deployment Objectives and Deployment Types **1-2**
What Is Software Deployment? **1-2**
Deployment Objectives and Deployment Types **1-3**
Applications and Components **1-4**
Subcomponents and Utilities **1-7**
Nesting and Peer Composition Rules **1-8**

Target Environment Services **1-9**

Code Interfaces and Code Interface Specification **1-11**

Data Communication Methods **1-14**

Code Interface Configurations **1-16**
Options for Code Interface Configuration Storage **1-17**
Model-Owned Embedded Coder Dictionary **1-18**
Shared Embedded Coder Dictionary **1-18**
Code Interface Configurations for Data Objects **1-19**

Deploy Generated Software **1-20**
Identify Requirements **1-21**
Create C Code Interface Configuration **1-22**
Develop Deployable Code **1-26**
Deploy Generated Code **1-31**

Code Generation Goal Summary **1-36**

Customize Code Generation Environment **1-39**

Embedded Coder Supported Hardware **1-41**

Model Architecture and Design

2 Design Preparation

Divide Software Problem 2-2

Map Embedded System Architecture to Simulink Modeling Environment 2-3
- Modeling Algorithms 2-3
- Modeling Interfaces 2-4
- Modeling Systems 2-8
- Modeling Target Platform Environments 2-9

3 Application and Component Interfaces

Model Code Interfaces 3-2

Configure Code Interfaces for Code Generation 3-4

Periodic and Aperiodic Function Interfaces 3-6
- Rate-Based Top Models 3-6
- Export-Function Top Models 3-7
- Periodic and Aperiodic Function Interface Configuration 3-9

Startup, Reset, and Shutdown Function Interfaces 3-10
- Generate Code for Initialize and Terminate Events 3-11
- Generate Code for Reset Events 3-14
- Event Names and Code Aggregation 3-16
- Limitation 3-17

Data Interfaces 3-19
- C Data Interfaces 3-19
- C++ Data Interfaces 3-19

Service Interfaces 3-21
- Receiver and Sender Service Interfaces 3-21
- Data Transfer Service Interfaces 3-22
- Nonvolatile Memory Interfaces 3-23
- Timer Service Interfaces 3-25
- Parameter and Parameter Argument Tuning Service Interfaces . . . 3-27
- Measurement Service Interfaces 3-28
- Service Interface Constraints and Limitations 3-29

Message Communication Interfaces 3-33
- Generate C++ Messages to Communicate Data Between Simulink and an Operating System or Middleware 3-33

Generate C++ Messages to Communicate Data Between Simulink Components . . . 3-37
Generate C Messages to Communicate Data Between Simulink Components . . . 3-42
Model Message-Based Communication Integrated with POSIX Message Queues . . . 3-47
Generate Code for Wireless Sensor in Tire Pressure Monitoring Messaging System . . . 3-54

Client-Server Communication Interfaces . . . 3-65
Client-Server C Code Commuication . . . 3-65
Client-Server C++ Code Communication . . . 3-70

Device Driver I/O Interfaces . . . 3-75

Multi-Instantiation . . . 3-77

Model Single-Core, Rate-Monotonic Multitasking Application . . . 3-78

Model Multicore Concurrent Tasking Application . . . 3-81

Generate Code for Atomic Subsystems . . . 3-86

Deploy Export-Function Component Configured for C Service Interface Code Generation . . . 3-89

Deploy Rate-Based Component Configured for C Service Interface Code Generation . . . 3-103

Model Single-Core, Single-Tasking Application . . . 3-114

4 Subsystems in Simulink Coder

Generate Code and Executables for Individual Subsystems . . . 4-2
Subsystem Build Limitations . . . 4-3

Generate Inlined Subsystem Code . . . 4-5
Configure Subsystem to Inline Code . . . 4-5
Exceptions to Inlining . . . 4-6
See Also . . . 4-6

Generate Subsystem Code as Separate Function and Files . . . 4-7
Subsystem Function Dependence . . . 4-7
Generate Subsystem as a Reusable Function . . . 4-7
Generate Subsystem as a Nonreusable Function . . . 4-8

5 Subsystems in Embedded Coder

Generate Predictable Function Interface to Match Graphical Block Interface **5-2**

6 Referenced Models in Simulink Coder

Generate Code for Model Reference Hierarchy **6-2**
Model Reference Technology **6-2**
Generate Code for Model Reference Hierarchy **6-2**
Working with Code Generation Folders **6-6**
Build Model Reference Targets **6-7**

Set Configuration Parameters for Code Generation of Model Hierarchies **6-8**
Configuration Parameter Requirements **6-8**
Configuration Parameters Changed During Code Generation **6-10**
Naming Requirements **6-11**
Custom Target Requirements **6-11**
Miscellaneous Considerations **6-11**

Convert Subsystem to Referenced Model and Generate Code **6-13**
Configure Code Mappings **6-13**
Convert Subsystem to Referenced Model **6-13**
Copy Code Mappings After Subsystem Conversion **6-15**
Review Generated Code **6-15**

Generate Reusable Code from Model Reference Hierarchy **6-16**

Storage Classes for Parameters and Signals Used in Model Blocks **6-19**
Storage Classes for Parameters in Model Blocks **6-19**
Storage Classes for Signals in Model Blocks **6-20**
Signal Name Mismatches Across Model Reference Boundary **6-20**

Code Generation Limitations for Model Reference **6-22**
Storage Class Limitations **6-22**
Data Logging Limitations **6-22**
State Initialization Limitations **6-22**
Model Reuse Limitations **6-22**
S-Functions Limitations **6-23**
Subsystems Limitations **6-23**
Targets Limitation **6-23**
Conditional Models Limitation **6-23**
Local Solver Limitation **6-23**

Simulink Cache Files for Incremental Code Generation **6-24**
Contents of Simulink Cache Files **6-24**
Share Code Generation Artifacts **6-24**

Specify Instance-Specific Parameter Values for Reusable Referenced Model **6-27**
Pass Parameter Data to Referenced Model Entry-Point Functions as Arguments **6-27**
Control Data Types of Model Arguments and Argument Values **6-49**

Generated Functions in a Model Reference Hierarchy **6-51**
Generated Functions in a Top Model **6-51**
Generated Functions in a Referenced Model **6-52**

7 Combined Models in Simulink Coder

Combine Code Generated for Multiple Models **7-2**
Considerations for Combining Code for Different Models **7-2**
Control Ownership of Data **7-3**
Combine Code Generated for Multiple Models or Multiple Instances of a Model **7-3**

Function Reuse in Generated Code **7-5**

File Packaging for Models (Code and Data) **7-10**

8 Code Reuse for Simulink Coder

What Is Code Reuse? **8-2**

What Is Reentrant Code? **8-3**

Choose a Componentization Technique for Code Reuse **8-4**

Simulink Function Blocks and Code Generation **8-6**
Why Generate Code from Simulink Function Blocks and Function Callers? **8-6**
Implementation Options **8-6**
Choose a Modeling Pattern **8-6**
Specify Function Scope **8-8**
Decide Whether to Generate Export Function Code **8-8**
Configure Function Code Interfaces **8-8**
Uncalled Simulink Function Blocks **8-8**
Requirements **8-9**
Limitations **8-9**
Generate Code for Simulink Function and Function Caller **8-9**

Generate Reentrant Code from Top Models **8-13**
Generate Reentrant, Multi-Instance Code **8-13**
Share Data Between Instances **8-16**

Generate Reentrant Code from Simulink Function Blocks **8-18**
Identify Requirements **8-18**
Create Model **8-18**
Configure Model and Model Elements **8-20**
Generate and Inspect C Code **8-22**
Generate and Inspect C++ Code **8-24**
Limitations **8-26**

Generate Reentrant Code from Subsystems **8-28**
Function Reuse in Generated Code **8-29**
Limitations **8-33**

Generate Reusable Code from Library Subsystems Shared Across Models **8-36**
What Is a Reusable Library Subsystem? **8-36**
Reusable Library Subsystem Code Placement and Naming **8-36**
Configure Models That Include Reusable Library Subsystems **8-38**
Generate Reusable Code for Subsystems Shared Across Models . . . **8-39**

Generate Reusable Code from Stateflow Atomic Subcharts **8-46**
Generate Reusable Code for Linked Atomic Subcharts **8-46**
Generate Reusable Code for Unlinked Atomic Subcharts **8-46**
Generate Reusable Code for Unit Testing **8-47**

Generate Shared Utility Code **8-52**
When to Generate Shared Utility Code **8-52**
Configure Naming of Generated Functions **8-52**
Control Placement of Shared Utility Code **8-53**
Control Placement of rtwtypes.h for Shared Utility Code **8-54**
Avoid Duplicate Header Files for Exported Data **8-54**
Reduce Shared Utility Code Generation with Incremental Builds . . **8-54**
Manage the Shared Utility Code Checksum **8-55**

Generate Shared Utility Code for Fixed-Point Functions **8-60**

Generate Shared Utility Code for Custom Data Types **8-62**

Shared Constant Parameters for Code Reuse **8-64**
Suppress Shared Constants in the Generated Code **8-65**

Determine Why Subsystem Code Is Not Reused **8-67**
Review Subsystems Section of HTML Code Generation Report **8-67**
Compare Subsystem Checksum Data **8-67**

Generate Reusable Code from For Each Subsystems **8-70**

9 Library-Based Code Generation

Library-Based Code Generation for Reusable Library Subsystems **9-2**
Example Model and Library **9-2**

Configure Reusable Library Subsystems 9-3
Specify Unique Function Interface Names 9-4
Configure and Manage Function Interfaces 9-5
Build the Library 9-8
Generate Code from Model Containing a Reusable Library Subsystem Instance 9-9
Verify Code Generated from Reusable Library Subsystems 9-10
Limitations 9-11

10 Configure Model Parameters for Simulink Coder

Configure Run-Time Environment Options 10-2
Configure Production and Test Hardware 10-2
Production Hardware Considerations 10-8
Test Hardware Considerations 10-8
Example Production Hardware Setting That Affects Normal Mode Simulation 10-9

Using the Target Framework 10-11
Target Framework Overview 10-11
Target Framework Workflows 10-11

Register New Hardware Devices 10-13
Specify Hardware Implementation for New Device 10-13
Specify Hardware Implementation That Persists Over MATLAB Sessions 10-14
Create Hardware Implementation by Modifying Existing Implementation 10-14
Create Hardware Implementation by Reusing Existing Implementation 10-15
Validate Hardware Device Data 10-15
Export Hardware Device Data 10-16
Create Alternative Identifier for Target Object 10-16
Upgrade Data Definitions for Hardware Devices 10-17

11 Model Protection in Simulink Coder

Protect Models to Conceal Contents 11-2
Prepare the Parent Model 11-2
Protect the Referenced Model 11-3
Protected Model Requirements and Limitations 11-6
Code Generation Requirements and Limitations 11-7
Nested Protected Model Requirements and Limitations 11-8

Create Protected Models with Multiple Targets 11-9

Test Protected Models 11-10

Package and Share Protected Models **11-12**
Harness Model **11-12**
MAT-File with Base Workspace Definitions **11-12**
Simulink Data Dictionary **11-13**
Protected Model File Contents **11-13**

Specify Custom Obfuscators for Protected Models **11-16**

Define Callbacks for Protected Models **11-17**
Creating Callbacks **11-17**
Defining Callback Code **11-17**
Create a Protected Model with Callbacks **11-18**

Sign a Protected Model **11-19**
Sign Your Protected Model **11-19**

Best Practices for Creating Protected Models **11-21**
Coordinate Settings with the User of Your Protected Model **11-21**
Minimize the Model External Interface **11-21**
Contents to Include in Your Protected Model **11-23**

12 Stateflow Blocks in Simulink Coder

Code Generation of Stateflow Blocks **12-2**
Comparison of Code Generation Methods **12-2**

Generate Code from Atomic Subcharts **12-5**
Generate Reusable Code for Unlinked Atomic Subcharts **12-5**
Generate Reusable Code for Linked Atomic Subcharts **12-5**

Inline State Functions in Generated Code **12-7**
Inlined Generated Code for State Functions **12-7**
How to Set the State Function Inline Option **12-8**
Best Practices for Controlling State Function Inlining **12-8**

Air-Fuel Ratio Control System with Stateflow Charts **12-10**

13 Block Authoring and Code Generation for Simulink Coder

S-Functions and Code Generation **13-2**
Types of S-Functions **13-2**
Files Required for Implementing Noninlined and Inlined S-Functions **13-4**
Guidelines for Writing S-Functions that Support Code Generation **13-4**

Import Calls to External Code into Generated Code with Legacy Code Tool **13-6**
Legacy Code Tool and Code Generation **13-6**
Generate Inlined S-Function Files for Code Generation **13-6**
Apply Code Style Settings to Legacy Functions **13-7**
Address Dependencies on Files in Different Locations **13-8**
Deploy S-Functions for Simulation and Code Generation **13-8**
Integrate External C++ Objects **13-8**

Legacy Code Tool Examples **13-11**
Integrate External C Functions That Pass Input Arguments by Value Versus Address **13-11**
Integrate External C Functions That Pass the Output Argument as a Return Argument **13-14**
Integrate External C++ Object Methods **13-16**
Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type **13-18**
Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type **13-21**
Integrate External C Functions with Instance-Specific Persistent Memory **13-23**
Integrate External C Functions That Use Structure Arguments . . . **13-25**
Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data **13-28**
Integrate External C Functions That Pass Arguments That Have Inherited Dimensions **13-30**
Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals **13-33**
Integrate External C Functions That Implement Start and Terminate Actions **13-35**
Integrate External C Functions That Implement N-Dimensional Table Lookups **13-37**
Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals **13-40**
Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized **13-43**

Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type **13-47**

Integrate External C Functions That Implement N-Dimensional Table Lookups **13-50**

Integrate External C++ Object Methods **13-53**

External Code Integration Examples **13-56**
Integrate External C++ Code into a Model Using S-Functions . . . **13-56**
Insert External C and C++ Code into Stateflow Charts for Code Generation **13-57**
Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters **13-59**
Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters **13-60**
Insert External C++ Code into Stateflow Charts for Code Generation **13-62**

S-Functions That Support Expression Folding **13-64**
Categories of Output Expressions **13-64**
Acceptance or Denial of Requests for Input Expressions **13-67**
Expression Folding in a TLC Block Implementation **13-69**

S-Functions That Specify Port Scope and Reusability **13-73**

S-Functions That Specify Sample Time Inheritance Rules **13-77**

S-Functions for Code Reuse **13-81**

S-Functions for Multirate Multitasking Environments **13-86**
About S-Functions for Multirate Multitasking Environments **13-86**
Rate Grouping Support in S-Functions **13-86**
Create Multitasking, Multirate, Port-Based Sample Time S-Functions **13-86**

Write Noninlined S-Function **13-91**
Guidelines for Writing Noninlined S-Functions **13-91**
Noninlined S-Function Parameter Type Limitations **13-92**

Write Wrapper S-Function and TLC Files **13-93**
MEX S-Function Wrapper **13-93**
TLC S-Function Wrapper **13-96**
Code Overhead for Noninlined S-Functions **13-96**
Inline a Wrapper S-Function **13-97**
The Inlined Code **13-99**

Write Fully Inlined S-Functions **13-100**
Multiport S-Function **13-100**
Guidelines for Writing Inlined S-Functions **13-100**

Write Fully Inlined S-Functions with mdlRTW Routine **13-102**
S-Function RTWdata **13-102**
Direct-Index Lookup Table Algorithm **13-103**
Direct-Index Lookup Table Example **13-104**
Error Handling **13-105**
User Data Caching **13-105**
mdlRTW Usage **13-105**

14 What is the Target Language Compiler?

Target Language Compiler Basics **14-2**
Target Language Compiler Overview **14-2**
Overview of the TLC Process **14-2**
Overview of the Code Generation Process **14-4**

Why Use the Target Language Compiler? **14-6**
Customizing Output **14-6**
Inlining S-Functions **14-6**
Defining Advanced Storage Classes **14-6**

The Advantages of Inlining S-Functions **14-8**
When To Avoid Inlining **14-8**
Inlining Process **14-8**
Search Algorithm for Locating TLC Files **14-9**
Availability for Inlining and Noninlining **14-9**

15 Getting Started

Code Architecture **15-2**
Code Generation Process **15-3**
How TLC Determines S-Function Inlining Status **15-3**
Inlined and Noninlined S-Function Code **15-3**

Target Language Compiler Process **15-6**
model.rtw Structure **15-6**
Operating Sequence **15-6**

Inlining S-Functions **15-8**
Inlining an S-function **15-8**
Noninlined S-Function **15-8**
Types of Inlining **15-9**
Fully Inlined S-Function Example **15-9**
Wrapper Inlined S-Function Example **15-11**

16 Target Language Compiler Tutorials

Advice About TLC Tutorials **16-2**

Read Record Files with TLC **16-4**
Tutorial Overview **16-4**
Structure of Record Files **16-4**
Interpret Records **16-5**
Anatomy of a TLC Script **16-6**
Modify read-guide.tlc **16-11**
Pass and Use a Parameter **16-14**
Review **16-16**

Inline S-Functions with TLC **16-17**
timesN Tutorial Overview **16-17**
Noninlined Code Generation **16-17**
Why Use TLC to Inline S-Functions? **16-18**
Create an Inlined S-Function **16-18**

Explore Variable Names and Loop Rolling **16-21**
timesN Looping Tutorial Overview **16-21**
Getting Started **16-21**
Modify the Model **16-21**

Change the Loop Rolling Threshold 16-23
More About TLC Loop Rolling 16-24

Debug Your TLC Code 16-26
tlcdebug Tutorial Overview 16-26
Getting Started 16-26
Generate and Run Code from the Model 16-27
Start the Debugger and Use Its Commands 16-28
Debug timesN.tlc 16-29
Fix the Bug and Verify 16-30

TLC Code Coverage to Aid Debugging 16-32
tlcdebug Execute Tutorial Overview 16-32
Getting Started 16-32
Open the Model and Generate Code 16-32

Wrap User Code with TLC 16-34
wrapper Tutorial Overview 16-34
Why Wrap User Code? 16-34
Getting Started 16-36
Generate Code Without a Wrapper 16-36
Generate Code Using a Wrapper 16-37

17 Code Generation Architecture

Build Process 17-2
Build Process Overview 17-2
Create and Use Target Language File 17-2

Configure TLC 17-5
Set Command-Line Arguments 17-5
Configure for TLC Debugging 17-6

Code Generation Concepts 17-7
Output Streams 17-7
Variable Types 17-7
Records 17-8
Record Aliases 17-9

TLC Files 17-11
TLC Program 17-11
Available Target Files 17-11
Target File Usage 17-12
System Target Files 17-13
Compatibility of TLC Files with Generated Code Data Types 17-13

Data Handling with TLC 17-15
Matrix Parameters 17-15
Code Generator Matrix Parameters 17-15

18 model.rtw File and Authoring S-Functions and Data Objects

model.rtw File and Scopes **18-2**
Scopes in the model.rtw File **18-2**

Data Object Information in model.rtw File **18-5**
Object Records for Parameters **18-5**
Object Records for Signals **18-6**
Access Data Object Information by Using Target Language Compiler (TLC) Code **18-7**

Data References in the model.rtw File **18-9**
Data Reference Overview **18-9**
Control Data Reference Threshold **18-9**
Expand Data References **18-9**
Avoid Data Reference Expansion **18-10**
Restart Code Generation **18-10**

Exception to Using the Library Functions that Access model.rtw **18-11**
Example Exception to Using the Library Functions **18-11**
Caution Against Directly Accessing Record Fields **18-12**

19 Directives and Built-In Functions

Target Language Compiler Directives **19-2**
Syntax **19-2**
Directives **19-2**
Comments **19-12**
Line Continuation **19-13**
Target Language Value Types **19-13**
Target Language Expressions **19-14**
Formatting **19-19**
Conditional Inclusion **19-19**
Multiple Inclusion **19-20**
Object-Oriented Facility for Generating Target Code **19-23**
Output File Control **19-25**
Input File Control **19-26**
Asserts, Errors, Warnings, and Debug Messages **19-27**
Built-In Functions and Values **19-27**
TLC Reserved Constants **19-36**
Identifier Definition **19-36**
Variable Scoping **19-39**
Target Language Functions **19-46**

Command-Line Arguments **19-49**
Target Language Compiler Switches **19-49**
Filenames and Search Paths **19-51**

20 Debugging TLC Files

Using the TLC Debugger **20-2**
Tips for Debugging TLC Code **20-2**
Invoking the Debugger **20-2**
TLC Debugger Command Summary **20-3**

TLC Coverage **20-6**
Using the TLC Coverage Option **20-6**
Example .log File **20-6**
Analyzing the Results **20-8**

TLC Profiler **20-9**
Using the Profiler **20-9**
Analyzing the Report **20-9**
Nonexecutable Directives **20-10**
Improving Performance **20-10**

21 Inlining S-Functions

Inline S-Functions **21-2**
Inline S-Functions with Block Target Files **21-2**
Inline MATLAB File S-Functions **21-3**
Inline Fortran (F-MEX) S-Functions **21-4**

Inline C MEX S-Functions **21-8**
Inline S-Function Overview **21-8**
S-Function Parameters **21-9**
Sample Code for S-Function **21-9**

TLC Coding Conventions **21-17**
Begin Identifiers with Uppercase Letters **21-17**
Begin Global Variable Assignments with Uppercase Letters **21-17**
Begin Local Variable Assignments with Lowercase Letters **21-18**
Begin Functions Declared in block.tlc Files with Fcn **21-18**
Do Not Hard-Code Variables Defined in commonsetup.tlc **21-18**
Conditional Inclusion in Library Files **21-19**
Code Defensively **21-20**

Block Target File Methods **21-21**
Block Functions Overview **21-21**
BlockInstanceSetup(block, system) **21-21**
BlockTypeSetup(block, system) **21-22**
Enable(block, system) **21-23**
Disable(block, system) **21-23**
Start(block, system) **21-23**
InitializeConditions(block, system) **21-24**
Outputs(block, system) **21-25**
Update(block, system) **21-26**

Derivatives(block, system) 21-26
Terminate(block, system) 21-26

Loop Rolling 21-27

22 TLC Function Library Reference

Target Language Compiler Library Functions Overview 22-2

Target Language Compiler Function Conventions 22-3
Common Function Arguments 22-3
Overloading sigIdx 22-4

Input Signal Functions 22-7
LibBlockInputHasSymbolicWidth(portIdx) 22-7
LibBlockInputPortIndexMode(block, pidx) 22-7
LibBlockInputPortIsContinuousQuantity(portIdx) 22-7
LibBlockInputSignal(portIdx, ucv, lcv, sigIdx) 22-8
LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx) 22-12
LibBlockInputSignalAliasedThruDataTypeName(portIdx, reim) . . . 22-13
LibBlockInputSignalAllowScalarExpandedExpr(block,portIdx) . . . 22-13
LibBlockInputSignalASCIIEscapedUnitExpr(portIdx) 22-13
LibBlockInputSignalConnected(portIdx) 22-14
LibBlockInputSignalDataTypeId(portIdx) 22-14
LibBlockInputSignalDataTypeName(portIdx, reim) 22-14
LibBlockInputSignalDimensions(portIdx) 22-14
LibBlockInputSignalIsComplex(portIdx) 22-14
LibBlockInputSignalIsExpr(portIdx) 22-14
LibBlockInputSignalIsFrameData(portIdx) 22-15
LibBlockInputSignalIsTrivialExpr(portIdx) 22-15
LibBlockInputSignalLocalSampleTimeIndex(portIdx) 22-15
LibBlockInputSignalNumDimensions(portIdx) 22-15
LibBlockInputSignalOffsetTime(portIdx) 22-15
LibBlockInputSignalSampleTime(portIdx) 22-15
LibBlockInputSignalSampleTimeIndex(portIdx) 22-15
LibBlockInputSignalSymbolicDimensions(portIdx) 22-15
LibBlockInputSignalSymbolicWidth(portIdx) 22-16
LibBlockInputSignalWidth(portIdx) 22-16
LibBlockInputSignalUnitExpr(portIdx) 22-16
LibBlockInputSignalUnitId(portIdx) 22-16
LibBlockNumInputPorts(block) 22-16

Output Signal Functions 22-17
LibBlockAssignOutputSignal(portIdx, ucv, lcv, sigIdx, rhs) 22-17
LibBlockGetNumFcnCallOutputs(block) 22-17
LibBlockGetNumFcnCallOutputsDsts(block) 22-17
LibBlockNumOutputPorts(block) 22-17
LibBlockOutputHasSymbolicWidth(portIdx) 22-17
LibBlockOutputPortIndexMode(block, pidx) 22-17
LibBlockOutputPortIsContinuousQuantity(portIdx) 22-18
LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx) 22-18

LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx) 22-18
LibBlockOutputSignalAliasedThruDataTypeName(portIdx, reim) . 22-19
LibBlockOutputSignalBeingMerged(portIdx) 22-19
LibBlockOutputSignalConnected(portIdx) 22-19
LibBlockOutputSignalDataTypeId(portIdx) 22-19
LibBlockOutputSignalDataTypeName(portIdx, reim) 22-20
LibBlockOutputSignalDimensions(portIdx) 22-20
LibBlockOutputSignalIsComplex(portldx) 22-20
LibBlockOutputSignalIsConstExpr(portIdx) 22-20
LibBlockOutputSignalIsExpr(portldx) 22-20
LibBlockOutputSignalIsExprInOutputs(portIdx) 22-21
LibBlockOutputSignalIsFrameData(portIdx) 22-21
LibBlockOutputSignalIsNonConstExpr(portIdx) 22-21
LibBlockOutputSignalIsTrivialExpr(portIdx) 22-21
LibBlockOutputSignalLocalSampleTimeIndex(portIdx) 22-21
LibBlockOutputSignalNumDimensions(portIdx) 22-21
LibBlockOutputSignalOffsetTime(portIdx) 22-21
LibBlockOutputSignalSampleTime(portIdx) 22-21
LibBlockOutputSignalSampleTimeIndex(portIdx) 22-22
LibBlockOutputSignalSymbolicDimensions(portIdx) 22-22
LibBlockOutputSignalSymbolicWidth(portIdx) 22-22
LibBlockOutputSignalWidth(portIdx) 22-22
LibBlockOutputSignalUnitExpr(portIdx) 22-22
LibBlockOutputSignalUnitId(portIdx) 22-22

Parameter Functions 22-23
LibBlockMatrixParameter 22-23
LibBlockMatrixParameterAddr 22-23
LibBlockMatrixParameterBaseAddr 22-23
LibBlockParamSetting 22-23
LibBlockParameter 22-23
LibBlockParameterAddr 22-25
LibBlockParameterBaseAddr 22-25
LibBlockParameterDataTypeId 22-25
LibBlockParameterDataTypeName 22-25
LibBlockParameterDimensions 22-25
LibBlockParameterIsComplex 22-26
LibBlockParameterSize 22-26
LibBlockParameterString 22-26
LibBlockParameterValue 22-26
LibBlockParameterWidth 22-27
LibGetParameterValue 22-27

Block State and Work Vector Functions 22-29
LibBlockAssignDWork(dwork, ucv, lcv, sigIdx, rhs) 22-29
LibBlockContinuousState(ucv, lcv, idx) 22-29
LibBlockContinuousStateDerivative(ucv, lcv, idx) 22-29
LibBlockContStateDisabled(ucv, lcv, idx) 22-29
LibBlockDWork(dwork, ucv, lcv, idx) 22-29
LibBlockDWorkAddr(dwork, ucv, lcv, idx) 22-30
LibBlockDWorkDataTypeId(dwork) 22-30
LibBlockDWorkDataTypeName(dwork, reim) 22-30
LibBlockDWorkIsComplex(dwork) 22-30
LibBlockDWorkName(dwork) 22-30
LibBlockDWorkStorageClass(dwork) 22-30
LibBlockDWorkStorageTypeQualifier(dwork) 22-30

LibBlockDWorkUsedAsDiscreteState(dwork) 22-30
LibBlockDWorkWidth(dwork) 22-31
LibBlockDiscreteState(ucv, lcv, idx) 22-31
LibBlockIWork(definediwork, ucv, lcv, idx) 22-31
LibBlockMode(ucv, lcv, idx) 22-31
LibBlockNonSampledZC(ucv, lcv, NSZCIdx) 22-31
LibBlockPWork(definedpwork, ucv, lcv, idx) 22-31
LibBlockRWork(definedrwork, ucv, lcv, idx) 22-31
LibBlockZCSignalValue(ucv, lcv, zcsIdx, zcElIdx) 22-32

Block Path and Error Reporting Functions 22-33
LibBlockReportError(block, errorstring) 22-33
LibBlockReportFatalError(block, errorstring) 22-33
LibBlockReportWarning(block, warnstring) 22-33
LibGetBlockName(block) 22-33
LibGetBlockPath(block) 22-34
LibGetFormattedBlockPath(block) 22-34

Code Configuration Functions 22-35
LibAddSourceFileCustomSection(file, builtInSection, newSection) 22-35
LibAddToCommonIncludes(incFileName) 22-35
LibAddToModelSources(newFile) 22-35
LibCacheDefine(buffer) 22-36
LibCacheExtern(buffer) 22-36
LibCacheFunctionPrototype(buffer) 22-36
LibCacheTypedefs(buffer) 22-37
LibCallModelInitialize() 22-37
LibCallModelStep(tid) 22-37
LibCallModelTerminate() 22-37
LibCallSetEventForThisBaseStep(buffername) 22-37
LibClearFileSectionContents(fileIdx, attrib) 22-37
LibCreateSourceFile(type, creator, name) 22-38
LibGetFileRecordName (file) 22-38
LibGetMdlPrvHdrBaseName() 22-38
LibGetMdlPubHdrBaseName() 22-38
LibGetMdlSrcBaseName() 22-39
LibGetMdlDataSrcBaseName() 22-39
LibGetMdlTypesHdrBaseName() 22-39
LibGetMdlCapiHdrBaseName() 22-39
LibGetMdlCapiSrcBaseName() 22-39
LibGetMdlCapiHostHdrBaseName() 22-39
LibGetMdlTestIfHdrBaseName() 22-39
LibGetMdlTestIfSrcBaseName() 22-39
LibGetDataTypeTransHdrBaseName() 22-40
LibGetModelDotCFile() 22-40
LibGetModelDotHFile() 22-40
LibGetModelName() 22-40
LibGetNumSourceFiles() 22-40
LibGetRTModelErrorStatus() 22-41
LibGetSourceFileAttribute(fileIdx, attrib) 22-41
LibGetSourceFileFromIdx(fileIdx) 22-41
LibGetSourceFileSection(fileIdx, section) 22-42
LibGetSourceFileTag(fileIdx) 22-42
LibMdlRegCustomCode(buffer, location) 22-42
LibMdlStartCustomCode(buffer, location) 22-43

LibMdlTerminateCustomCode(buffer, location) 22-43
LibNotifySymbolExportedFromFile 22-44
LibNotifySymbolUsedByFile 22-44
LibSetRTModelErrorStatus(str) 22-45
LibSetSourceFileCodeTemplate(opFile, name) 22-45
LibSetSourceFileCustomSection(file, attrib, value) 22-46
LibSetSourceFileOutputDirectory(opFile, name) 22-46
LibSetSourceFileSection(fileH, section, value) 22-46
LibSystemDerivativeCustomCode(system, buffer, location) 22-47
LibSystemDisableCustomCode(system, buffer, location) 22-48
LibSystemEnableCustomCode(system, buffer, location) 22-49
LibSystemInitializeCustomCode(system, buffer, location) 22-50
LibSystemOutputCustomCode(system, buffer, location) 22-51
LibSystemUpdateCustomCode(system, buffer, location) 22-52
LibWriteModelData() 22-53
LibWriteModelInput(tid, rollThreshold) 22-53
LibWriteModelInputs() 22-53
LibWriteModelOutput(tid, rollThreshold) 22-54
LibWriteModelOutputs() 22-54

Sample Time Functions 22-55
LibAsynchronousTriggeredTID(tid) 22-55
LibAsyncTaskAccessTimeInFcn(tid, fcnType) 22-55
LibBlockSampleTime(block) 22-55
LibGetClockTick(tid) 22-55
LibGetClockTickDataTypeId(tid) 22-55
LibGetClockTickHigh(tid) 22-55
LibGetClockTickStepSize(tid) 22-56
LibGetElapseTime(system) 22-56
LibGetElapseTimeCounter(system) 22-56
LibGetElapseTimeCounterDTypeId(system) 22-56
LibGetElapseTimeResolution(system) 22-56
LibGetGlobalTIDFromLocalSFcnTID(sfcnTID) 22-56
LibGetNumAsyncTasks() 22-57
LibGetNumSFcnSampleTimes(block) 22-57
LibGetNumSyncPeriodicTasks() 22-58
LibGetNumTasks() 22-58
LibGetSampleTimePeriodAndOffset(tid, idx) 22-58
LibGetSFcnTIDType(sfcnTID) 22-58
LibGetTaskTime(tid) 22-59
LibGetTaskTimeFromTID(block) 22-59
LibGetTID01EQ() 22-59
LibIsContinuous(TID) 22-59
LibIsDiscrete(TID) 22-60
LibIsSFcnSampleHit(sfcnTID) 22-60
LibIsSFcnSingleRate(block) 22-60
LibIsSFcnSpecialSampleHit(sfcnSTI, sfcnTID) 22-60
LibIsSingleRateModel() 22-61
LibIsSingleTasking() 22-61
LibIsZOHContinuous(TID) 22-61
LibNumAsynchronousSampleTimes() 22-61
LibNumDiscreteSampleTimes() 22-61
LibNumSynchronousSampleTimes() 22-62
LibPortBasedSampleTimeBlockIsTriggered(block) 22-62
LibSetVarNextHitTime(block, tNext) 22-62
LibTriggeredTID(tid) 22-62

Miscellaneous Functions **22-63**
LibAddtoSystemCustomIncludes(system, incFileName) **22-63**
LibBlockExecuteFcnCall(block, callIdx) **22-63**
LibBlockExecuteFcnDisable(block, callIdx) **22-63**
LibBlockExecuteFcnEnable(block, callIdx) **22-64**
LibBlockInputSignalAliasedThruDataTypeId(idx) **22-64**
LibBlockOutputSignalAliasedThruDataTypeId(idx) **22-64**
LibGenConstVectWithInit(data, typeId, varId) **22-64**
LibGetBlockAttribute(block, attr) **22-64**
LibGetCallerClockTickCounter(sfcnBlock) **22-65**
LibGetCallerClockTickCounterHigh(sfcnBlock) **22-65**
LibGetDataTypeComplexNameFromId(id) **22-65**
LibGetDataTypeEnumFromId(id) **22-65**
LibGetDataTypeIdAliasedThruToFromId(id) **22-66**
LibGetDataTypeIdAliasedToFromId(id) **22-66**
LibGetDataTypeIdResolvesToFromId(id) **22-66**
LibGetDataTypeNameFromId(id) **22-66**
LibGetDataTypeSLSizeFromId(id) **22-66**
LibGetDataTypeStorageIdFromId(id) **22-66**
LibGetFcnCallBlock(sfcnblock,callIdx) **22-67**
LibGetRecordDataTypeId(rec) **22-67**
LibGetRecordDimensions(rec) **22-67**
LibGetRecordIsComplex(rec) **22-67**
LibGetRecordWidth(rec) **22-67**
LibGetT() **22-67**
LibIsComplex(arg) **22-68**
LibIsFirstInitCond() **22-68**
LibIsMajorTimeStep() **22-68**
LibIsMinorTimeStep() **22-68**
LibManageAsyncCounter(sfcnBlock, callIdx) **22-68**
LibMaxIntValue(dtype) **22-69**
LibMinIntValue(dtype) **22-69**
LibNeedAsyncCounter(sfcnBlock, callIdx) **22-69**
LibSetAsyncClockTicks(sfcnBlock, callIdx, buf1, buf2) **22-69**
LibSetAsyncCounter(sfcnBlock, callIdx, buf) **22-70**
LibSetAsyncCounterHigh(sfcnBlock, callIdx, buf) **22-70**
LibTIDInSystem(system, fcnType) **22-70**
LibIsRowMajor **22-71**
Obsolete Functions **22-71**

Advanced Functions **22-73**
LibAppendToModelReferenceUserData(data) **22-73**
LibBlockInputSignalBufferDstPort(portIdx) **22-73**
LibBlockInputSignalStorageClass(portIdx, sigIdx) **22-74**
LibBlockInputSignalStorageTypeQualifier(portIdx, sigIdx) **22-74**
LibBlockOutputSignalIsGlobal(portIdx) **22-74**
LibBlockOutputSignalIsInBlockIO(portIdx) **22-74**
LibBlockOutputSignalIsValidLValue(portIdx) **22-74**
LibBlockOutputSignalStorageClass(portIdx) **22-75**
LibBlockOutputSignalStorageTypeQualifier(portIdx) **22-75**
LibBlockSrcSignalBlock(portIdx, sigIdx) **22-75**
LibBlockSrcSignalIsDiscrete(portIdx, sigIdx) **22-76**
LibBlockSrcSignalIsGlobalAndModifiable(portIdx, sigIdx) **22-76**
LibBlockSrcSignalIsInvariant(portIdx, sigIdx) **22-76**
LibGetModelReferenceUserData(modelName) **22-76**
LibGetReferencedModelNames() **22-76**

LibIsModelReferenceRTWTarget() 22-77
LibIsModelReferenceSimTarget() 22-77
LibIsModelReferenceTarget() 22-77

Block Compatibility Functions 22-78
LibBlockSetIsExpressionCompliant(block) 22-78
LibBlockSetIntegerDowncastUnnecessary(block) 22-78
LibBlockSetCustomStorageComplianceLevel (block, level) 22-78

A TLC Error Handling

TLC Error Handling A-2
Error Reporting A-2
Generating Errors from TLC Files A-2
Using TLC Error Messages to Troubleshoot A-4
%closefile or %selectfile or %flushfile argument must be a valid open file A-4
%define no longer supported, use %function instead A-4
%error directive: text A-4
%exit directive: text A-4
%filescope has already been used in this file A-5
%trace directive: text A-5
%warning directive: text A-5
A %implements directive must appear within a block template file and must match the %language and type specified A-5
A %switch statement can only have one %default A-5
A language choice must be made using the %language directive prior to using GENERATE or GENERATE_TYPE A-5
A non-homogeneous vector was passed to GENERATE_FORMATTED_VALUE A-6
Ambiguous reference to identifier — must use array index to refer to one of multiple scopes A-6
An %if statement can only have one %else A-6
Argument to identifier must be a string A-6
Arguments to directive must be records A-7
Arguments to TLC from the MATLAB command line must be strings A-7
Assertion failed A-7
Assignment to scope identifier is only allowed when using the + operator to add members A-7
Attempt to define a function identifier on top of an existing variable or function A-7
Attempt to divide by zero A-7
Bad cast - unable to cast this expression to type A-8
Bad directory (dirname) in O: filename A-8
builtin was expecting expression of type type, got one of type type A-8
Cannot %undef any builtin functions or variables A-8
Cannot convert string your_string to a number A-8
Changing value of identifier from the RTW file A-8
Error opening filename A-8
Error writing to file error A-8

Errors occurred — aborting **A-8**
Expansion directives %<> cannot be nested **A-8**
Expansion directives %<> cannot span multiple lines; use \ at end of line **A-9**
Extra arguments to the function-name built-in function were ignored (Warning) **A-9**
File name too long (directory =dirname, name =filename) **A-9**
format is not a legal format value **A-9**
Function argument mismatch; function function_name expects number arguments **A-10**
Function reached the end and did not return a value **A-10**
Function values are not allowed **A-10**
Identifier identifier multiply defined. Second and succeeding definitions ignored. **A-10**
Identifier identifier used on a %foreach statement was already in scope (Warning) **A-10**
Illegal use of eval (i.e., %<...>) **A-10**
Indices may not be negative **A-10**
Indices must be constant integral numbers **A-10**
Invalid handle **A-10**
Invalid identifier range, the leading strings string1 and string2 must match **A-11**
Invalid identifier range, the lower bound (bound) must be less than the upper bound (bound) **A-11**
Invalid type for unary operator **A-11**
Invalid type type **A-11**
It is illegal to return a function from a function **A-11**
Named value identifier already exists within this scope-identifier; use %assign to change the value **A-11**
No %case statement(s) seen yet, statement ignored **A-11**
Only double and character arrays can be converted from MATLAB to TLC. This can occur if the MATLAB function does not return a value (see %matlab) **A-12**
Only one output is allowed from the TLC **A-12**
Only strings of length 1 can be assigned using the [ ] notation **A-12**
Only strings or cells of strings may be used as the argument to Query and ExecString **A-12**
Only vectors of the same length as the existing vector value can be assigned using the [ ] notation **A-12**
Output file identifier opened with %openfile was not closed **A-12**
Ranges, identifier ranges, and repeat values cannot be repeated . . **A-12**
String cannot modify the setting for the command line switch '-switch' **A-12**
string is not a recognized user defined property of this handle **A-13**
Syntax error **A-13**
The %break directive can only appear within a %foreach, %for, %roll, or %switch statement **A-13**
The %case and %default directives can only be used within the %switch statement **A-13**
The %continue directive can only appear within a %foreach, %for, or %roll statement **A-13**
The %foreach statement expects a constant numeric argument . . . **A-13**
The %if statement expects a constant numeric argument **A-13**
The %implements directive expects a string or string vector as the list of languages **A-13**

The %implements directive specifies type as the type where type was expected **A-14**
The %implements language does not match the language currently being generated (language) **A-14**
The %return statement can only appear within the body of a function **A-14**
The == and != operators can only be used to compare values of the same type **A-14**
The argument for %openfile must be a valid string **A-14**
The argument for %with must be a valid scope **A-14**
The argument for an [ ] operation must be a repeated scope symbol, a vector, or a matrix **A-14**
The argument to %addincludepath must be a valid string **A-15**
The argument to %include must be a valid string **A-15**
The begin directive must be in the same file as the corresponding end directive. **A-15**
The begin directive on this line has no matching end directive **A-15**
The construct %matlab function_name(...) construct is illegal in standalone tlc **A-15**
The FEVAL() function can accept only 2-dimensional arrays from MATLAB, not number dimensions **A-16**
The FEVAL() function can accept vectors of numbers or strings only when calling MATLAB **A-16**
The FEVAL() function requires the name of a function to call **A-16**
The final argument to %roll must be a valid block scope **A-16**
The first argument of a ? : operator must be a Boolean expression . **A-16**
The first argument to GENERATE or GENERATE_TYPE must be a valid scope **A-16**
The function name requires at least number arguments **A-16**
The GENERATE function requires at least 2 arguments **A-16**
The GENERATE_TYPE function requires at least 3 arguments **A-16**
The ISINF(), ISNAN(), ISFINITE(), REAL(), and IMAG() functions expect a real or complex valued argument **A-17**
The language being implemented cannot be changed within a block template file **A-17**
The language being implemented has changed from old-language to new-language (Warning) **A-17**
The left-hand side of a . operator must be a valid scope identifier . . **A-17**
The left-hand side of an assignment must be a simple expression comprised of ., [ ], and identifiers **A-17**
The number of columns specified (specified-columns) did not match the actual number of columns in the rows (actual-columns) **A-17**
The number of rows specified (specified-rows) did not match the actual number of rows seen in the matrix (actual-rows) **A-17**
The operator_name operator only works on Boolean arguments . . . **A-18**
The operator_name operator only works on integral arguments . . . **A-18**
The operator_name operator only works on numeric arguments . . . **A-18**
The return value from the RollHeader function must be a string . . . **A-18**
The roll argument to %roll must be a nonempty vector of numbers or ranges **A-18**
The second value in a Range must be greater than the first value . . **A-18**
The specified index (index) was out of the range 0 - number-of-elements - 1 . . . . . **A-18**
The STRINGOF built-in function expects a vector of numbers as its argument **A-19**

The SYSNAME built-in function expects an input string of the form xxx/yyy **A-19**
The threshold on a %roll statement must be a single number **A-19**
The use of feature is being deprecated and will not be supported in future versions of TLC. See the TLC manual for alternatives. . . . **A-19**
The WILL_ROLL built in function expects a range vector and an integer threshold **A-19**
There are no more free contexts. Use tlc('close', HANDLE) to free up a context **A-19**
There was no type associated with the given block for GENERATE **A-19**
This assignment would overwrite an identifier-value pair from the RTW file. To avoid this error either qualify the left-hand side, or choose another identifier. **A-20**
TLC has leaked number symbols. You may have created a cyclic record. If this not the case then please report this leak to The MathWorks. **A-20**
Unable to find identifier within the scope-identifier scope **A-20**
Unable to open %include file filename **A-20**
Unable to open block template file filename from GENERATE or GENERATE_TYPE **A-20**
Unable to open output file filename **A-21**
Undefined identifier identifier_name **A-21**
Unknown type type in CAST expression **A-21**
Unrecognized command line switch passed to string: switch **A-21**
Unrecognized directive directive-name seen **A-21**
Unrecognized type output-type for function **A-22**
Unterminated multiline comment. **A-22**
Unterminated string **A-22**
Usage: tlc [options] file **A-22**
Use of feature incurs a performance hit, please see TLC manual for possible workarounds. **A-22**
Value of type type cannot be compared **A-22**
Values of specified_type type cannot be expanded **A-22**
Values of type Special, Macro Expansion, Function, File, Full Identifier, and Index cannot be converted to MATLAB variables **A-23**
When appending to a buffer stream, the variable must be a string . **A-23**
TLC Function Library Error Messages **A-23**

23 Guidelines and Standards for Embedded Coder

Support for Standards and Guidelines **23-2**

Modeling Guidelines for Generated Code **23-5**
Code Generation Modeling Guidelines **23-5**

Industry Standards **23-7**
IEC 61508: 2010 . . . . . **23-7**
ISO 26262: 2018 . . . . . **23-7**
IEC 62304: 2015 . . . . . **23-7**
EN 50128 / EN 50128: 2011 . . . . . **23-8**

ISO 25119 Standard 23-8
DO-178C Standard 23-8
AUTOSAR Standard 23-9

Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards 23-11
High-Integrity System Modeling Guidelines 23-11
MAB Modeling Guidelines 23-11
JMAAB Modeling Guidelines 23-11

Qualification of Embedded Coder and AUTOSAR Blockset for Use in Processes that Must Comply with Industry Standards 23-13
Validate Traceability 23-13

Coding Standards 23-14
MISRA C 23-14
Secure Coding 23-14

Model Advisor Checks for Verifying Compliance with Secure Coding Standards 23-16

Develop a Model That Complies with the AUTOSAR Standard . . . 23-17

Develop a Model That Complies with the IEC 61508 Standard . . . 23-24

CERT C Compliance Summary Tables 23-26
CERT C Code Compliance Evaluation Summary 23-27
Explanatory Notes 23-36

24 MISRA C:2012 Compliance and Deviations for Code Generated by Using Embedded Coder

Developing a MISRA C:2012 Compliance Statement 24-2
MathWorks Process for Identifying Violations of MISRA C:2012 Guidelines in Generated C Code 24-2

Evaluate Your Generated Code for MISRA C:2012 Compliance 24-4

MISRA C:2012 Compliance Information Summary Tables 24-6
MISRA C:2012 Main Document 24-6
MISRA C:2012 Amendment 1 24-15
MISRA C:2012 Amendment 2 24-16
Explanatory Notes 24-16

Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012 24-20
MISRA C:2012 Model Advisor Checks Rationale 24-20
High-Integrity System Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012 24-24

Deviations Rationale for MISRA C:2012 Compliance **24-27**
MISRA C:2012 Directive 4.7 Deviation Rationale **24-27**
MISRA C:2012 Rule 13.5 Deviation Rationale **24-30**

25 Modeling in Simulink Coder

Use Products and Blocks Supported for Code Generation **25-2**
Related Products **25-2**
Blocks Supported for Code Generation **25-4**
Block Set Support for Code Generation **25-4**

Consider Modeling Semantics when Generating Code **25-5**
Data Propagation **25-5**
Sample Time Propagation **25-6**
Latches for Subsystem Blocks **25-7**
Block Execution Order **25-8**
Algebraic Loops **25-9**

Modeling Guidelines for Blocks **25-11**

Modeling Guidelines for Subsystems **25-12**

Modeling Guidelines for Charts **25-13**

Modeling Guidelines for MATLAB Functions **25-14**

Modeling Guidelines for Model Configuration **25-15**

26 Patterns for C Code in Embedded Coder

Prepare a Model for Code Generation **26-2**
Configure Input Ports, Output Ports, and Arbitrary Signals **26-2**
Initialize and Configure States **26-2**
Set Up Configuration Parameters for Code Generation **26-2**
Set Up an Example Model With a Stateflow Chart **26-3**
Set Up an Example Model With a MATLAB Function Block **26-4**

Definition, Initialization, and Declaration of Parameter Data **26-5**

Definition and Declaration of Signal Data **26-7**

Data Type Conversion **26-9**

Type Qualifiers **26-12**

Relational and Logical Operators **26-14**

Bitwise Operations **26-17**

Enumeration **26-20**

If-Else **26-23**

Switch **26-27**

For Loop **26-31**

While Loop **26-36**

Do While Loop **26-41**

Function Call **26-45**

Function Prototyping **26-47**

External C Functions **26-50**

Macro Definitions (#define) **26-56**

Conditional Inclusions (#if / #endif) **26-58**

Typedef **26-62**

Structures of Parameters **26-64**

Structures of Signals **26-67**

Nested Structures of Signals **26-70**

Bitfields **26-74**

Arrays for Parameters **26-76**

Arrays for Signals **26-78**

Pointers **26-80**

27 Variant Systems in Embedded Coder

Implement Symbolic Dimensions for Array Sizes in Generated Code **27-2**

Dimension Variants **27-2**

Set Parameter Value Based on Variant Choice **27-9**

Code Generation Optimization Considerations **27-10**

Backward Compatibility **27-11**

Supported Blocks **27-11**

Limitations **27-12**

Code Generation for Variant Blocks **27-15**
Restrictions on Variant Subsystem Code Generation **27-15**
Generated Code Components Not Compiled Conditionally **27-16**
Code Generation for Variant Blocks with One Variant Choice **27-17**
Global Data Guarding: Signals and States **27-18**
Global Data Guarding: Parameters **27-19**
Guarding Reference Model Headers **27-19**
Guarding S-Function Block Headers **27-20**

Represent Subsystem and Variant Models in Generated Code . . . **27-22**
Step 1: Represent Variant Choices in Simulink **27-22**
Step 2: Specify Conditions That Control Variant Choice Selection **27-25**
Step 3: Configure Model for Generating Preprocessor Conditionals **27-27**
Step 4: Review Generated Code **27-28**
Limitations **27-31**

Generate Preprocessor Conditionals for Variant Systems **27-32**
Define Variant Controls **27-32**
Configure Model for Generating Preprocessor Conditional Directives **27-33**
Special Considerations for Generating Preprocessor Conditionals **27-33**
Generate Variant Control Macros in Same Header File **27-34**

Represent Variant Source and Sink Blocks in Generated Code . . . **27-37**
Represent Variant Source and Variant Sink Blocks in Simulink . . . **27-37**
Specify Conditions That Control Variant Choice Selection **27-40**
Review the Generated Code **27-41**
Generate Code with Zero Active Variant Controls **27-43**

Configure Symbolic Dimensions for S-Function Blocks **27-45**
Generate Code from an S-Function Block with Dimension Variants **27-46**
S-Function That Supports Forward and Backward Propagation of Symbolic Dimensions **27-51**

Options to Represent Variant Parameters in Generated Code **27-53**
Overview of Variant Parameters **27-53**
Choice of Values in Generated Code **27-53**
Types of Variables in Generated Code **27-54**
Storage Classes for Different Variant Activation Times **27-54**
Symbolic Dimensions in Generated Code **27-55**
Group Variant Parameter Values in Generated Code **27-56**
Generate C Code for Variant Parameters **27-56**

Generate Code for Variant Subsystem with Child Subsystems of Different Output Signal Dimensions **27-62**
Example Model **27-62**
Simulate Model **27-62**
Generate Code **27-63**

Use Variant Subsystem to Generate Code That Uses C Preprocessor Conditionals **27-65**

Use Variant Models to Generate Code That Uses C Preprocessor Conditionals **27-70**

Compile Code Conditionally for All Values of Variant Parameters with Same and Different Dimensions **27-76**

28 Timers in Simulink Coder

Absolute and Elapsed Time Computation **28-2**
About Timers **28-2**
Timers for Periodic and Asynchronous Tasks **28-2**
Allocation of Timers **28-2**
Integer Timers in Generated Code **28-3**
Elapsed Time Counters in Triggered Subsystems **28-3**

Access Timers Programmatically **28-4**
About Timer APIs **28-4**
C API for S-Functions **28-4**
TLC API for Code Generation **28-6**

Specify Clock Resolution Used by Target Environment Clock **28-7**

Generate Code for an Elapsed Time Counter **28-11**

Absolute Time Limitations **28-14**

29 Time-Based Scheduling in Simulink Coder

Time-Based Scheduling and Code Generation **29-2**
Sample Time Considerations **29-2**
Tasking Modes **29-2**
Model Execution and Rate Transitions **29-3**
Execution During Simulink Model Simulation **29-4**
Model Execution in Real Time **29-4**
Single-Tasking Versus Multitasking Operation **29-4**

Modeling for Single-Tasking Execution **29-6**
Single-Tasking Mode **29-6**
Build a Program for Single-Tasking Execution **29-6**
Single-Tasking Execution **29-6**

Modeling for Multitasking Execution **29-9**
Multitasking and Pseudomultitasking Modes **29-9**
Build a Program for Multitasking Execution **29-11**
Execute Multitasking Models **29-11**
Multitasking Execution **29-12**

Handle Rate Transitions **29-16**
Rate Transitions **29-16**
Data Transfer Problems **29-17**
Data Transfer Assumptions **29-17**
Rate Transition Block Options **29-18**
Automatic Rate Transition **29-19**
Visualize Inserted Rate Transition Blocks **29-20**
Periodic Sample Rate Transitions **29-22**
Protect Data Integrity with volatile Keyword **29-26**
Separate Rate Transition Block Code and Data from Algorithm Code and Data **29-26**

Configure Time-Based Scheduling **29-33**
Configure Start and Stop Times **29-33**
Configure the Solver Type **29-33**
Configure the Tasking Mode **29-33**

Time-Based Scheduling Example Models **29-35**
Optimize Memory Usage for Time Counters **29-35**
Single-Rate Modeling (Bare Board, No OS) **29-40**
Multirate Modeling in Single-Tasking Mode (Bare Board, No OS) . **29-40**
Multirate Modeling in Multitasking Mode (Bare Board, No OS) . . . **29-41**
Multirate Modeling in Multitasking Mode (VxWorks® OS) **29-42**
Trade Determinism and Data Integrity to Improve System Performance **29-43**

30 Event-Based Scheduling in Simulink Coder

Asynchronous Events **30-2**
Asynchronous Support **30-2**
Block Library for Calls to an Example Real-Time Operating System **30-2**
Examples and Additional Information **30-3**

Generate Interrupt Service Routines **30-4**
Connecting the Async Interrupt Block **30-4**
Requirements and Restrictions **30-4**
Performance Considerations **30-5**
Dual-Model Approach for Developing Real-Time Systems that Include ISRs **30-5**
Dual-Model Approach: Simulation **30-6**
Dual-Model Approach: Code Generation **30-6**

Spawn and Synchronize Execution of RTOS Task **30-11**

Pass Asynchronous Events in RTOS as Input to a Referenced Model **30-28**

Rate Transitions and Asynchronous Blocks **30-34**
About Rate Transitions and Asynchronous Blocks **30-34**
Handle Rate Transitions for Asynchronous Tasks **30-35**

Handle Multiple Asynchronous Interrupts 30-35
Protect Data Integrity with volatile Keyword 30-37

Timers in Asynchronous Tasks 30-38

Create a Customized Asynchronous Library 30-40
About Implementing Asynchronous Blocks 30-40
Async Interrupt Block Implementation 30-40
Task Sync Block Implementation 30-43
asynclib.tlc Support Library 30-45

Import Asynchronous Event Data for Simulation 30-47
Capabilities 30-47
Input Data Format 30-47
Example 30-47

Asynchronous Support Limitations 30-50
Asynchronous Task Priority 30-50
Convert an Asynchronous Subsystem into a Model Reference 30-50

31 Scheduling Considerations in Embedded Coder

Use Discrete and Continuous Time 31-2
Support for Discrete and Continuous Time Blocks 31-2
Support for Continuous Solvers 31-2
Support for Stop Time 31-2

Optimize Multirate Multitasking Execution for RTOS Run-Time Environments 31-4
Use rtmStepTask 31-4
Schedule Code for Real-time Model without an RTOS 31-4
Schedule Code for Multirate Multitasking on an RTOS 31-5
Suppress Redundant Scheduling Calls 31-5

32 Representing a Software Architecture by Creating Code Generation Definitions

Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture 32-2
Create Code Definitions for Use in the Code Mappings Editor 32-2
Create Code Definitions for External Data Objects 32-2
Use Package Code Definitions in Embedded Coder Dictionaries . . . 32-3
Constrain Use of Storage Class Code Mappings 32-3
Deploy Code Generation Definitions to Users 32-3
Interact with Code Definitions in Data Interface Configuration Programmatically 32-3

Create Data Interface Configuration Programmatically **32-5**

Create a Service Interface Configuration **32-11**
Identify Software Platform Requirements **32-11**
Create a Service Interface Configuration in a Standalone Embedded Coder Dictionary **32-12**
Create Code Definitions for the Service Interface Configuration . . **32-13**
Test the Service Interface Configuration **32-14**
Deploy Code Interface Configuration Definition to Models **32-14**

Deploy Code Generation Definitions **32-16**
Share Code Generation Definitions Between Multiple Models and Users **32-16**
Dictionary Usage for Models Created with Different Versions of Simulink **32-16**
Configure Default Code Mapping in a Shared Dictionary **32-16**
Share Code Interface Configuration Between Models **32-19**
Migrate Data Interface Configuration from Model File to Shared Data Dictionary **32-21**

Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models **32-24**
Share Common Code Generation Settings among Multiple Models **32-24**
Copy Code Generation Settings to New Models **32-25**

Manage an Embedded Coder Dictionary **32-27**
Share Available Code Definitions with New Models **32-27**
Migrate Package Definitions to Embedded Coder Dictionary to Use in Code Mappings **32-27**
Change Code Generation Definitions **32-28**

Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings **32-31**
Effects of Migration **32-31**
Considerations Before Migrating **32-31**
Considerations After Migrating **32-32**

Flexible Storage Class for Different Model Hierarchy Contexts . . **32-34**

Separate C Code Interface Configuration Definition from Model-Specific Configurations **32-38**
Set Up Models to Use Standard Code Interface Configuration . . . **32-38**
Apply Code Interface Definitions to Model Elements with Flexibility **32-39**
Generate Code for Different Platforms **32-40**

Data, Function, and File Definition

33 Configuring Data and Functions in the Generated Code

C Code Generation Configuration for Model Interface Elements **33-2**
Migration **33-2**
Code Definitions **33-3**
Data Default Configurations **33-3**
Function Default Configurations **33-6**
Configurations for Individual Data Elements and Functions **33-8**
Configure Data **33-9**
Configure Functions **33-10**

Code Definition and Mapping Limitations and Considerations . . . **33-11**
Limitations **33-11**
Default Code Mapping Constraints **33-11**
Unresolved Code Definitions **33-12**
Model Editing Operations and Individual Data Element Code Mappings **33-12**
Storage Classes and Reentrant, Multi-Instance Models and Components **33-13**
Copy Code Mappings When Converting Subsystems to Referenced Models **33-14**

Configure Default C Code Generation for Categories of Data Elements and Functions **33-15**
Set Up Example Environment **33-15**
Configure Default Code Generation for Data **33-16**
Configure Default Code Generation for Functions **33-20**
Configure Model Elements for Code Generation with Definitions Stored in Shared Data Dictionary **33-22**
Configure Default Data and Function Code Generation Programmatically **33-24**

Configure Root-Level Inport Blocks for C Code Generation **33-30**
Choose Customization Options for Inport Interface **33-30**
Configure Default Code Generation Settings for Root-Level Inports **33-31**
Configure Code Generation Settings for Individual Root-Level Inports **33-33**
Configure Code Generation Settings for Root-Level Inports Programmatically **33-35**
Choose Storage Class and Storage Class Properties for Root-Level Inports **33-38**

Configure Root-Level Outport Blocks for C Code Generation **33-41**
Choose Customization Options for Outport Interface **33-41**
Configure Default Code Generation Settings for Root-Level Outports **33-42**

Configure Code Generation Settings for Individual Root-Level Outports 33-44
Configure Code Generation Settings for Root-Level Outports Programmatically 33-45
Choose Storage Class and Storage Class Properties for Root-Level Outports 33-47

Configure Signal Data for C Code Generation **33-50**
Choose Code Configuration Options for Signals 33-50
Add Signals to Model Code Mappings 33-51
Configure Default Code Generation Settings for Signals 33-53
Configure Code Generation Settings for Individual Signals 33-54
Remove Signals from Model Code Mappings 33-56
Configure Code Generation Settings for Signals Programmatically 33-56
Choose Storage Class and Storage Class Properties for Signals . . 33-58

Configure Parameters for C Code Generation **33-61**
Choose Customization Options for Parameters 33-62
Configure Default Code Generation Settings for Parameters 33-63
Configure Code Generation for Individual Parameters 33-65
Organize Parameter Data into a Structure by Using Struct Storage Class 33-67
Configure Code Generation Settings for Parameters Programmatically 33-69
Choose Storage Class and Storage Class Properties for Model Parameters 33-71

Configure Block States for C Code Generation **33-74**
Choose Code Configuration Options for States 33-74
Configure Default Code Generation Settings for States 33-75
Configure Code Generation Settings for Individual States 33-76
Configure Code Generation Settings for States Programmatically 33-78
Choose Storage Class and Storage Class Properties for States . . . 33-79

Configure Data Stores for C Code Generation **33-82**
Choose Customization Options for Data Stores 33-82
Configure Default Code Generation Settings for Data Stores 33-83
Configure Code Generation for Individual Data Stores 33-85
Configure Code Generation Settings for Data Stores Programmatically 33-87
Choose Storage Class and Storage Class Properties for Data Stores 33-89

Manage Multiple Code Mappings for a Model **33-92**
Configure Alternative Code Mappings by Using Configuration Sets 33-92
Activate Code Mappings in the Simulink Editor 33-93
Create and Activate Code Mappings Programmatically 33-94
Code Mappings and Embedded Coder Dictionary 33-94

Configure Model Data Interface for C Code Generation **33-96**

Migration of Model Data Configurations to Code Mappings **33-103**
Effects of Migration **33-103**
Considerations Before Migrating **33-104**
Copy Blocks and Signal Lines Between Models **33-107**
Switch Between GRT- and ERT-Based System Target Files **33-108**
Workflow Changes **33-109**

Dimension Preservation of Multidimensional Arrays **33-111**
Dimension Preservation in Generated Code **33-111**
Difference in Dimension Preservation Between MATLAB and C . . **33-111**

Configure Generated C Function Interface for Model Entry-Point Functions **33-114**
What Is an Entry-Point Function? **33-114**
Types of Generated Entry-Point Functions **33-114**
Configure Whether Entry-Point Functions Are Reusable **33-117**
Choose an Approach for Configuring Generated C Function Interfaces **33-118**
Configure Default Code Generation Settings for a Function Category **33-118**
Configure Names for Individual C Entry-Point Functions **33-119**
Configure Name and Arguments for Individual Step Functions . . **33-120**
Configure Code Generation Settings for C Entry-Point Functions Programmatically **33-128**
How to Interface with Generated Entry-Point Functions **33-129**
C Function Interface Customization Limitations **33-129**

Configure Multi-Instance Code Generation **33-131**
Configure a Top Model for Multi-Instance Code Generation **33-131**
Configure a Referenced Model for Multi-Instance Code Generation **33-131**

Configure Data Interfaces **33-132**
Configure Default Code Generation for Data **33-132**
Override Default Data Interface for Individual Data Elements . . . **33-132**

Configure Entry-Point Function Interfaces **33-133**
Define Function Customization Templates **33-133**
Configure Default Code Generation for Entry-Point Functions . . . **33-133**
Override Default Code Generation for Entry-Point Functions . . . **33-133**

Configure Internal Data **33-135**
Configure Default Representation of Internal Data **33-135**
Override Default Representation for Individual Internal Data Elements **33-135**

Configure Subsystem Function Packaging and Interface **33-136**

34 Configuring Service Interfaces in the Generated Code

Generate Sender and Receiver C Interface Code for Component Deployment **34-2**

Generate C Data Transfer Service Interface Code for Component Deployment **34-6**

Generate C Timer Service Interface Code for Component Deployment **34-13**

Generate C Parameter Tuning Service Interface Code for Component Deployment **34-20**

Generate C Measurement Service Interface Code for Component Deployment **34-24**

35 Configuring Service Interfaces in the Generated Code

Configure Sender and Receiver Service Interfaces for Model Inports and Outports **35-2**
- Configure Sender and Receiver Service Interfaces by Using the Code Mappings Editor **35-2**
- Configure Sender and Receiver Service Interfaces Programmatically **35-4**

Configure Data Transfer Service Interfaces for Data Transfer Signals **35-5**
- Configure Data Transfer Service Interfaces by Using the Code Mappings Editor **35-5**
- Configure Data Transfer Service Interfaces Programmatically **35-7**

Configure Timer Service Interfaces **35-9**
- Configure Timer Service Interfaces by Using the Code Mappings Editor **35-9**
- Configure Timer Service Interfaces Programmatically **35-11**

Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments **35-13**
- Configure Parameter Tuning Service Interfaces by Using the Code Mappings Editor **35-13**
- Configure Parameter Tuning Service Interfaces Programmatically **35-14**

Configure Measurement Service Interfaces for Signals, States, and Data Stores **35-16**
- Configure Measurement Service Interfaces for States by Using the Code Mappings Editor **35-16**

Configure Measurement Service Interfaces for Signals by Using the Code Mappings Editor 35-18
Configure Measurement Service Interfaces for States and Signals Programmatically 35-20

36 Data Representation in Simulink Coder

Preserve Variables in Generated Code 36-2
Access Signal, State, and Parameter Data During Execution 36-2
Configure Data Accessibility 36-15
Limitations 36-16

Access Signal, State, and Parameter Data During Execution 36-17

Data Structures in the Generated Code 36-31
Standard Data Structures 36-31
Real-Time Model Data Structure 36-32

How Generated Code Exchanges Data with an Environment 36-35
Data Interfaces in the Generated Code 36-35
Configure Data Interface 36-42

Control Data and Function Interface in Generated Code 36-44
Control Type Names, Field Names, and Variable Names of Standard I/O Structures (Embedded Coder) 36-44
Control Names of Generated Entry-Point Functions (Embedded Coder) 36-44
Control Data Interface for Nonreentrant Code 36-44
Control Data Interface for Reentrant Code 36-46
Prevent Unintended Changes to the Interface 36-46
Reduce Number of Arguments by Using Structures 36-47
Control Data Types of Arguments 36-48
Promote Data Item to the Interface 36-48

How Generated Code Stores Internal Signal, State, and Parameter Data 36-49
Internal Data in the Generated Code 36-49
Local Variables in the Generated Code 36-65
Appearance of Test Points in the Generated Code 36-65
Appearance of Workspace Variables in the Generated Code 36-65
Promote Internal Data to the Interface 36-66
Control Default Representation of Internal Data (Embedded Coder) 36-67

Choose Data Configuration Approach 36-70
Data Element Categories 36-70
Where to Store Data Elements Used by Your Model 36-71
When to Configure a Data Element for Code Generation 36-71
Configure a Data Element for Code Generation 36-72

Choose Storage Class for Controlling Data Representation in Generated Code **36-75**
Storage Class Properties **36-75**
Default Storage Class **36-75**
Built-in and Predefined Storage Classes **36-75**
Auto **36-77**
Default **36-77**
Model Default **36-77**
Dictionary Default **36-77**
ExportedGlobal **36-78**
ImportedExtern, ImportedExternPointer **36-78**
Bitfield **36-79**
CompilerFlag **36-79**
Const, Volatile, and ConstVolatile **36-79**
Define, ImportedDefine **36-80**
ExportToFile **36-80**
FileScope **36-81**
GetSet **36-81**
ImportFromFile **36-82**
Localizable **36-82**
Reusable **36-83**
Struct **36-83**
MultiInstance **36-83**
ParamStruct, SignalStruct **36-84**
Storage Class Limitations **36-84**

Use Enumerated Data in Generated Code **36-86**
Enumerated Data Types **36-86**
Specify Integer Data Type for Enumeration **36-86**
Customize Enumerated Data Type **36-87**
Control Enumerated Type Implementation in Generated Code . . . **36-91**
Type Casting for Enumerations **36-92**
Enumerated Type Limitations **36-92**

Data Stores in Generated Code **36-94**
About Data Stores **36-94**
Generate Code for Data Store Memory Blocks **36-94**
Storage Classes for Data Store Memory Blocks **36-95**
Data Store Buffering in Generated Code **36-96**
Data Stores Shared by Instances of a Reusable Model **36-99**
Structures in Generated Code Using Data Stores **36-99**

Specify Single-Precision Data Type for Embedded Application . . **36-104**
Use single Data Type as Default for Underspecified Types **36-104**

Tune Phase Parameter of Sine Wave Block During Code Execution **36-107**

Switch Between Output Waveforms During Code Execution for Waveform Generator Block **36-109**

Create Tunable Calibration Parameter in the Generated Code . . **36-112**
Represent Block Parameter as Tunable Global Variable **36-112**
Apply Storage Class When Block Parameter Refers to Numeric MATLAB Variable **36-115**

Create Storage Class That Represents Calibration Parameters (Embedded Coder) 36-115
Initialize Parameter Value From System Constant or Other Macro (Embedded Coder) 36-118
Code Generation Impact of Storage Location for Parameter Objects 36-119
Configure Accessibility of Signal Data 36-120
Programmatic Interfaces for Tuning Parameters 36-120
Set Tunable Parameter Minimum and Maximum Values 36-120
Considerations for Other Modeling Goals 36-121

Limitations for Block Parameter Tunability in Generated Code 36-122
Tunable Expression Limitations 36-122
Linear Block Parameter Tunability 36-123
Parameter Structures 36-124

Code Generation of Parameter Objects With Expression Values 36-125
Considerations and Limitations 36-125
Expression Preservation 36-125

Configure Packaging of Parameter Arguments in Generated Code 36-127
Code Generation Behavior 36-127
Limitations 36-130

Parameter Data Types in the Generated Code 36-131
Significance of Parameter Data Types 36-131
Typecasts Due to Parameter Data Type Mismatches 36-131
Considerations for Other Modeling Patterns 36-132

Generate Efficient Code by Specifying Data Types for Block Parameters 36-135
Eliminate Unnecessary Typecasts and Shifts by Matching Data Types 36-135
Reduce Memory Consumption by Storing Parameter Value in Small Data Type 36-137

Reuse Parameter Data in Different Data Type Contexts 36-143

Organize Data into Structures in Generated Code 36-147
Techniques to Create Structures 36-150
Default Application of Structured Storage Class 36-151
Direct Application of Structured Storage Class 36-152
Nonvirtual Buses and Parameter Structures 36-152
Combine Techniques to Work Around Limitations 36-159
Arrays of Structures 36-160
Structure Padding 36-160
Limitations 36-160

Switch Between Sets of Parameter Values During Simulation and Code Execution 36-161

Design Data Interface by Configuring Inport and Outport Blocks 36-169

Generate Efficient Code for Bus Signals **36-173**
Code Efficiency for Bus Signals **36-173**
Set Bus Diagnostics **36-173**
Optimize Virtual and Nonvirtual Buses **36-174**

Control Signal and State Initialization in the Generated Code . . **36-176**
Signal and State Initialization in the Generated Code **36-176**
Generate Tunable Initial Conditions **36-177**
Generate Tunable Initial Condition Structure for Bus Signal **36-181**

Initialization of Signal, State, and Parameter Data in the Generated Code **36-187**
Static Initialization and Dynamic Initialization **36-187**
Real-World Ground Initialization Requiring Nonzero Bit Pattern . **36-187**
Initialization of Signal and State Data **36-188**
Initialization of Parameter Data **36-189**
Data Initialization in the Generated Code **36-189**
Modeling Goals **36-194**

Optimize Generated Code Using Fixed-Point Data with Simulink, Stateflow, and MATLAB **36-196**

Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box **36-197**
Declare Existing Workspace Variables as Tunable Parameters . . **36-197**
Declare New Tunable Parameters **36-197**
Set Tunable Parameter Code Generation Options **36-198**
Programmatically Declare Workspace Variables as Tunable Parameters **36-198**

Share Data Between Code Generated from Simulink, Stateflow, and MATLAB **36-200**

Air-Fuel Ratio Control System with Fixed-Point Data **36-202**

37 Data Definition and Declaration Management in Embedded Coder

Control Placement of Global Data Definitions and Declarations in Generated Files **37-2**
Organize Data to Support Component-Based, Team-Oriented Model Development **37-5**
Specify Default Placement **37-7**
Override Default Placement for Individual Data Items **37-8**
Prevent Name Clashes by Configuring Data Item as static **37-8**
Code Generation Impact of Storage Location for Parameter Objects **37-8**
Specify Default #include Syntax for Data Header Files **37-8**

Establish Data Ownership in a Model Hierarchy **37-10**

38 Data Types in Embedded Coder

Manage Replacement of Simulink Data Types in Generated Code **38-2**
C99 Data Types in Generated Code **38-2**
Control Names of Simulink Coder Primitive Data Types **38-3**
Control Names of Structure Types **38-10**
Generate Code That Reuses Data Types From External Code **38-10**
Create Data Type Alias in the Generated Code **38-10**
Create a Named Fixed-Point Data Type in the Generated Code . . . **38-15**
Rename Data Type Object **38-18**
Display Signal Data Types on Block Diagram **38-18**
Data Type Replacement Limitations **38-18**

Control File Placement of Custom Data Types **38-21**
Data Scope and Header File **38-21**
Macro Guards **38-22**

Replace and Rename Simulink Coder Data Types to Conform to Coding Standards **38-24**
Inspect External C Code **38-24**
Explore Example Model and Default Generated Code **38-24**
Reuse External Data Type Definitions **38-25**
Create Meaningful Data Type Aliases for Individual Data Items . . **38-26**
Create Single Point of Definition for Primitive Types **38-27**
Permanently Store Data Type Objects **38-28**
Create and Maintain Objects Corresponding to Multiple C typedef Statements **38-28**

Exchange Structured and Enumerated Data Between Generated and External Code **38-30**
Inspect External Code **38-30**
Create Simulink Model **38-31**
Configure Generated Code to Write Output Data to Existing Structure Variable **38-32**
Configure Generated Code to Define Parameter Data **38-34**
Generate, Compile, and Inspect Code **38-35**
Replace Data Type Names Throughout Model **38-35**

Specify Boolean and Data Type Limit Identifiers **38-37**
Data Type Limit Identifiers **38-37**
Boolean Identifiers **38-37**
Boolean and Data Type Limit Identifier Header Files **38-38**

39 Module Packaging Tool (MPT) Data Objects in Embedded Coder

MPT Data Object Properties **39-2**
Specify Persistence Level for Signals and Parameters **39-10**

40 Storage Classes in Embedded Coder

Reuse Parameter Data from External Code in the Generated Code **40-2**

Import Parameter Data with Conditionally Compiled Dimension Length **40-6**

Access Structured Data Through a Pointer That External Code Defines **40-10**

Create Storage Classes by Using the Custom Storage Class Designer **40-14**
- Create and Apply a Storage Class **40-14**
- Modify a Built-In Storage Class **40-17**
- Control Data Representation by Configuring Storage Class Properties **40-17**
- Avoid Errors During Code Generation by Validating Storage Class Configuration **40-20**
- Make Storage Classes Available Outside the Current Folder **40-20**
- Make Storage Classes Available in the Code Mappings Editor **40-21**
- Share Storage Class Between Packages **40-21**
- Control Appearance of Storage Class Drop-Down List **40-21**
- Protect Custom Storage Class Definitions **40-22**
- Further Customize Generated Code by Writing TLC Code **40-22**

Finely Control Data Representation by Writing TLC Code for a Storage Class **40-23**
- Create Custom Attributes Class for Storage Class **40-23**
- Write TLC Code for Storage Class **40-23**
- Create Storage Class by Using Custom Storage Class Designer . . **40-23**

Access Data Through Functions with Storage Class GetSet **40-25**
- Access Legacy Data Using Get and Set Functions **40-25**
- Use GetSet with Vector Data **40-28**
- Use GetSet with Structured Data **40-31**
- Use GetSet with Matrix Data **40-35**
- Specify Header File or Function Naming Scheme for Data Items . **40-39**
- Access Scalar and Array Data Through Macro Instead of Function Call **40-40**
- GetSet Storage Class Restrictions **40-40**

Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary **40-41**
- Open Embedded Coder Dictionary **40-41**
- Access Legacy Data by Value **40-42**
- Access Legacy Data by Pointer **40-44**

Integrate External Application Code with Code Generated from PID Controller **40-45**

Configure Generated Code According to Interface Control Document Specifications **40-54**

Generate Local Variables with Localizable Storage Class **40-62**
Example Model **40-62**
Generate Code with Localizable Storage Class **40-62**
Generate Code Without Localizable Storage Class **40-63**
Additional Information **40-64**

41 Data Object Wizard in Embedded Coder

Create Data Objects for Code Generation with Data Object Wizard **41-2**

42 Entry-Point Functions and Scheduling in Simulink Coder

Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks **42-2**
Configure Generated C Function Interface for Global Simulink Function Block **42-2**
Configure Generated C Function Interface for Exported Scoped Simulink Function Block **42-6**
Configure Generated C++ Function Interface for Exported Scoped Simulink Function Block **42-10**
Simulink Function Code Interface Limitations **42-14**

Generate C++ Class Interface to Model or Subsystem Code **42-15**
Generate C++ Class Interface to Model Code **42-15**
Generate C++ Class Interface to Nonvirtual Subsystem Code . . . **42-16**
C++ Class Interface Limitations **42-16**

Execution of Code Generated from a Model **42-18**
Program Execution **42-18**
Program Timing **42-19**
External Mode Communication **42-19**
Data Logging in Single-Tasking and Multitasking Model Execution **42-20**
Non-Real-Time Single-Tasking Systems **42-20**
Non-Real-Time Multitasking Systems **42-21**
Real-Time Single-Tasking Systems **42-22**
Real-Time Multitasking Systems **42-23**
Multitasking Systems Using Real-Time Tasking Primitives **42-24**
Rapid Prototyping and Embedded Model Execution Differences . . **42-25**

Rapid Prototyping Model Functions **42-27**

43 Function and Class Interfaces in Embedded Coder

Generate Modular Function Code for Nonvirtual Subsystems **43-2**
About Nonvirtual Subsystem Code Generation **43-2**
Configure Subsystem for Generating Modular Function Code **43-2**
Modular Function Code for Nonvirtual Subsystems **43-5**
Partition Functions in Generated Code **43-8**
Nonvirtual Subsystem Modular Function Code Limitations **43-17**

Generate Reentrant, Multi-Instance Code **43-19**

Generate Code for String Blocks by Using the Standard C++ String Library **43-22**

Select Code Generation Output for Target Platform Deployment **43-25**
Example Model Hierarchy **43-25**
Select Output for Simulation Harness **43-25**
Specify Code Generation Output **43-26**
Select C Code Interface Configuration **43-27**

Configure C Code Deployment Types for Model Hierarchy **43-29**
Set Up Test Harness and Model Hierarchy **43-29**
Specify Deployment Types for the Models **43-30**
Generate and Inspect Code **43-32**

44 Configure C++ Class Interface

Interactively Configure C++ Interface **44-2**
Interactive Workflow Overview **44-2**
Open Environment **44-2**
Configure Model as Class **44-4**
Configure Model Data Elements as Class Members **44-5**
Configure Model Functions as Class Methods **44-15**
Generate C++ Class Interface **44-22**
Considerations and Limitations **44-23**

Programmatically Configure C++ Interface **44-25**
Programmatic Workflow Overview **44-25**
Open Environment **44-25**
Configure Model as Class **44-27**
Configure Model Data Elements as Class Members **44-28**
Configure Model Functions as Class Methods **44-38**
Generate C++ Class Interface **44-43**
Considerations and Limitations **44-44**

Configure C++ Class Interface for Rate-Based Models **44-46**

Configure C++ Class Interface for Export-Function Models **44-50**

45 Memory Sections in Embedded Coder

Control Data and Function Placement in Memory by Inserting Pragmas **45-2**
Insert Pragmas by Using Memory Sections **45-2**
Configure Pragma to Surround Groups of Definitions **45-6**
Override Default Memory Placement for Individual Data Elements **45-6**
Choose Where to Create and Store Memory Section Definition **45-7**
Share Memory Section Definition Between Models **45-7**
Override Default Memory Placement for Subsystem Functions and Data **45-7**
Share Memory Section Between Packages (Package Memory Sections Only) **45-9**
Control Appearance of Memory Section Drop-Down List (Package Memory Sections Only) **45-10**
Protect Definitions of Package Memory Sections (Package Memory Sections Only) **45-10**
Limitations **45-10**
Insert Pragmas for Functions and Data in Generated Code **45-11**

Protect Global Data with const and volatile Type Qualifiers **45-13**
Maintain const Correctness for Arguments of Entry-Point Functions **45-13**
Incorrect Results or Undefined Behavior When Passing Volatile Data to a Generated Function **45-13**

46 Array Layout

Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks **46-2**

Interpolation Algorithm for Row-Major Array Layout **46-10**

Interpolation with Subtable Selection Algorithm for Row-Major Array Layout **46-16**

Direct Lookup Table Algorithm for Row-Major Array Layout **46-22**

Generate Row-Major Code for S-Functions **46-29**

Row-Major Algorithms for Row-Major Array Layout **46-32**

Generate Row-Major Code for Model That Contains a MATLAB Function Block **46-35**
Row-Major Code Generation **46-35**
Mixed-Majority Code Generation **46-36**

47 Preserve Dimensions

Preserve Dimensions of Multidimensional Arrays in Generated Code **47-2**

Preserve Dimensions of Bus Elements in Generated Code **47-6**

Code Generation

48 Configuration for Simulink Coder

Code Generation Configuration **48-2**
Open the Model Configuration for Code Generation **48-2**
Configuration Tools **48-2**

Configure Code Generation Parameters for Model Programmatically **48-4**
Modify Parameters to Support Execution efficiency **48-4**

Configure Model from Command Line **48-6**

Use Configuration Reference to Select Code Generation Target . . **48-11**

Check Model and Configuration for Code Generation **48-16**
Check Model for Code Efficiency with Model Advisor **48-16**
Check Model During Code Generation by using the Code Generation Advisor **48-16**

Application Objectives Using Code Generation Advisor **48-18**
High-Level Code Generation Objectives **48-18**
Configure Model for Code Generation Objectives Using Code Generation Advisor **48-19**
Configure Model for Code Generation Objectives by Using Configuration Parameters Dialog Box **48-19**

Simulink Coder Model Advisor Checks for Standards and Code Efficiency **48-21**

Configure Code Comments **48-22**

Include MATLAB Code as Comments in Generated Code **48-23**
Include MATLAB Code as Comments in the Generated Code **48-23**
Location of Comments in Generated Code **48-23**
Include MATLAB User Comments in Generated Code **48-25**
Limitations of MATLAB Source Code as Comments **48-26**

Construction of Generated Identifiers **48-27**

Identifier Name Collisions and Mangling **48-28**
Identifier Name Collisions with Referenced Models **48-28**

Specify Identifier Length to Avoid Naming Collisions **48-29**

Specify Reserved Names for Generated Identifiers **48-30**

Reserved Keywords **48-31**
C Reserved Keywords **48-31**
C++ Reserved Keywords **48-31**
Other Common Reserved Keywords **48-32**
Code Generation Code Replacement Library Keywords **48-32**

Configure Model for Debugging **48-34**

49 Configuration in Embedded Coder

Configure Model for Code Generation Objectives by Using Code Generation Advisor **49-2**
High-Level Code Generation Objectives **49-2**
Specify Objectives in Referenced Models **49-3**
Configure Model Using Code Generation Advisor **49-3**
Configure Model for Code Generation Objectives by Using Configuration Parameters Dialog Box **49-4**

Configure Code Generation Objectives Programmatically **49-7**

Check Model and Configuration for Code Generation **49-8**
Check Model During Code Generation by Using the Code Generation Advisor **49-8**
Embedded Coder Model Advisor Checks for Standards, Guidelines, and Code Efficiency **49-9**

Create Custom Code Generation Objectives **49-10**
Specify Parameters in Custom Objectives **49-10**
Specify Checks in Custom Objectives **49-10**
Determine Checks and Parameters in Existing Objectives **49-11**
Steps to Create Custom Objectives **49-12**

Configuration Variations **49-14**

Configure and Optimize Model with Configuration Wizard Blocks **49-15**
Configuration Wizard Block Library **49-15**
Add a Configuration Wizard Block **49-16**
Use Configuration Wizard Blocks to Configure Your Model **49-17**
Create a Custom Configuration Wizard Block **49-17**

Aircraft Position Radar Model **49-22**

50 System Target File Configuration

Configure a System Target File **50-2**
Select a Solver That Supports Code Generation **50-2**
Select a System Target File from STF Browser **50-2**
Select a System Target File Programmatically **50-3**
Develop Custom System Target Files **50-4**

Configure STF-Related Code Generation Parameters **50-5**
Specify Generated Code Interfaces **50-5**
Configure Numeric Data Support **50-8**
Configure Time Value Support **50-8**
Configure Noninlined S-Function Support **50-9**
Configure Model Function Generation and Argument Passing **50-9**
Configure Code Reuse Support **50-10**

Configure Standard Math Library for Target System **50-12**

Compare System Target File Support Across Products **50-17**
Compare Product System Target Files **50-18**
Compare Code Styles and STF Support **50-20**
Compare Generated Code Features by Product **50-21**
Compare Generated Code Features by STF **50-23**

51 Internationalization Support in Embedded Coder

Internationalization and Code Generation **51-2**
Locale Settings **51-2**
Prepare to Generate Code for Mixed Languages and Locales **51-2**
Character Set Limitations **51-2**
XML Escape Sequence Replacements **51-3**
CGT Files and XML Escape Sequence Replacements **51-3**
Generate and Review Code with Mixed Languages and Mixed Locales **51-4**

52 Source Code Generation in Simulink Coder

Configure Model, Generate Code, and Simulate **52-2**
About This Example **52-2**
Functional Design of the Model **52-2**
View the Top Model **52-2**
View the Subsystems **52-3**
Simulation Test Environment **52-4**
Run Simulation Tests **52-8**
Key Points **52-9**

Learn More 52-10

Configure Model and Generate Code 52-11
About This Example 52-11
Configure the Model for Code Generation 52-11
Save Your Model Configuration as a MATLAB Function 52-12
Check Model Conditions and Configuration Settings 52-13
Generate Code for the Model 52-13
Review the Generated Code 52-13
Generate an Executable 52-14
Key Points 52-14

Configure Data Interface 52-16
About This Example 52-16
Declare Data 52-16
Use Data Objects 52-17
Add New Data Objects 52-19
Enable Data Objects for Generated Code 52-19
Effects of Simulation on Data Typing 52-20
Manage Data 52-21
Key Points 52-22

Call External C Functions 52-23
About This Example 52-23
Include External C Functions in a Model 52-23
Create a Block That Calls a C Function 52-23
Validate External Code in the Simulink Environment 52-25
Validate C Code as Part of a Model 52-26
Call a C Function from Generated Code 52-27
Key Points 52-27

Reload Generated Code 52-29

Manage Build Process Folders 52-30
File Generation Control Parameters 52-30
Build Process Folders 52-31

Manage Build Process Files 52-35
model.bat 52-39
model.h 52-39
rtwtypes.h 52-40

Manage Build Process File Dependencies 52-43
System Header Files 52-44
Code Generator Header Files 52-46

Add Build Process Dependencies 52-52
File Dependency Information for the Build Process 52-52
Folder Dependency Information for the Build Process 52-54

Build Process Support for File and Folder Names 52-57
Filenames with Spaces 52-57
Folder Names with Spaces 52-57
Troubleshooting Errors When Folder Names Have Spaces 52-60
Folder Names with Special Characters 52-61

Very Long Folder Paths 52-61

Code Generation of Matrices and Arrays 52-63
Array Storage in Computer Memory 52-63
Code Generator Matrix Parameters 52-65
Internal Data Storage for Complex Number Arrays 52-67
Unsupported Blocks for Row-Major Code Generation 52-67
Limitations for Row-Major Code Generation 52-68

Cross-Release Shared Utility Code Reuse 52-69
Workflow to Reuse Shared Utility Code 52-69
Required Editing for Shared Utility Code Reuse 52-70

Cross-Release Code Integration 52-71
Workflow 52-71
Limitations 52-73
Simulink.Bus Support 52-74
Root-Level I/O Through Global Variables in Generated Code 52-75
Communicate Between Current and Previous Release Components Through Global Data Stores 52-77
Parameter Tuning 52-78
Use Multiple Instances of Code Generated from Reusable Referenced Model 52-79
Compare Simulation Behavior of Model Component in Current Release and Generated Code from Previous Release 52-80
Import AUTOSAR Code from Previous Releases 52-81

Integration of Code from Multiple Folders 52-82

Integrate Generated Code by Using Cross-Release Workflow 52-85

Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code 52-93

Compile Code in Another Development Environment 52-99

Generate Code and Simulate Models in a Project 52-103

Generate Code and Simulate Models Programmatically with Project 52-106

53 Source Code Generation in Embedded Coder

Generate Code Using Embedded Coder 53-2

Generate Code by Using the Quick Start Tool 53-9
Quick Start Model Analysis 53-9
Configuration Parameter Changes for Models That Have a Configuration Reference 53-10
Next Steps 53-11

Manage File Packaging of Generated Code Modules **53-12**
Generated Code Modules **53-12**
User-Written Code Modules **53-14**
Customize Generated Code Modules **53-14**

Tutorial for Generating C Code from a Control Algorithm for an Embedded System **53-17**
Third-Party Software **53-17**

Prepare a Control Algorithm Model for C Code Generation **53-18**

Configure Data Interface in the Generated Code **53-29**

Partition Functions in Generated Code **53-39**

Call External C Code from Model and Generated Code **53-48**

Build Integrated Code Outside the Simulink Environment **53-53**

Test Generated Code **53-58**

Evaluate Performance of the Generated Code **53-64**

Install and Use Cygwin and Eclipse **53-68**

54 Report Generation in Embedded Coder

Reports for Code Generation **54-2**
HTML Code Generation Report Content **54-2**
HTML Code Generation Report Location **54-2**
HTML Code Generation Report for Referenced Models **54-2**
HTML Code Generation Report Extensions **54-3**

Generate Code Generation Report **54-5**
Configure Model **54-5**
Generate Report **54-5**
Share Report **54-5**
Limitation **54-6**

Configure and Generate Code Generation Report Programmatically **54-7**

Web View of Model in Code Generation Report **54-10**
Generate HTML Code Generation Report with Model Web View . . **54-10**
Model Web View Limitations **54-12**

Analyze Generated Data Code Interface **54-13**
Navigating Code Interface Report Subsections **54-13**
Interpreting the Entry-Point Functions Subsection **54-14**
Interpreting the Inports and Outports Subsections **54-17**
Interpreting the Interface Parameters Subsection **54-18**

Interpreting the Data Stores Subsection 54-19
Code Interface Report Limitations 54-19

Analyze Generated Service Code Interface Report 54-21
Navigate Code Interface Report Subsections 54-21
Interpret the Execution Interfaces Section 54-22
Interpret the Service Interfaces Section 54-23
Code Interface Report Limitations 54-24

Static Code Metrics 54-26
Static Code Metrics 54-26
Static Code Metrics Analysis 54-26
View Static Code Metrics and Definitions Within the Generated Code 54-28
Static Code Metrics Report Limitations 54-28

Generate Static Code Metrics Report for Simulink Model 54-29

Generating a Static Code Metrics Report for Code Generated from MATLAB Code 54-33
Example Static Code Metrics Report 54-33
Requirements for Running Static Code Metrics Analysis After Code Generation 54-35
Running Static Code Metrics at Code Generation Time 54-35

Analyze Code Replacements in Generated Code 54-36

Document Generated Code with Simulink Report Generator 54-38
Generate Code for the Model 54-38
Open the Report Generator 54-39
Set Report Name, Location, and Format 54-41
Include Models and Subsystems in a Report 54-42
Customize the Report 54-42
Generate the Report 54-43

Get Code Description of Generated Code 54-44

Get Generated Type and Name of Inport Structures 54-51

Get Metadata About Service Interface 54-54

55 Code Appearance in Embedded Coder

Customize Code Comments to Enhance Readability and Traceability 55-2

Add Custom Comments for Variables in the Generated Code 55-3
Embed Handwritten Comments for Signals or Parameters 55-3
Generate Dynamic Comments Based on Data Properties 55-3

Add Global Comments in the Generated Code **55-5**
Use a Simulink DocBlock to Add a Comment **55-5**
Use a Simulink Annotation to Add a Comment **55-7**
Use Sorted Notes to Add Comments **55-8**
Limitations **55-9**

Customize Generated Identifier Naming Rules **55-10**
Apply Naming Rules to Identifiers Globally **55-10**
Apply Custom Naming Conventions to Identifiers **55-11**

Identifier Format Control **55-17**
$N Embedded Coder Dictionary Resolution **55-21**
Control Case by Using Token Decorators **55-22**
Modify Identifiers by Using Regular Expression Decorators **55-22**
Control Formatting of Identifiers **55-24**

Control Name Mangling in Generated Identifiers **55-25**
Minimize Name Mangling **55-25**

Avoid Identifier Name Collisions with Referenced Models **55-26**
Use Model Advisor to Detect Identifier Names Changed During Code Generation **55-26**

Maintain Traceability for Generated Identifiers **55-27**

Exceptions to Identifier Formatting Conventions **55-28**

Identifier Format Control Parameters Limitations **55-29**

Control Parentheses in Generated Code **55-31**
Control Use of Parentheses **55-31**

Optimize Code by Reordering Commutable Operands **55-33**
Example Model **55-33**
Generate Code **55-33**
Generate Code with Optimization **55-33**

Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements **55-35**
Example **55-35**
Generate Code with Default Cases for Unreachable Stateflow Switch Statements **55-37**
Suppress Default Cases for Unreachable Stateflow Switch Statements **55-37**

Replace Multiplication by Powers of Two with Signed Bitwise Shifts **55-38**
Example **55-38**
Generate Code with Signed Bitwise Shifts **55-38**
Generate Code with Multiplication by Powers of Two **55-38**

Generate Code Containing Right Shifts on Signed Integers **55-40**
Example Model **55-40**
Generate Code Containing Right Shifts on Signed Integers **55-40**

Generate Code That Does Not Allow Right Shifts on Signed Integers 55-40

Control Cast Expressions in Generated Code 55-42
Enable Nominal Casting Mode and Generate Code 55-42
Enable Standards Compliant Casting Mode and Generate Code . . 55-43
Enable Explicit Casting Mode and Generate Code 55-44

Customize Code Organization and Format 55-46
Custom File Processing Components 55-46
Custom File Processing Configuration 55-47

Specify Templates For Code Generation 55-48

Code Generation Template (CGT) Files 55-49
Default CGT file 55-49
CGT File Structure 55-49
Built-In Tokens and Sections 55-50
Subsections 55-51

Format Generated Code Files by Using Templates 55-53

Custom File Processing (CFP) Templates 55-55
Custom File Processing (CFP) Template Structure 55-55
Generate Additional Files Required to Interface with an Application 55-56

Change the Organization of a Generated File 55-58

Customize Generated File Names 55-60

Generate Source and Header Files with a Custom File Processing (CFP) Template 55-62
Generate Code with a CFP Template 55-62
Analysis of the Example CFP Template and Generated Code 55-63
Generate a Custom Section 55-66
Custom Tokens 55-67

Comparison of a Template and Its Generated File 55-68
Template and Generated File 55-68

Code Template API Summary 55-71

Generate Custom File and Function Banners 55-74
Create a Custom File and Function Banner Template 55-74
Customize a Code Generation Template (CGT) File for File and Function Banner Generation 55-75

Template Symbols and Rules 55-81
Introduction 55-81
Template Symbol Groups 55-81
Template Symbols 55-83
Rules for Modifying or Creating a Template 55-85

Annotate Code for Justifying Polyspace Checks 55-86

Enhance Readability of Code for Flow Charts **55-95**
Appearance of Generated Code for Flow Charts **55-95**
Convert If-Elseif-Else Code to Switch-Case Statements **55-97**
Example of Converting Code to Switch-Case Statements **55-99**

Enhance Code Readability for MATLAB Function Blocks **55-105**
Requirements for Using Readability Optimizations **55-105**
Converting If-Elseif-Else Code to Switch-Case Statements **55-105**
Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements **55-107**

Improve Data Coherency in Generated Code **55-111**
Example Model **55-111**
Generate Code Without Parameter Enabled **55-111**
Generate Code With Parameter Enabled **55-112**

56 Code Replacement in Simulink Coder

What Is Code Replacement? **56-2**
Code Replacement Libraries **56-2**
Code Replacement Terminology **56-4**
Code Replacement Limitations **56-5**

Choose a Code Replacement Library **56-6**
About Choosing a Code Replacement Library **56-6**
Explore Available Code Replacement Libraries **56-6**
Explore Code Replacement Library Contents **56-6**

Replace Code Generated from Simulink Models **56-8**

57 Code Replacement for Simulink Models in Embedded Coder

What Is Code Replacement? **57-2**
Code Replacement Libraries **57-2**
Code Replacement Terminology **57-4**
Code Replacement Limitations **57-5**

Replace Code Generated from Simulink Models **57-6**

58 External Code Integration in Simulink Coder

Choose an External Code Integration Workflow **58-2**
Choose a Software Execution Framework for Scheduling Code Execution **58-3**
Evaluate Characteristics of External Code **58-4**
Identify Integration Requirements **58-6**
Choose a Workflow **58-7**

Configure Target Hardware Characteristics **58-9**

Check Code Generation Assumptions **58-12**
Check Code Generator Assumptions for Development Computer . . **58-12**
Coder Assumptions List **58-13**

Call Reusable External Algorithm Code for Simulation and Code Generation **58-15**
Workflow **58-15**
Choose an Integration Approach **58-16**
Insert External Code into Stateflow Charts **58-21**

Place External C/C++ Code in Generated Code **58-23**
Workflow **58-23**
Choose an Integration Approach **58-24**
Integrate External Code by Using Custom Code Blocks **58-25**
Integrate External Code by Using Model Configuration Parameters **58-27**
Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters **58-28**

Call External Device Drivers **58-31**

Apply Function and Operator Code Replacements **58-32**

Build Integrated Code Within the Simulink Environment **58-33**
Workflow **58-33**
Configure Parameters for Integrated Code Build Process **58-33**
Preserve External Code Files in Build Folder **58-34**
Build Support for S-Functions **58-35**

Generate Component Source Code for Export to External Code Base **58-40**
Modeling Options **58-40**
Requirements **58-40**
Limitations for Export-Function Subsystems **58-42**
Workflow **58-42**
Choose an Integration Approach **58-43**
Generate C Function Code for Export-Function Model **58-44**

Generate C++ Function and Class Code for Export-Function Model 58-48
Generate Code for Export-Function Subsystems 58-52

Generate Shared Library for Export to External Code Base **58-54**
About Generated Shared Libraries 58-54
Workflow 58-54
Generate Shared Libraries 58-55
Create Application Code That Uses Generated Shared Libraries . . 58-55
Limitations 58-58

Interface to a Development Computer Simulator by Using a Shared Library **58-59**

Exchange Data Between External C/C++ Code and Simulink Model or Generated Code **58-61**
Import External Code into Model 58-61
Export Generated Code to External Environment 58-62
Simulink Representations of C Data Types and Constructs 58-62
Considerations for Other Modeling Goals 58-67

Exchange Data Between External Calling Code and Generated Code **58-69**
Data Exchange for Reentrant Generated Code 58-69
Data Exchange for Nonreentrant Generated Code 58-69

Generate Code That Matches Appearance of External Code **58-72**

59 Program Building, Interaction, and Debugging in Simulink Coder

Select C or C++ Programming Language **59-2**

Select and Configure C or C++ Compiler **59-3**
Language Standards Compliance 59-3
Programming Language Considerations 59-4
C++ Language Support Limitations 59-4
Code Generator Assumes Wrap on Signed Integer Overflows 59-5
Choose and Configure Compiler 59-5
Include S-Function Source Code 59-6

Troubleshoot Compiler Issues **59-7**
Compiler Version Mismatch Errors 59-7
Results for Model Simulation and Program Execution Differ 59-7
Generates Expected Code and Produces Unexpected Results 59-8
Compile-Time Issues 59-8
LCC Compiler Does Not Support Ampersands in Source Folder Paths 59-9
LCC Compiler Might Not Support Line Lengths of Rapid Accelerator Code 59-9

Approaches for Building Code Generated from Simulink Models **59-11**

Configure CMake Build Process **59-13**
Specify CMake Toolchain Definition **59-13**
Available CMake Toolchain Definitions **59-14**
Generate Only Code and CMake Configuration File for Algorithm Export **59-15**

Create Custom CMake Toolchain Definition **59-17**

Configure Toolchain (ToolchainInfo) or Template Makefile Build Process **59-20**
Toolchain Approach **59-20**
Upgrade Model to Use Toolchain Approach **59-22**
Template Makefile Approach **59-24**
Specify TLC for Code Generation **59-28**
Use of Checksums in Recompilation of Generated Code **59-28**

Define Custom Makefile-Based Toolchains Using Target Framework **59-30**
Overview of Makefile-Based Toolchain Definition **59-30**
Define GCC-Based Toolchain for Windows Development Computer **59-30**
Define Microsoft Visual Studio Toolchain for Building 32-Bit Application **59-32**

Template Makefiles and Make Options **59-36**
Types of Template Makefiles **59-36**
Specify Template Makefile Options **59-36**
Template Makefiles for UNIX Platforms **59-37**
Template Makefiles for the Microsoft Visual C++ Compiler **59-37**
Template Makefiles for the LCC Compiler **59-38**

Build Process Workflow for Real-Time Systems **59-39**
Working Folder **59-39**
Build Folder and Code Generation Folders **59-39**
Set Model Parameters for Code Generation **59-39**
Configure Build Process **59-40**
Set Code Generation Parameters **59-41**
Build and Run a Program **59-42**
Contents of the Build Folder **59-43**
Customized Makefile Generation **59-44**

Build Models from a Windows Command Prompt Window **59-45**

Rebuild a Model **59-48**

Control Regeneration of Top Model Code **59-49**
Regeneration of Top Model Code **59-49**
Force Regeneration of Top Model Code **59-49**

Reduce Build Time for Referenced Models by Using Parallel Builds **59-51**
Build Referenced Models in Parallel **59-51**

Monitor Parallel Building of Referenced Models 59-51

Relocate or Share Generated Code 59-56
Package and Relocate Generated Code 59-56
Package and Relocate Generated Code From Command Line 59-57
Limitations 59-59

Compile and Debug Generated C Code with Microsoft Visual Studio 59-61

Executable Program Generation 59-62

60 Host/Target Communication in Simulink Coder

External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling 60-2
Communication Mechanisms for External Mode 60-3
Simulink Feature Support for External Mode 60-4
External Mode Simulations by Using the Run on Custom Hardware App 60-5
External Mode Control Panel 60-6
Security for External Mode Simulations 60-6

External Mode Simulation by Using XCP Communication 60-8
Run XCP External Mode Simulation on Development Computer . . . 60-8
Triggered Signal Monitoring for XCP External Mode Simulations 60-12
Run XCP External Mode Simulation From Command Line 60-16
Memory Allocation for Communication Buffers During XCP External Mode Simulation 60-20
Target Application Arguments 60-20
XCP External Mode Limitations 60-21

Graphical Controls for XCP External Mode Simulations 60-27
Equivalent Controls on Hardware Tab and External Mode Control Panel 60-27
XCP External Mode Control Panel 60-28
XCP External Signal & Triggering Dialog Box 60-29

Set Up External Mode Connectivity Between Simulink and Target Hardware 60-35
Customise Connectivity for XCP External Mode Simulations 60-36
Customize Connectivity for TCP/IP or Serial External Mode Simulations 60-40
Execution Tool Template 60-41

Register Custom Launcher 60-43

Customize XCP Server Software 60-45
External Mode Abstraction Layer 60-46
XCP Server Protocol Layer 60-48

XCP Server Transport Layer 60-49
XCP Platform Abstraction Layer 60-49

External Mode Simulation with TCP/IP or Serial Communication **60-55**
Create and Configure Model 60-55
Build Target Executable 60-57
Run Target Application 60-58
Tune Parameters 60-58
Stop Target Application 60-61
Control Memory Allocation for Communication Buffers in Target . 60-61
TCP/IP or Serial External Mode Control Panel 60-61
Configure Host Monitoring of Target Application Signal Data 60-64
Configure Host Archiving of Target Application Signal Data 60-70
Summary of Graphical Controls for TCP/IP or Serial External Mode Simulation 60-72
Blocks and Subsystems Compatible with External Mode 60-73
External Mode Mechanism for Downloading Tunable Parameters 60-75
Choose Communication Protocol for Client and Server 60-78
Use External Mode Programmatically 60-84
Animate Stateflow Charts in External Mode 60-88
TCP/IP and Serial External Mode Limitations 60-90

Create a Transport Layer for TCP/IP or Serial External Mode Communication **60-94**
Design of External Mode 60-94
External Mode Communications Overview 60-96
External Mode Source Files 60-97
Implement a Custom Transport Layer 60-100

61 Logging in Simulink Coder

Log Program Execution Results **61-2**
Log Data for Analysis 61-2
Configure State, Time, and Output Logging 61-6
Log Data with Scope and To Workspace Blocks 61-8
Log Data with To File Blocks 61-8
Data Logging Differences Between Single- and Multitasking 61-8

62 Data Interchange Using the C API in Simulink Coder

Exchange Data Between Generated and External Code Using C API **62-2**
Generated C API Files 62-2
Generate C API Files 62-4
Description of C API Files 62-5

Generate C API Data Definition File for Exchanging Data with a Target System 62-16
C API Limitations 62-17

Use C API to Access Model Signals and States **62-19**

Use C API to Access Model Parameters **62-24**

63 ASAP2 Data Measurement and Calibration in Simulink Coder

Export ASAP2 File for Data Measurement and Calibration **63-2**
Generation of ASAP2 File Using Generate Calibration Files Tool . . . **63-2**
Generation of ASAP2 File Using ASAP2Interface Parameter **63-4**

Calibrate Generated Code and Monitor Signals **63-11**

64 Generate ASAP2 and CDF Calibration Files

Generate ASAP2 and CDF Calibration Files **64-2**
Open Generate Calibration Files Tool **64-2**
ASAP2 File Generation **64-3**
CDF File Generation **64-4**
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration **64-4**

Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration **64-6**

65 Customize Generated ASAP2 File

Customize Generated ASAP2 File **65-2**
Customize Computation Method Names **65-2**
Export Characteristics and Measurement Objects in Groups **65-3**
Generate Record Layouts **65-4**
Exclude Structures and 64-Bit Integers **65-4**
Generate ASAP2 Containing Default Event List **65-5**
Specify ECU Address Extension **65-6**
Merge ASAP2 Files **65-6**

Create and Add Custom Data Elements **65-8**

Customize an ASAP2 File 65-9

Customize Using Base Files 65-15
Customize ASAP2 Fields by Using Base Class 65-15
Customize ASAP2 Sections by Deriving from Base Class 65-15

Customize an ASAP2 File 65-17

66 Configure Model Data Elements for ASAP2 File Generation

Configure Model Data Elements for ASAP2 File Generation 66-2
View Calibration Attributes 66-2
Configure Inport for ASAP2 File Generation 66-3
Configure Parameter for ASAP2 File Generation 66-5

67 Direct Memory Access to Generated Code for Simulink Coder

Access Memory in Generated Code Using Global Data Map 67-2

68 Desktops in Simulink Coder

Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File 68-2
About Rapid Simulation 68-2
Rapid Simulation Advantage 68-2
General Rapid Simulation Workflow 68-2
Identify Rapid Simulation Requirements 68-3
Configure Inports to Provide Simulation Source Data 68-4
Configure and Build Model for Rapid Simulation 68-4
Set Up Rapid Simulation Input Data 68-6
Scripts for Batch and Monte Carlo Simulations 68-13
Run Rapid Simulations 68-13
Tune Parameters Interactively During Rapid Simulation 68-21
Rapid Simulation Target Limitations 68-22

Run Rapid Simulations over a Range of Parameter Values 68-24

Run Batch Simulations Without Recompiling Generated Code . . 68-30

Use MAT-Files to Feed Data to Inport Blocks for Rapid Simulations 68-35

Use S-Function Target for Model or Subsystem **68-40**
Required Files for S-Function Deployment **68-40**
Sample Time Propagation in Generated S-Functions **68-41**
Solver Type for Top Models with Generated S-Functions **68-41**
Tunable Parameters in Generated S-Functions **68-41**
Macro Parameters **68-43**
Checksums and the S-Function Target **68-43**
Generated S-Function Compatibility **68-44**
S-Function Target Limitations **68-44**

69 Desktops in Embedded Coder

Package Generated Code as Shared Libraries **69-2**
About Generated Shared Libraries **69-2**
Generate Shared Library Version of Model Code **69-2**
Create Application Code to Use Shared Library **69-3**
Shared Library Limitations **69-6**

70 Real-Time Systems in Simulink Coder

Deploy Algorithm Model for Real-Time Rapid Prototyping **70-2**
About Real-Time Rapid Prototyping **70-2**
Goals of Real-Time Rapid Prototyping **70-2**
Refine Code With Real-Time Rapid Prototyping **70-2**

Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation **70-4**
About Hardware-In-the-Loop Simulation **70-4**
Set Up and Run HIL Simulations **70-4**

Untrusted Custom Code, Custom Targets, and Callbacks **70-6**

71 Real-Time and Embedded Systems in Embedded Coder

Deploy Applications to Target Hardware **71-2**
Generate Code for an Application **71-2**
Generated Application Code **71-2**
Main Program **71-3**
rt_OneStep and Scheduling Considerations **71-4**
Static Main Program Module **71-8**
Rate Grouping Compliance and Compatibility Issues **71-12**
Generate Code That Dereferences Data from a Literal Memory Address **71-15**

Deploy Component Algorithm as Component Model Library by Using CMake **71-24**

Generate Main Program for Deployment to Bare-Metal Target Hardware **71-32**

Deploy Generated Component Software to Application Target Platforms **71-34**
Interface to an Example Real-Time Multitasking System **71-34**

72 Code Replacement Customization for Simulink Models in Embedded Coder

What Is Code Replacement Customization? **72-3**
Code Replacement Match and Replacement Process **72-3**
Code Replacement Customization Limitations **72-3**

Code You Can Replace From Simulink Models **72-6**
Math Functions – Simulink Support **72-6**
Math Functions – Stateflow Support **72-11**
Memory Functions **72-14**
Nonfinite Functions **72-15**
Mutex and Semaphore Functions **72-15**
Operators **72-16**

Develop a Code Replacement Library **72-20**

Quick Start Code Replacement Library Development - Simulink **72-21**

Semaphore and Mutex Function Replacement **72-30**

Identify Code Replacement Library Requirements **72-35**
What Is a Code Replacement Library? **72-35**
What Functions Can Be Optimized with a Code Replacement Library? **72-35**
How to Develop a Code Replacement Library **72-35**
Identify Basic Requirements **72-35**

Define Code Replacement Library Optimizations **72-36**
Interactively Develop a Code Replacement Library **72-36**
Programmatically Develop a Code Replacement Library **72-39**

Register Code Replacement Library **72-51**
Interactively Develop and Load a Registration file **72-51**
Programmatically Develop and Load a Registration File **72-52**

Verify Code Replacement Library **72-55**
Verify Library Structure **72-55**
Verify Library Replacement **72-55**
Verify Library Performance **72-56**

Deploy Code Replacement Library **72-57**

Troubleshoot Code Replacement Library Registration **72-58**

Troubleshoot Code Replacement Misses **72-59**
Miss Reason Messages **72-59**
Analyze and Correct Code Replacement Misses **72-60**

Math Function Code Replacement **72-64**
Interactively Develop a Code Replacement Library **72-64**
Programmatically Develop a Code Replacement Library **72-66**
Generate Code by Using Math Code Replacement Library **72-69**

Buffer Index Calculation Code Replacement **72-71**
Interactively Develop a Code Replacement Library **72-71**
Programmatically Develop a Code Replacement Library **72-72**
Generate Code by Using Circular Index Code Replacement Library **72-75**

Memory Function Code Replacement **72-78**
Interactively Develop a Code Replacement Library **72-78**
Programmatically Develop a Code Replacement Library **72-80**

Nonfinite Function Code Replacement **72-83**
Interactively Develop a Code Replacement Library **72-83**
Programmatically Develop a Code Replacement Library **72-85**

Algorithm-Based Code Replacement **72-88**
Interactively Develop a Code Replacement Library **72-88**
Programmatically Develop a Code Replacement Library **72-90**

Lookup Table Function Code Replacement **72-94**
Lookup Table Function Signatures **72-94**
Interactively Develop a Code Replacement Library **72-97**
Programatically develop a code replacement library **72-99**
Generate Code by Using Lookup Table Code Replacement Library **72-102**

Data Alignment for Code Replacement **72-105**
Code Replacement Data Alignment **72-105**
Specify Data Alignment Requirements for Function Arguments . **72-105**
Provide Data Alignment Specifications for Compilers **72-106**
Basic Example of Code Replacement Data Alignment **72-109**

Replace MATLAB Functions with Custom Code by Using the coder.replace Function **72-112**
Interactively Develop a Code Replacement Library **72-112**
Programmatically Develop a Code Replacement Library **72-114**

Replace coder.ceval Calls to External Functions **72-118**
Example Files **72-118**
Interactively Develop a Code Replacement Library **72-119**
Programmatically Develop a Code Replacement Library **72-121**

Custom Function Code Replacement **72-125**
Interactively Develop a Code Replacement Library **72-125**
Programmatically Develop a Code Replacement Library **72-126**
Generate Code by Using Custom Function Code Replacement . . **72-129**

Reserved Identifiers and Code Replacement **72-131**

Customize Match and Replacement Process **72-132**
Customize Code Match and Replacement for Functions **72-133**
Customize Code Match and Replacement for Nonscalar Operations **72-135**
Customize Code Match and Replacement for Scalar Operations . **72-139**

Scalar Operator Code Replacement **72-145**
Interactively Develop a Code Replacement Library **72-145**
Programmatically Develop a Code Replacement Library **72-147**

Addition and Subtraction Algorithm Types for Code Replacement **72-150**
Algorithm Options **72-150**
Algorithm Classification **72-150**
Limitations **72-151**

Replace Addition and Subtraction Operator Code **72-153**
Interactively Develop a Code Replacement Library **72-153**
Programmatically Develop a Code Replacement Library **72-155**
Generate Code using Addition and Subtraction Operator Code Replacement Library **72-157**

Matrix Multiplication Operation to MathWorks BLAS Code Replacement **72-160**
Interactively Develop a Code Replacement Library **72-160**
Programmatically Develop a Code Replacement Library **72-162**
Convert a Matrix Multiplication Operation to MathWorks BLAS Through Code Replacement **72-167**

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement **72-170**
Interactively Develop a Code Replacement Library **72-170**
Programmatically Develop a Code Replacement Library **72-172**

Small Matrix Operation to Processor Code Replacement **72-177**
Interactively Develop a Code Replacement Library **72-177**
Programmatically Develop a Code Replacement Library **72-179**
Generate Code to Optimize Performance of a Small Matrix Operation for Processor-Specific Intrinsic Function **72-182**

Remap Operator Output to Function Input **72-185**

Fixed-Point Operator Code Replacement **72-188**
Common Ways to Match Fixed-Point Operator Entries **72-188**
Fixed-Point Numbers and Arithmetic **72-189**
Addition **72-190**
Subtraction **72-190**
Multiplication **72-191**

Division 72-191
Data Type Conversion (Cast) 72-192
Shift 72-192

Binary-Point-Only Scaling Code Replacement 72-194
Programmatically Develop a Code Replacement Library 72-195

Slope Bias Scaling Code Replacement 72-199
Programmatically Develop a Code Replacement Library 72-200

Net Slope Scaling Code Replacement 72-205
Interactively Develop a Code Replacement Library 72-205
Programmatically Develop a Code Replacement Library 72-207
Limitations 72-211

Equal Slope and Zero Net Bias Code Replacement 72-212
Interactively Develop a Code Replacement Library 72-212
Programmatically Develop a Code Replacement Library 72-214

Data Type Conversions (Casts) and Operator Code Replacement 72-218
Interactively Develop a Code Replacement Library 72-218
Programmatically Develop a Code Replacement Library 72-220

Shift Left Operations and Code Replacement 72-224
Interactively Develop a Code Replacement Library 72-224
Programatically Develop a Code Replacement Library 72-226

Optimize Generated Code by Using Multiple Code Replacement Libraries 72-229

Compare Rounding and Overflow Modes in C/C++ Code and Simulink 72-233

73 Code Replacement Customization for MATLAB Code

What Is Code Replacement Customization? 73-3
Code Replacement Match and Replacement Process 73-3
Code Replacement Customization Limitations 73-3

Code You Can Replace from MATLAB Code 73-5
Math Functions 73-5
Memory Functions 73-8
Operators 73-9

Develop a Code Replacement Library 73-13

Quick Start Library Development 73-14

Identify Code Replacement Library Requirements 73-21
What Is a Code Replacement Library? 73-21

What Functions Can Be Optimized with a Code Replacement Library? 73-21
How to Develop a Code Replacement Library 73-21
Identify Basic Requirements 73-21

Define Code Replacement Library Optimizations 73-23
Interactively Develop a Code Replacement Library 73-23
Programmatically Develop a Code Replacement Library 73-26

Register Code Replacement Library 73-38
Interactively Develop and Load a Registration file 73-38
Programmatically Develop and Load a Registration File 73-39

Troubleshoot Code Replacement Library Registration 73-42

Verify Code Replacement Library 73-43
Verify Library Structure 73-43
Verify Library Replacement 73-43
Verify Library Performance 73-44
Review Code Replacements 73-44

Troubleshoot Code Replacement Misses 73-47
Miss Reason Messages 73-47
Analyze and Correct Code Replacement Misses 73-48

Deploy Code Replacement Library 73-50

Math Function Code Replacement 73-51
Interactively Develop a Code Replacement Library 73-51
Programmatically Develop a Code Replacement Library 73-53

Memory Function Code Replacement 73-57
Interactively Develop a Code Replacement Library 73-57
Programmatically Develop a Code Replacement Library 73-59

Specify In-Place Code Replacement 73-62
Interactively Develop a Code Replacement Library 73-62
Programmatically Develop a Code Replacement Library 73-64

Data Alignment for Code Replacement 73-67
Code Replacement Data Alignment 73-67
Specify Data Alignment Requirements for Function Arguments . . 73-67
Provide Data Alignment Specifications for Compilers 73-69
Specify Data Alignment in MATLAB Code for Imported Data 73-72

Array Layout and Code Replacement 73-73

Allow Shape Agnostic Match 73-75
User Interface Method- Code Replacement Tool 73-75
Programmatic Method 73-76
Limitations 73-78

Replace MATLAB Functions with Custom Code by Using the coder.replace Function 73-79
Interactively Develop a Code Replacement Library 73-79

Programmatically Develop a Code Replacement Library 73-81

Replace coder.ceval Calls to External Functions 73-85
Example Files 73-85
Interactively Develop a Code Replacement Library 73-86
Programmatically Develop a Code Replacement Library 73-88

Reserved Identifiers and Code Replacement 73-92

Customize Match and Replacement Process 73-93
Customize Match and Replacement Process for Operators 73-94

Scalar Operator Code Replacement 73-99
Interactively Develop a Code Replacement Library 73-99
Programmatically Develop a Code Replacement Library 73-101

Addition and Subtraction Operator Code Replacement 73-104
Algorithm Options 73-104
Algorithm Classification 73-104
Interactively Develop a Code Replacement Library 73-105
Programmatically Develop a Code Replacement Library 73-108
Limitations 73-110

Small Matrix Operation to Processor Code Replacement 73-111
Interactively Develop a Code Replacement Library 73-111
Programmatically Develop a Code Replacement Library 73-113

Matrix Multiplication Operation to MathWorks BLAS Code Replacement 73-117
Interactively Develop a Code Replacement Library 73-117
Programmatically Develop a Code Replacement Library 73-119

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement 73-125
Interactively Develop a Code Replacement Library 73-125
Programmatically Develop a Code Replacement Library 73-127

Remap Operator Output to Function Input 73-132

Fixed-Point Operator Code Replacement 73-134
Common Ways to Match Fixed-Point Operator Entries 73-134
Fixed-Point Numbers and Arithmetic 73-135
Addition 73-136
Subtraction 73-136
Multiplication 73-137
Division 73-137
Data Type Conversion (Cast) 73-138
Shift 73-138

Binary-Point-Only Scaling Code Replacement 73-140
Programmatically Develop a Code Replacement Library 73-141

Slope Bias Scaling Code Replacement 73-145
Programmatically Develop a Code Replacement Library 73-146

Net Slope Scaling Code Replacement **73-151**
Interactively Develop a Code Replacement Library **73-151**
Programmatically Develop a Code Replacement Library **73-153**

Equal Slope and Zero Net Bias Code Replacement **73-158**
Interactively Develop a Code Replacement Library **73-158**
Programmatically Develop a Code Replacement Library **73-160**

Data Type Conversions (Casts) and Operator Code Replacement **73-164**
Interactively Develop a Code Replacement Library **73-164**
Programmatically Develop a Code Replacement Library **73-166**

Shift Left Operations and Code Replacement **73-170**
Interactively Develop a Code Replacement Library **73-170**
Programatically Develop a Code Replacement Library **73-172**

Optimize Generated Code by Developing and Using Code Replacement Libraries - MATLAB **73-175**

74 Embedded Coder Support Package for Linux Applications

Support Package Installation **74-2**

Setup Linux Target Computer **74-3**
Target Computer requirements **74-3**
Steps to setup target **74-3**

Build Simulink Model and Deploy Application **74-6**
Deploy DDS Blockset Models **74-7**

Start and Stop Deployed Applications **74-9**

Tune Parameters and Measure Signal Data for Deployed Applications **74-11**

Analyze Logged Messages **74-13**

External Mode Simulation of Deployed Applications **74-15**

75 Data Communication Methods

Sample Target Environment Service Implementations of Data Communication Methods **75-2**
Direct-Access Data Communication **75-2**
Outside-Execution Data Communication **75-3**
During-Execution Data Communication **75-5**

76 Optimizations for Generated Code in Simulink Coder

Increase Code Generation Speed **76-3**
Build a Model in Increments **76-3**
Build Large Model Reference Hierarchies in Parallel **76-3**
Minimize Memory Requirements During Code Generation **76-4**
Generate Only Code **76-4**
Suppress Creation of a Code Generation Report **76-4**

Control Compiler Optimizations **76-6**

Design Techniques to Optimize Models for Efficient Code Generation **76-7**
Use the Model Advisor to Optimize a Model **76-7**
Design Techniques to Optimize a Model for Code Generation **76-7**
Design Techniques to Optimize Stateflow Objects for Code Generation **76-8**

Control Memory Allocation for Time Counters **76-10**

Execution Profiling for Generated Code **76-11**

Optimize Generated Code by Combining Multiple for Constructs **76-13**

Subnormal Number Execution Speed **76-16**
Simulation Time with and Without Subnormal Numbers **76-17**
Flush Subnormal Numbers to Zero **76-18**

Remove Code for Out-of-Range Floating Point to Integer Conversions **76-20**

Remove Code That Maps NaN to Integer Zero **76-23**

Disable Nonfinite Checks or Inlining for Math Functions **76-26**

Fold Expressions **76-31**

Minimize Computations and Storage for Intermediate Results at Block Outputs **76-35**
Expression Folding **76-35**
Example Model **76-35**
Generate Code **76-35**
Enable Optimization **76-36**
Generate Code with Optimization **76-36**

Inline Invariant Signals **76-38**
Optimize Generated Code Using Inline Invariant Signals **76-38**

Inline Numeric Values of Block Parameters **76-41**

Configure Loop Unrolling Threshold **76-46**

Use memcpy Function to Optimize Generated Code for Vector Assignments **76-48**
Example Model **76-48**
Generate Code **76-49**
Generate Code with Optimization **76-50**

Generate Target Optimizations Within Algorithm Code **76-51**

Remove Code for Blocks That Have No Effect on Computational Results **76-52**

Eliminate Dead Code Paths in Generated Code **76-56**

Floating-Point Multiplication to Handle a Net Slope Correction . **76-59**

Use Conditional Input Branch Execution **76-62**

Optimize Generated Code for Complex Signals **76-66**

Speed Up Linear Algebra in Code Generated from a MATLAB Function Block **76-68**
Specify LAPACK Library **76-68**
Write LAPACK Callback Class **76-68**
Generate LAPACK Calls by Specifying a LAPACK Callback Class . . **76-69**
Locate LAPACK Library in Execution Environment **76-70**

Speed Up Matrix Operations in Code Generated from a MATLAB Function Block **76-71**
Specify BLAS Library **76-71**
Write BLAS Callback Class **76-71**
Generate BLAS Calls by Specifying a BLAS Callback Class **76-73**
Locate BLAS Library in Execution Environment **76-73**
Usage Notes and Limitations for OpenBLAS Library **76-74**

Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block **76-75**
Use Built-in FFTW Library Callback for FFTW Library Usage **76-75**
Use Custom FFT Library Callback for FFTW Library Usage **76-75**

Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block **76-78**
Prerequisites **76-78**
Create a Model with a MATLAB Function Block That Calls an FFT Function **76-78**
Write Supporting C Code **76-79**
Create an FFT Library Callback Class **76-79**
Configure Code Generation Parameters and Build the Model **76-80**

Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block **76-82**
Provide Upper Bounds for Variable-Size Arrays **76-82**

Disable Dynamic Memory Allocation for MATLAB Function Blocks 76-82
Modify the Dynamic Memory Allocation Threshold 76-82

Optimize Memory Usage for Time Counters **76-84**

Optimize Generated Code Using Boolean Data for Logical Signals **76-89**

Reduce Memory Usage for Boolean and State Configuration Variables **76-92**

Customize Stack Space Allocation **76-93**

Optimize Generated Code Using memset Function **76-95**

Vector Operation Optimization **76-99**

Enable and Reuse Local Block Outputs in Generated Code **76-102**
Example Model **76-102**
Generate Code Without Optimization **76-102**
Enable Local Block Outputs and Generate Code **76-103**
Reuse Local Block Outputs and Generate Code **76-104**

Generate SIMD Code from Simulink Blocks **76-105**
Blocks That Support SIMD Code Generation **76-105**
Generate SIMD Code Compared to Plain C Code **76-107**
Limitations **76-109**

Optimize Code for Reduction Operations by Using SIMD **76-110**

77 Configuration in Embedded Coder

Set Hardware Implementation Parameters **77-2**

78 Data Copy Reduction in Embedded Coder

Optimize Global Variable Usage **78-2**
Use Global to Hold Temporary Results **78-2**
Minimize Global Data Access **78-6**

Reuse Global Block Outputs in the Generated Code **78-12**

Virtualized Output Ports Optimization **78-15**

Specify Buffer Reuse for Signals in a Path **78-17**

Specify Buffer Reuse for MATLAB Function Blocks in a Path **78-25**
Example Model **78-25**
Generate Code with Optimization **78-25**

Remove Data Copies by Reordering Block Operations in the Generated Code **78-27**
Example Model **78-27**
Generate Code without Optimization **78-27**
Generate Code with Optimization **78-29**

Data Copy Reduction for Data Store Read and Data Store Write Blocks **78-32**

Reduce Data Copies for Bus Assignment Blocks in Generated Code **78-36**
Example Model **78-36**
Generate Code Without Optimization **78-36**
Generate Code with Optimization **78-37**

Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse **78-39**

79 Execution Speed in Embedded Coder

Reduce Memory Usage for Signals **79-2**

Remove Zero-Initialization Code **79-4**
Remove Zero Initialization Code for Internal Data **79-4**
Remove Initialization Code from Root-Level Inports and Outports Set to Zero **79-6**
Additional Information **79-8**

Simplify Multiply Operations in Array Indexing **79-9**
Example Model **79-9**
Generate Code **79-9**
Generate Code with Optimization **79-10**

Replace boolean with Specific Integer Data Type **79-12**
Example Model **79-12**
Generate Code That Contains the Default boolean Data Type **79-12**
Generate Code That Contains the Target boolean Data Type **79-13**

Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow **79-15**

Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data **79-18**

Optimize Generated Code by Consolidating Redundant If-Else Statements **79-22**

Optimize Generated Code for Fixed-Point Data Operations **79-26**

Unroll Parallel for-Loop That Has Small Number of Iterations . . . **79-28**

Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink **79-31**

Improve Execution Efficiency by Reordering Block Operations in the Generated Code **79-52**

Optimize SIMD Code by Performing Fused Multiply Add Operations **79-63**

Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface **79-67**
Generate OpenMP Code from MATLAB Function or MATLAB System Blocks **79-67**

Accelerate Pedestrian Detection with SIMD Code **79-69**

80 Memory Usage in Embedded Coder

Optimize Generated Code Using Minimum and Maximum Values **80-2**
Configure Your Model **80-2**
Optimize Generated Code Using Specified Minimum and Maximum Values **80-3**
Limitations **80-6**

Reduce Global Variables in Nonreusable Subsystem Functions . . . **80-8**
Generate void-void Function **80-8**
Generate Function with Arguments **80-9**

Optimize Generated Code by Packing Boolean Data into Bitfields **80-11**

Optimize Generated Code by Passing Reusable Subsystem Outputs as Individual Arguments **80-14**

Convert Data Copies to Pointer Assignments **80-18**

Reuse Buffers of Different Sizes and Dimensions **80-23**

Remove Reset and Disable Functions from the Generated Code . **80-29**

Control Operator Type in Generated Code **80-32**

Reduce Memory Usage for Models Containing Referenced Models **80-35**

Optimize Generated Code Using Code Efficiency Tools and Techniques **80-39**
Techniques to Refine Models for Efficient Code Generation **80-39**
Techniques to Identify Areas for Optimizing Memory Usage **80-42**
Techniques to Optimize Code for Memory Efficiency **80-44**
Techniques to Identify Areas for Optimizing Code Execution-Time **80-45**
Explore Parallelization Opportunities **80-46**

81 Code Execution and Stack Usage Profiling in Embedded Coder

Create Execution-Time Profile for Generated Code **81-2**
Configure Code Execution Profiling **81-2**
Control Profiling Granularity **81-3**

View and Compare Code Execution Times **81-6**
Run Simulation That Generates Execution-Time Metrics **81-6**
Analyze Task and Function Execution Times **81-9**
View Code Execution Profiles for Multiple Model Blocks **81-16**
Visualize Task Scheduling **81-17**

Code Execution Profiling Report **81-19**
Summary **81-19**
Feasibility Checks **81-19**
Profiled Sections of Code **81-20**
Execution Times in Percentages **81-24**
CPU Utilization **81-25**
Definitions **81-25**
Comparison with Baseline Performance **81-26**
Profiling Reports for Multiple Model Blocks **81-26**

Analyze Code Execution Data **81-29**

Code Execution Profiling From the Command Line **81-31**

Remove Instrumentation Overheads from Execution Time Measurements **81-32**
Approach for Estimating and Removing Instrumentation Overhead **81-32**
Remove Instrumentation Overheads by Using target Package **81-33**
Remove Instrumentation Overheads by Using Functions from rtw.connectivity.Config **81-34**
Retrieve Benchmark Results **81-35**

Reduce Bandwidth Usage During Code Execution Profiling **81-37**
Capture Main Profiling Metrics on Target Hardware **81-37**
Specify Memory Allocation for Code Execution Profiling **81-38**
Display Absolute Time **81-39**

Visualize Task Scheduling in XCP External Mode Simulation **81-40**
Model Configuration **81-40**
Run-Time Visualization **81-42**
Offline Visualization **81-45**

Tips and Limitations **81-50**
Triggered Model Block **81-50**
Outliers in Execution-Time Profiles **81-50**
Hardware-Specific Timer **81-51**
Effect of Instrumentation Code on Instruction Cycle **81-51**
Task Context Switching Due to Preemption **81-51**
Subsystem Code Reuse **81-51**
Cannot Load Execution-Time Measurements from Previous Release **81-52**

Stack Usage Profiling for Code Generated from Simulink Models **81-53**
Benefits of Dynamic Stack Usage Profiling **81-53**
Configure Stack Usage Profiling **81-53**
View Stack Usage Metrics for Tasks and Functions **81-54**
Compare Stack Usage Against Baseline **81-58**
Summary Panel (Stack Usage) **81-59**
Code Stack Profiling Report **81-60**
Implement Driver to Obtain Stack Usage Data During PIL Simulation **81-62**

82 Code Execution Profiling for MATLAB Coder

Execution Time Profiling for SIL and PIL **82-2**

Generate Execution Time Profile **82-3**

View Execution Times **82-4**
Code Execution Profiling Report **82-4**
Visualize Task Scheduling **82-8**

Analyze Execution Time Data **82-9**
Extract Execution Time Data for Kalman Estimator Code **82-9**

Stack Usage Profiling for Code Generated From MATLAB Code . . **82-13**
Benefits of Dynamic Stack Usage Profiling **82-13**
Generate Stack Usage Profile **82-13**
View Stack Usage Profile From SIL Execution **82-14**
Implement Driver to Obtain Stack Usage Data During PIL Execution **82-17**

Removal of Unused Class Properties in Generated C/C++ Code . . **82-19**
Preserve Unused Class Properties or Structure Fields by using MATLAB Coder App **82-19**
Preserve Unused Class Properties or Structure Fields at the Command Line **82-19**

Examples 82-20
Usage Notes 82-21

83 High-throughput Applications in Simulink Coder

Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection 83-2

Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models 83-10

Generate Code for Deep Learning Networks Using MATLAB Function Block 83-22
Classify Images by Using GoogLeNet 83-22
Create GoogLeNet Model 83-23
Simulate Model 83-25
Generate Code 83-26

Generate Code for Blocks from Deep Neural Networks Library . . 83-27
Classify Images by Using GoogLeNet 83-27
Create GoogLeNet Model 83-28
Simulate Model 83-30
Generate Code 83-31

Workflow for Deep Learning C/C++ Code Generation for Simulink Models 83-32

Verification

84 Simulation and Code Comparison in Simulink Coder

Simulation and Code Comparison 84-2
Configure Signal Data for Logging 84-2
Log Simulation Data 84-3
Run Executable and Load Data 84-4
Visualize and Compare Results 84-5
Compare States for Simulation and Code Generation 84-7

85 Code Tracing in Embedded Coder

Verify Generated Code by Using Code Tracing **85-2**
Traceable Elements **85-2**
Traceability in Generated Code **85-3**
Traceability Tags **85-5**
Operator Traceability **85-6**
Traceability Limitations **85-7**

Trace Simulink Model Elements in Generated Code **85-8**
Code-to-Model Traceability **85-8**
Model-to-Code Traceability **85-10**

Trace Stateflow Elements in Generated Code **85-12**
Inline Traceability for Stateflow Elements **85-12**
Bidirectional Traceability for States and Transitions **85-13**
Bidirectional Traceability for State Transition Tables **85-15**
Bidirectional Traceability for Truth Table Blocks **85-17**
Bidirectional Traceability for Graphical Functions **85-19**
Code-to-Model Traceability for Events **85-19**
Model-to-Code Traceability for Junctions **85-20**
Format of Traceability Comments for Stateflow Elements **85-21**

Link Generated Code to Requirements **85-24**

Reload Existing Traceability Information **85-27**

Customize Traceability Reports **85-29**
Generate a Traceability Matrix **85-30**

Trace Generated Code to Blocks **85-32**

Use Traceability in MATLAB Function Blocks **85-34**
Extent of Traceability in MATLAB Function Blocks **85-34**
Traceability Requirements **85-34**
Using Traceability in a MATLAB Function Block **85-34**

86 Component Verification in Embedded Coder

Component Verification in Target Environment **86-2**
Goals of Component Verification **86-2**
Run Component Tests **86-4**

87 Component Verification With a Real-Time Target Environment in Embedded Coder

Real-Time Software Component Verification **87-2**

88 Numerical Equivalence Checking in Embedded Coder

SIL and PIL Simulations **88-2**
What Are SIL and PIL Simulations? **88-2**
Why Use SIL and PIL **88-2**
How SIL and PIL Simulations Work **88-3**
Comparison of SIL and PIL Simulations **88-4**
Code Interfaces for SIL and PIL **88-4**
Scheduling Considerations **88-6**
Imported Data and Additional Generated Code **88-6**

Choose a SIL or PIL Approach **88-11**
Test Top-Model Code **88-12**
Test Referenced Model Code **88-12**
Test Subsystem Code **88-12**
Summary **88-13**

Configure and Run SIL Simulation **88-15**
Simulation with Top Model **88-15**
Simulation with Model Blocks **88-16**
Simulation with Subsystem Blocks **88-17**
SIL or PIL Block Simulation **88-18**
Configure Hardware Implementation Settings **88-18**
Log Signals of a Component **88-20**
Prevent Code Changes in Multiple Simulations **88-21**
Speed Up Testing **88-22**
Simulation with Function Calls **88-23**

Configure and Run PIL Simulation **88-24**
Simulation with Top Model **88-24**
Simulation with Model Blocks **88-25**
Simulation with Subsystem Blocks **88-26**
SIL or PIL Block Simulation **88-27**
Log Signals of a Component **88-27**
Prevent Code Changes in Multiple Simulations **88-28**
Speed Up Testing **88-29**
Simulation with Function Calls **88-30**

Unit Test Subsystem Code with SIL/PIL Manager **88-31**

Simulation Mode Override Behavior in Model Reference Hierarchy **88-33**

Debug Generated Code During SIL or PIL Simulation **88-35**
SIL Debugging **88-35**
PIL Debugging **88-37**

Create PIL Target Connectivity Configuration for Simulink **88-38**
Target Connectivity Configurations for PIL **88-38**
Create a Target Connectivity API Implementation **88-38**
Register a Connectivity API Implementation **88-40**
Verify Target Connectivity Configuration **88-40**
Target Connectivity API Examples **88-40**

Host-Target Communication for Simulink PIL Simulation **88-42**
Communications rtiostream API **88-42**
Synchronize Host and Target **88-43**
Test an rtiostream Driver **88-43**
Troubleshooting Host-Target Communication **88-43**
Word Addressable Target Hardware **88-44**

Specify Hardware Timer for Simulink **88-45**

Set Up PIL Connectivity by Using target Package **88-48**
Use rtiostream API for PIL Target Connectivity **88-48**
Support PIL Debugging **88-51**
DebugExecutionTool Template **88-52**
Use Debugger for PIL Target Connectivity **88-57**
DebugIOTool Template **88-58**

Custom Toolchain Directives Required for Code Coverage and Execution Profiling **88-68**

PIL Simulation Sequence **88-69**

Verification of Code Generation Assumptions **88-71**

View SIL and PIL Files in Code Generation Report **88-74**

SIL/PIL Manager Verification Workflow **88-76**
Open the App **88-76**
Select the Mode **88-76**
Prepare the Model **88-76**
Run Automated Verification, Model Simulation, or SIL/PIL Simulation **88-79**
Analyze Results and Export Test Cases **88-79**

Security for PIL Simulations **88-82**

SIL and PIL Limitations **88-83**
About SIL and PIL Limitations **88-83**
General SIL and PIL Limitations **88-83**
Top-Model SIL/PIL Limitations **88-90**
Model Block SIL/PIL Limitations **88-91**
Subsystem Block SIL/PIL Limitations **88-92**

Test Generated Code with SIL and PIL Simulations **88-94**

Create a Target Communication Channel for Processor-in-the-Loop (PIL) Simulation **88-108**

Configure Processor-In-The-Loop (PIL) for a Custom Target . . . **88-113**

Field-Oriented Control of Permanent Magnet Synchronous Machine **88-118**

Check Configuration **88-135**

Verify Numerical Equivalence with CGV **88-136**

Verify Numerical Equivalence Between Two Modes of Execution of a Model **88-137**
- Configure the Model **88-137**
- Execute the Model **88-137**
- Compare All Output Signals **88-138**
- Compare Individual Output Signals **88-139**
- Plot Output Signals **88-140**

89 Numerical Consistency between Model and Generated Code

Numerical Consistency of Model and Generated Code Simulation Results **89-2**
- Numerical Consistency **89-2**
- Numerical Consistency in Complex Systems **89-2**
- Reasons for Block-Level Numerical Differences **89-4**

90 Software-in-the-Loop Execution for MATLAB Coder

Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution **90-2**

Software-in-the-Loop Execution with the MATLAB Coder App **90-4**

Software-in-the-Loop Execution From Command Line **90-6**
- Tutorial Files: Kalman Filter **90-6**
- SIL Execution of Code Generated for a Kalman Estimator **90-6**

Debug Generated Code During SIL or PIL Execution **90-9**
- SIL Debugging **90-9**
- PIL Debugging **90-11**

Create PIL Target Connectivity Configuration for MATLAB **90-12**
- Target Connectivity Configurations for PIL **90-12**

Create a Target Connectivity API Implementation 90-12
Register Target Connectivity Configuration 90-14
Verify Target Connectivity Configuration 90-14

Host-Target Communication for MATLAB PIL Execution 90-16
Communications rtiostream API 90-16
Synchronize Host and Target 90-17
Test an rtiostream Driver 90-17
Troubleshooting Host-Target Communication 90-17

Specify Hardware Timer for MATLAB 90-19

Custom Toolchain Directives Required for Code Execution Profiling 90-22

Processor-in-the-Loop Execution with the MATLAB Coder App . . 90-23

Processor-in-the-Loop Execution From Command Line 90-25
Tutorial Files: Kalman Filter 90-25
PIL Execution of Code Generated for a Kalman Estimator 90-25

Verification of Code Generation Assumptions 90-29

SIL/PIL Execution Support and Limitations 90-30

Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization 90-32
Disable Constant Input Checking or Global Data Synchronization at the Command Line 90-32
Disable Constant Input Checking or Global Data Synchronization in the MATLAB Coder App 90-32

SIL and PIL Verification for Deployment on Raspberry Pi 90-33

Software-in-the-loop Execution For MATLAB Function with Multiple Signatures 90-39
Generate Multisignature SIL MEX File for an Entry-Point Function 90-39
Generate Multisignature SIL MEX File for Multiple Entry-Point Functions 90-40

91 Code Coverage in Embedded Coder

Simulink Code Coverage Metrics 91-2
Statement Coverage 91-2
Condition Coverage 91-2
Decision Coverage 91-3
Modified Condition/Decision Coverage (MCDC) 91-4
Cyclomatic Complexity 91-4
Relational Boundary Coverage 91-5

Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode **91-6**
Enable SIL or PIL Code Coverage for a Model **91-6**
Review the Coverage Results for Models in SIL or PIL Mode **91-6**
Limitations **91-8**

Configure Code Coverage with Third-Party Tools **91-9**

View Code Coverage Information at the End of SIL or PIL Simulations **91-11**
View LDRA Tool Suite Results **91-11**
View BullseyeCoverage Results **91-11**

Configure Code Coverage Programmatically **91-13**
Set Up Third Party Code Coverage From Command Line **91-13**
Code Coverage for Model That Uses Configuration Reference **91-14**

Code Coverage Summary and Annotations **91-15**
LDRA Tool Suite Coverage **91-15**
BullseyeCoverage Information **91-17**

Code Coverage Tool Support **91-20**

Tips and Limitations **91-21**
Model Build and SIL/PIL Blocks Not Supported **91-21**
BullseyeCoverage License Wait **91-21**
Current Working Folder Cannot be UNC Path **91-21**
Characters in matlabroot and File Path **91-21**
Header Files with Identical Names **91-21**
Code Coverage for Source Files in Shared Utility Folders **91-21**
BullseyeCoverage Behavior with Inline Macros **91-22**
SIL and PIL Simulations with Open LDRA GUI **91-22**
Minor SIL and PIL Differences for LDRA Tool Suite **91-22**
PIL Zero Coverage LDRA Annotations **91-23**
Custom Header File Changes and Regeneration of LDRA Instrumentation **91-23**
PIL Support for BullseyeCoverage **91-23**
Modify Legacy Code **91-24**

Collect Code Coverage Metrics with a Third-Party Tool **91-25**
Initial Setup **91-25**
Run a Simulation in Normal Mode **91-25**
Run a Second Simulation in Normal Mode **91-26**
Configure the Model to Measure Code Coverage **91-26**
Run Simulations in SIL Mode **91-26**
Concluding Remarks **91-27**

Embedded IDEs and Embedded Targets

92 Run-Time Data Interface Extensions in Simulink Coder

Customize Generated ASAP2 File **92-2**
About ASAP2 File Customization **92-2**
ASAP2 File Structure on the MATLAB Path **92-2**
Customize the Contents of the ASAP2 File **92-2**
ASAP2 Templates **92-3**
Customize Computation Method Names **92-5**
Suppress Computation Methods for FIX_AXIS **92-6**

93 Build Process Integration in Simulink Coder

Control Build Process Compiling and Linking **93-2**

Control Library Location and Naming During Build **93-3**
Library Control Parameters **93-3**
Specify the Location of Precompiled Libraries **93-4**
Location of Model Reference Libraries **93-5**
Control the Suffix Applied to Library File Names **93-5**

Customize Post-Code-Generation Build Processing **93-7**
Workflow for Setting Up Customizations **93-7**
Build Information Object **93-7**
Program a Post Code Generation Command **93-8**
Define a Post Code Generation Command **93-9**
Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment **93-9**
Suppress Makefile Generation **93-11**

Configure Generated Code with TLC **93-12**
Assigning Target Language Compiler Variables **93-12**
Set Target Language Compiler Options **93-14**

Use makecfg to Customize Generated Makefiles for S-Functions **93-15**

Use rtwmakecfg.m API to Customize Generated Makefiles **93-17**
About the rtwmakecfg Function **93-17**
Create the rtwmakecfg Function **93-17**
Modify the Template Makefile for rtwmakecfg **93-19**

Register Custom Toolchain and Build Executable **93-21**

Customize Build Process with STF_make_rtw_hook File **93-25**
STF_make_rtw_hook File **93-25**

Conventions for Using the STF_make_rtw_hook File 93-25
STF_make_rtw_hook.m Function Prototype and Arguments 93-25
Applications for STF_make_rtw_hook.m 93-27
Control Code Regeneration Using STF_make_rtw_hook.m 93-27
Use STF_make_rtw_hook.m for Your Build Procedure 93-28
Hook Method after_tlc 93-28

Customize Build Process with sl_customization.m 93-29
The sl_customization.m File 93-29
Register Build Process Hook Functions Using sl_customization.m 93-30
Variables Available for sl_customization.m Hook Functions 93-31
Example of Build Process Customization with sl_customization.m 93-31

Replace STF_rtw_info_hook Supplied Target Data 93-33

94 Custom Target Development in Simulink Coder

Custom Targets 94-2

Sample Custom Targets 94-4

Target Development Folders, Files, and Builds 94-6
Folder and File Naming Conventions 94-6
Components of a Custom Target 94-6
Key Folders Under Target Root (mytarget) 94-10
Key Files in Target Folder (mytarget/mytarget) 94-12
Additional Files for Externally Developed Targets 94-14
Target Development and the Build Process 94-14

Customize System Target Files 94-19
Control Code Generation With the System Target File 94-19
System Target File Naming and Location Conventions 94-19
System Target File Structure 94-19
Define and Display Custom Target Options 94-25
Tips and Techniques for Customizing Your System Target File . . . 94-30
Create a Custom Target Configuration 94-33

Customize Template Makefiles 94-44
Template Makefiles and Tokens 94-44
Invoke the make Utility 94-49
Structure of the Template Makefile 94-50
Customize and Create Template Makefiles 94-52

Custom Target Optional Features 94-57

Support Toolchain Approach with Custom Target 94-59

Support Model Referencing **94-61**
Requirements for Model Referencing with a Custom System Target File **94-61**
Declaring Model Referencing Compliance **94-61**
Providing Model Referencing Support in the TMF **94-62**
Controlling Configuration Option Value Agreement **94-63**
Preventing Resource Conflicts (Optional) **94-64**

Support Compiler Optimization Level Control **94-65**
About Compiler Optimization Level Control and Custom Targets . . **94-65**
Declaring Compiler Optimization Level Control Compliance **94-65**
Providing Compiler Optimization Level Control Support in the Target Makefile **94-66**

Support C Function Prototype Control **94-67**
About C Function Prototype Control and Custom Targets **94-67**
Declaring C Function Prototype Control Compliance **94-67**
Providing C Function Prototype Control Support in the Custom Static Main Program **94-68**

Support C++ Class Interface Control **94-69**
About C++ Class Interface Control and Custom Targets **94-69**
Declaring C++ Class Interface Control Compliance **94-69**
Providing C++ Class Interface Control Support in the Custom Static Main Program **94-69**

Support Concurrent Execution of Multiple Tasks **94-71**

Device Drivers **94-72**

95 Build Configurations for Embedded Targets in Embedded Coder

Model Setup **95-2**
Block Selection **95-2**
Configure Target Hardware Resources **95-2**

96 Processor-Specific Optimizations for Embedded Targets in Embedded Coder

Replace Code for Embedded Targets **96-2**
Using a Processor-Specific Code Replacement Library to Optimize Code **96-2**
Process of Determining Optimization Effects Using Real-Time Profiling Capability **96-2**

97 Build Configuration for Code Generation from MATLAB Code

Specify Comment Style for C/C++ Code 97-2
Specify Comment Style Using the MATLAB Coder App 97-2
Specify Comment Style Using the Command-Line Interface 97-2

Specify Indent Style for C/C++ Code 97-4
Specify Indent Style Using the MATLAB Coder App 97-4
Specify Indent Style Using the Command-Line Interface 97-5

Generate Custom File and Function Banners for C/C++ Code 97-6

Code Generation Template Files for MATLAB Code 97-9
Default CGT File 97-9
CGT File Structure 97-9
Components of the CGT File Sections 97-10

Customize Generated Identifiers 97-18
Customize Identifiers by Using the MATLAB Coder App 97-18
Customize Generated Identifiers by Using the Command-Line Interface 97-19

Customize C/C++ File Names Generated from MATLAB Code . . . 97-20
Customize File Names by Using the MATLAB Coder App 97-20
Customize Generated File Names by Using the Command-Line Interface 97-20

Control Signed Left Shifts in Generated Code 97-22
Control Signed Left Shifts Using the MATLAB Coder App 97-22
Control Signed Left Shifts Using the Command-Line Interface . . . 97-22

Control Data Type Casts in Generated Code 97-23
Specify Casting Mode Using the MATLAB Coder App 97-24
Specify Casting Mode Using the Command-Line Interface 97-24

Simplify Multiply Operations for Array Indexing in Loops 97-26

Generate SIMD Code for MATLAB Functions 97-27
MATLAB Functions That Support SIMD Code 97-27
Generate SIMD Code Versus Plain C Code 97-29
Generate SIMD Instructions for MEX Code 97-31
Limitations 97-32

Customize Data Type Replacement 97-33
Specify Custom Data Type Names by Using the MATLAB Coder App 97-34

Specify Custom Data Type Names by Using the Command-Line Interface 97-34

Import Custom Data Type Definitions from External Header Files **97-35**
Import Custom Types by Using the MATLAB Coder App 97-35
Import Custom Types by Using the Command-Line Interface 97-36

MATLAB Code Patterns That Require a Nonempty Initialize Function in Generated Code **97-37**
Issue 97-37
Solution 97-37

98 MISRA and AUTOSAR Compliance of Code Generated from MATLAB

Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance **98-2**
Scope of Code Generation Support 98-2
Configure Code Generation Configuration Object Properties at Command Line 98-3
Configure MATLAB Coder App Settings 98-4
Additional Settings for MISRA C++ Compliance 98-5
Check Compliance of Generated Code Using Static Analysis Tool . . 98-5

C Coding Directives and Rules Supported for Code Generation . . . **98-6**

C++ Coding Rules Supported for Code Generation **98-20**

99 Storage Classes for Code Generation from MATLAB Code

Storage Classes for Code Generation from MATLAB Code **99-2**

Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code **99-4**

100 Verification of Code Generated from MATLAB Code

Highlight Potential Data Type Issues in a Report **100-2**
Enable Highlight Option Using the MATLAB Coder App 100-2
Enable Highlight Option Using the Command Line Interface 100-2

Find Potential Data Type Issues in Generated Code **100-4**
Data Type Issues Overview **100-4**
Enable Highlighting of Potential Data Type Issues **100-4**
Find and Address Cumbersome Operations **100-5**
Find and Address Expensive Rounding **100-5**
Find and Address Expensive Comparison Operations **100-6**
Find and Address Multiword Operations **100-7**

PIL Execution with ARM Cortex-A at the Command Line **100-9**

PIL Execution with ARM Cortex-A by Using the MATLAB Coder App **100-11**

Interactively Trace Between MATLAB Code and Generated C/C++ Code **100-12**
Create the MATLAB Source Code **100-12**
Prepare for Code Generation **100-15**
Produce a Code Generation Report with Traceability **100-16**
Access Trace Mode in the Report **100-16**
Trace Code **100-17**
View Multiple Traces **100-19**
View Traces to Different Files **100-19**
Switch the Locations of the Source and Generated Code **100-20**
Enable Code Tooltips and Links **100-20**
Disable Traceability **100-21**

Polyspace Verification of C/C++ Code Generated by MATLAB Coder **100-22**
Run Polyspace Analysis in the MATLAB Coder App **100-22**
Run Polyspace Analysis on Code Generated by codegen **100-22**
Review Analysis Results **100-22**

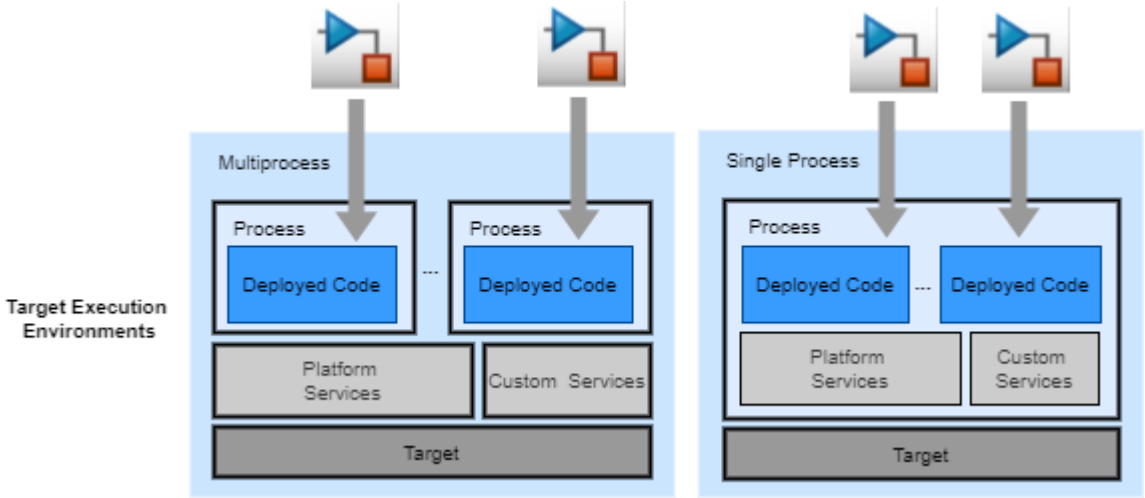
Embedded Coder Fundamentals

Embedded Coder Fundamentals

Software Deployment Objectives and Deployment Types

What Is Software Deployment?

Software deployment is the process of producing units of code from Simulink® models with the intention of integrating the code with other software for use in a target execution environment. The target execution environment can be single- or multiprocess, as this figure shows.

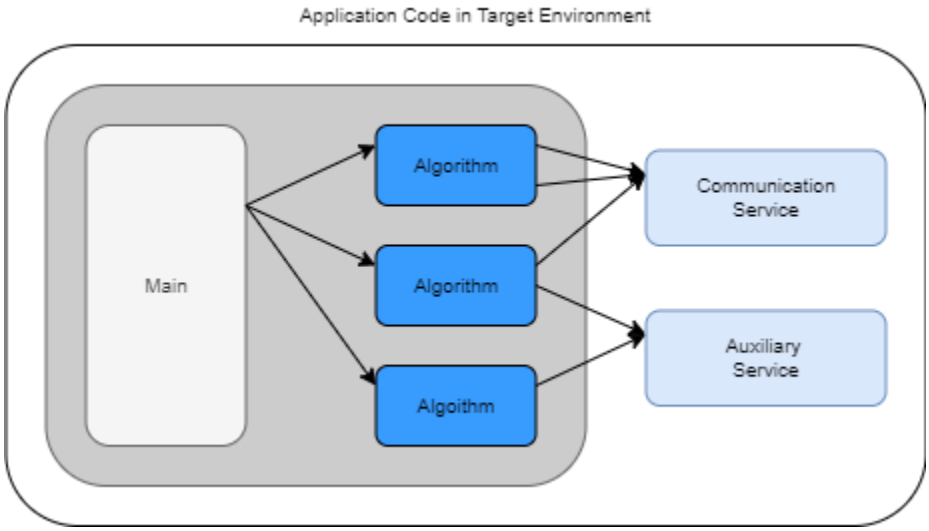


Within a target execution environment, a process is an instance of a program that runs in its own address space. Deployed units of code that run as part of a single process share and must synchronize access to this address space. Processes and units of code within a process exchange data and synchronize their operation by using services.

A target execution environment consists of these parts:

Parts	Description
Target	Hardware and software (for example, a virtual machine) that supports execution of processes.
Platform services	Software that enables algorithms to perform their intended function without knowledge of the target. They include or leverage operating system services, middleware services, and device drivers that handle scheduling, communication, and auxiliary real-time functions.
Custom services	Services that are not provided by the platform, but are required for deployment. Examples include specialized middleware and device drivers. For generated code to gain access to custom services, you must develop and use custom blocks and storage classes.
Deployed code	Units of code generated from Simulink top models.

Deployed code can consist of a main program and algorithm code that the main program calls or code for individual algorithms that get integrated with an existing external main program. Algorithm code calls service implementations that are provided by the target environment for communication and auxiliary support. The arrows in this figure represent function calls.



Target execution environments for embedded system applications vary in composition and complexity. For example, the target might be a development computer, rapid-prototyping board, microprocessor, microcontroller, field programmable gate array (FPGA), system-on-chip (SoC), or virtual machine, and might include a real-time operating system (RTOS). To ensure an optimized and accurate model design and to accurately configure the model for code generation, you or someone else contributing to your project must identify the parts, characteristics, and requirements for your generated code to interface with the target environment.

Deployment Objectives and Deployment Types

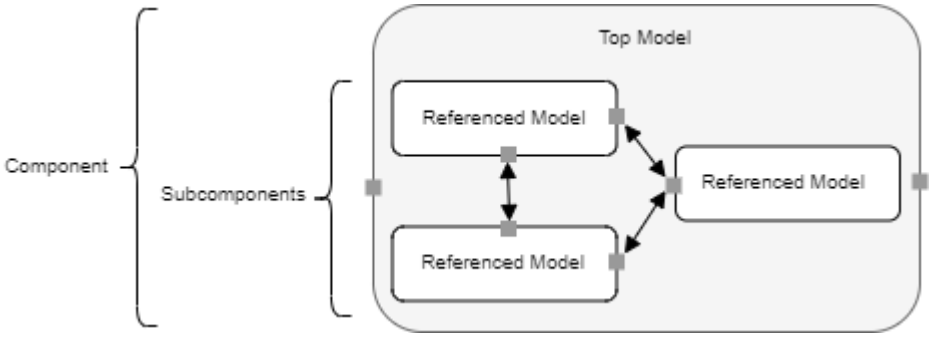
Embedded Coder tooling supports various deployment objectives, enabling you to generate and verify code for Simulink models incrementally. You can use Embedded Coder to deploy an application, component, subcomponent, or utility. Applications and components are units of code, which the code generator produces from top models, that you deploy for use in target execution environments.

Subcomponents and utilities are units of code that you can produce and test independently, but are deployed within the context of an application or component. For verification, you might include an application or component inside a simulation test harness.

For a deployment objective, the code generator evaluates a model for:

- Valid code generation configurations
- Composition rules for nesting and peer relationships
- Relevant code generation workflows to enable

This figure shows the relationship between a top model intended for deployment as a component and its subcomponents. The top model contains referenced models that the code generator deploys as subcomponents of the parent component. The small gray squares represent interface ports. Subcomponent interface ports handle communication within the component, while component interface ports handle communication between components and with target environment software.



The code generator automatically determines the types of deployment that are relevant for a model and lists them as selections in the **Deployment Type** menu of the Embedded Coder app toolstrip. To facilitate incremental development, you can switch between deployment types. This table lists **Deployment Type** options that you can switch between depending on your initial deployment objective.

Deployment Objective	Deployment Type Options
Application	• `Automatic` (Default) • `Subcomponent`
Component	• `Component` (Default) • `Subcomponent`
Subcomponent	• `Subcomponent` • `Automatic` or `Component` (Default)
Utility	Not applicable

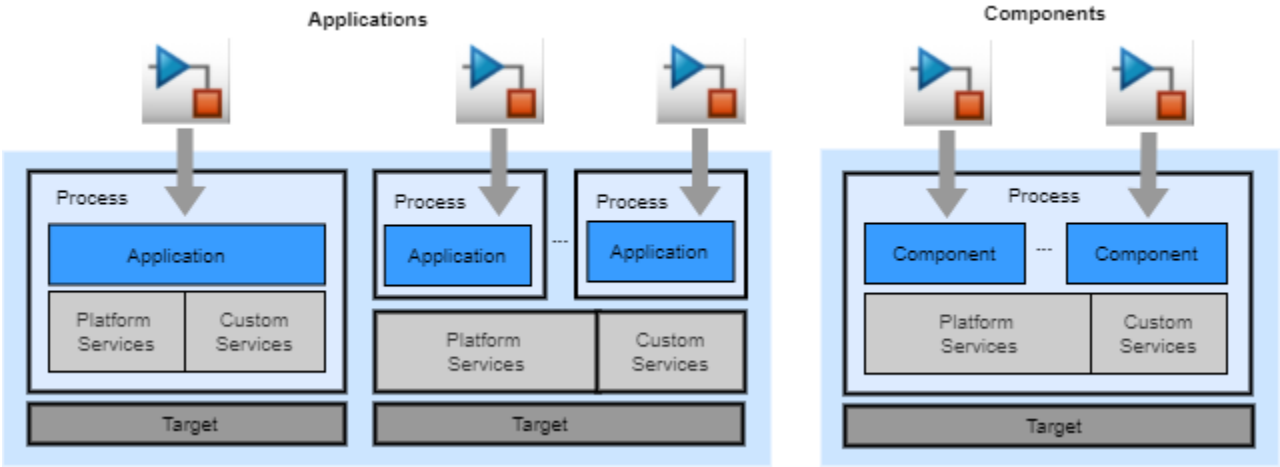
By default, **Deployment Type** is set to `Automatic` or `Component`, depending on the type of code interface configured for the model. You can change the setting to `Subcomponent`.

For more information, see “Check Deployment Types” on page 1-29.

For more information about designing models for application and component deployment, see “Architecture and Component Design”.

Applications and Components

You deploy code generated from Simulink top models for use in a target execution environment as applications or components. Applications can run in a single- or multiprocessing target environment. Components run in a single-process target environment.



For applications and components, the code generator produces a unit of code, a code descriptor file to facilitate integration, and a code interface report. For components configured to use service code interfaces, the code generator also produces a services header file (`services.h`).

The following table compares application and component deployment. Choose the type of deployment that best aligns with your project goals. Your choice determines how you design your model and how you set up coder tooling and configure the model.

Deployment Attribute	Application	Component
Scheduler	Generated code includes an example main program that applies the scheduling scheme (single-tasking, rate-monotonic scheduling (RMS), or concurrent) that is configured for the model and assumes how entry-point functions are assigned to tasks. You can use the generated scheduling and data transfer code as is, change it, or replace it. If you change or replace it, the implementation must match the scheduling assumptions made in the generated code.	Generated code does not make assumptions about the scheduling scheme used to execute the code. The code generator imposes modeling constraints to accommodate an external function scheduler.
Composition	Can include components, subcomponents, and utilities.	Can include subcomponents and utilities.

Deployment Attribute	Application	Component
Single-processing, multiprocessing	Generated code is optimized for a specific target device and runs on a single- or multiprocess architecture. In the single-processing case, application code is packaged and runs in the context of the same process as other target environment software. In the multiprocessing case, applications, target platform services, and custom services, run as separate processes.	You integrate generated component code with other components, platform services, and, if applicable, custom services to run as a single process on a single-process target architecture.
Process communication	An application implementation uses interprocess service interfaces (communicate data between processes) that the target environment implements.	A component implementation uses intraprocess service interfaces (communicate data within a process) that the target environment implements.
Interleaving of functions	Because the code generator produces code that meets specific scheduling requirements, you cannot interleave the generated application function with functions of other applications or components.	Because the code generator produces separate code modules for each function and the function code is portable from a scheduling perspective, you can interleave component functions with functions of different components.
Generated auxiliary code for customization and testing	The code generator combines algorithm code and the example main program with platform services and, if applicable, custom services to form a process.	The code generator produces an example service implementation for software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing. The example service implementation generated for SIL and PIL testing is designed to match simulation results.
Type of code interface	You define and apply a data interface configuration that consists of function customization templates and storage classes.	You define and apply a service code interface configuration that consists of function customization templates and well-defined programming interfaces and safeguards for maintaining data coherence for interacting with platform services.

Deployment Attribute	Application	Component
Project goal	You are deploying code to a single-process architecture for rapid prototyping, hardware-in-the-loop (HIL) simulation, or production code that runs as a complete, hard real-time application or you are deploying code to a multiprocess architecture for an application that requires high throughput or distributed execution.	Typically, deployment involves handing off the generated C source code module as-is for system integration.
Modeling guidelines		Modeling guidelines and Model Advisor checks are available to minimize model design rework.

For more information, see “Target Environment Services” on page 1-9 and “Code Interfaces and Code Interface Specification” on page 1-11.

For more information about designing models for application and component deployment, see “Architecture and Component Design”.

Subcomponents and Utilities

Subcomponents and utilities are units of code that you can produce and test independently, but are deployed within the context of an application or component. Identify subcomponents as you decompose your software problem (see “Divide Software Problem” on page 2-2). Whether you choose to use subcomponents or utilities in your design depends on considerations such as requirements for the generated code to be free of target environment context requirements.

A subcomponent is a model referenced by an application or component. Target environment software does not have direct access to execution and communication interfaces for subcomponents. Instead, the code generator exposes subcomponents as entry points that access target environment services by using a proxy. You can create multiple instances of subcomponents provided that the context of uses are equivalent. The code generator validates equivalence of the instances by using a checksum. If a subcomponent uses shared data and runs in multiple execution contexts, the subcomponent code must use data transfer services prescribed by the target environment.

A utility is a linked subsystem that adheres to specific modeling constraints. The code generator uses the constraints to produce code that is free of target environment context requirements. The generated code is part of a shared library, such as a dynamic link library (DLL), that is callable from anywhere. Unlike a subcomponent, a utility is not symbolically scoped to a top model and executes independently of top model services. You can use multiple instances of a utility provided that the context of uses are equivalent. The code generator validates equivalence of the instances by using a checksum.

Include utilities in model compositions when you want to:

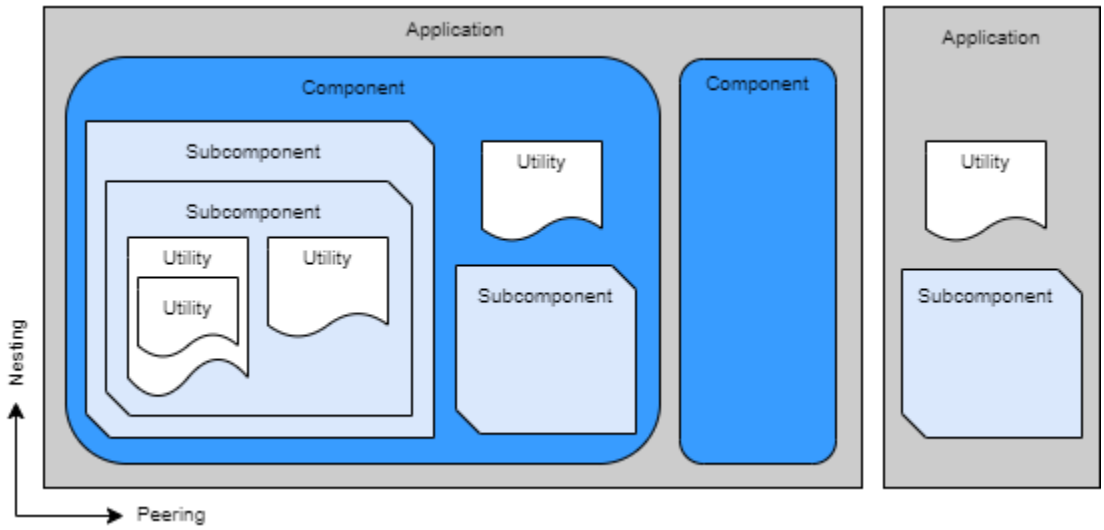
- Generate code for a set of reusable components that models can share.
- Lock down the interface for a subsystem.
- Make a model the owner of its generated code.

For more information about modeling for subcomponents and utilities, see “Architecture and Component Design”.

Nesting and Peer Composition Rules

When producing code, the code generator enforces nesting and peer composition rules based on deployment types configured for elements of a model hierarchy. Peer rules scope code symbolically to the same level of a software hierarchy. Nesting rules scope code to a parent in the software hierarchy.

This figure shows valid nesting and peer composition rules.



A deployable model composition can consist of:

- One or more applications that include components, subcomponents, and utilities.
- Components that include subcomponents, nested subcomponents, and utilities.
- Subcomponents, within applications and components. You can nest subcomponents.
- Utilities, within applications, components, and subcomponents. You can nest utilities.

Enforcement of the nesting and peer composition rules is important for running generated code in a constrained target environment. Applications interface as peers, components interface as peers, and each scope their data and functions according to software process constraints. The rules also facilitate reuse, testability, and top-down and bottom-up software development.

See Also

Related Examples

- “Check Deployment Types” on page 1-29
- “Architecture and Component Design”
- “Standards, Guidelines, and Block Usage”
- “Design Model Architecture”

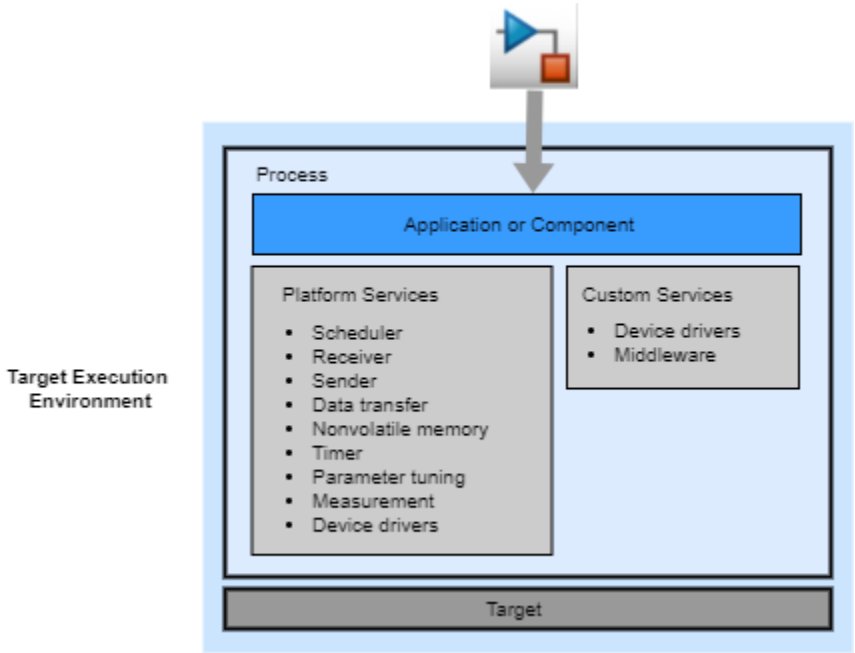
Target Environment Services

Target platform and custom services supplement tasks performed by an operating system to support the execution of applications and components. Service code insulates application code from target implementations. Platform services can consist of:

- Scheduling and task execution services
- Communication services, including support for device driver, send-receive, client-server, message-trigger and data transfer communication
- Auxiliary services, such as timer, nonvolatile memory, parameter tuning, and measurement services

Custom services include specialized device drivers and middleware.

This figure and the table that follows identify services with which embedded system application and component code typically interact.



Service	Description
Scheduler	Selects a task to execute based on a sample time (period or offset) or an event and execution order. Goals of the service include maximizing throughput, minimizing wait time, minimizing latency, maximizing fairness, and meeting deadlines.
Receiver	Receives input data from applications or components, nonvolatile memory, or the target environment.
Sender	Sends output data to applications or components, nonvolatile memory, or the target environment.
Data transfer	Handles sharing of data between callable functions. The service takes complete ownership of the shared data, including the setting of initial values.

Service	Description
Nonvolatile memory	Reads data from and writes data to nonvolatile memory.
Timer	Implements aspects to time, such as a clock tick with a specific tick resolution.
Parameter tuning	Enables access to and the ability to change parameter values while algorithm code executes.
Measurement	Enables access to and the ability read signal data while algorithm code executes.

When designing a model, in addition to aspects of the algorithm, consider how the application or component will interact with the target environment services. The interaction occurs through generated code interfaces. For the code generator to produce interface code that can interact with a service, you must provide the code generator with information about the target environment interface requirements, such as the service function name and data access method. Model inports, for example, represent requests to a target platform receiver service. You inform the code generator about how to generate interface code for the service requests by mapping the inports to a defined code interface.

See Also

Related Examples

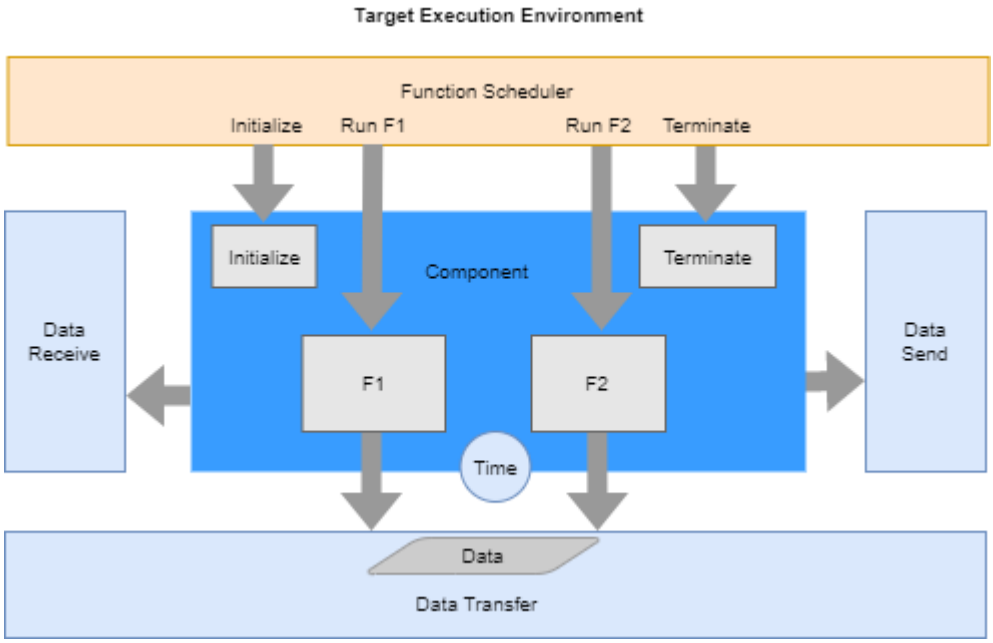
- “Code Interfaces and Code Interface Specification” on page 1-11
- “Application and Component Interfaces”

Code Interfaces and Code Interface Specification

Generated application and component code interacts with target environment services through these generated code interfaces:

- Entry-point functions that target software, such as a scheduler, can call
- Function calls to target software, such as platform service functions

This figure shows an example of interaction between a component and platform services in a target execution environment.



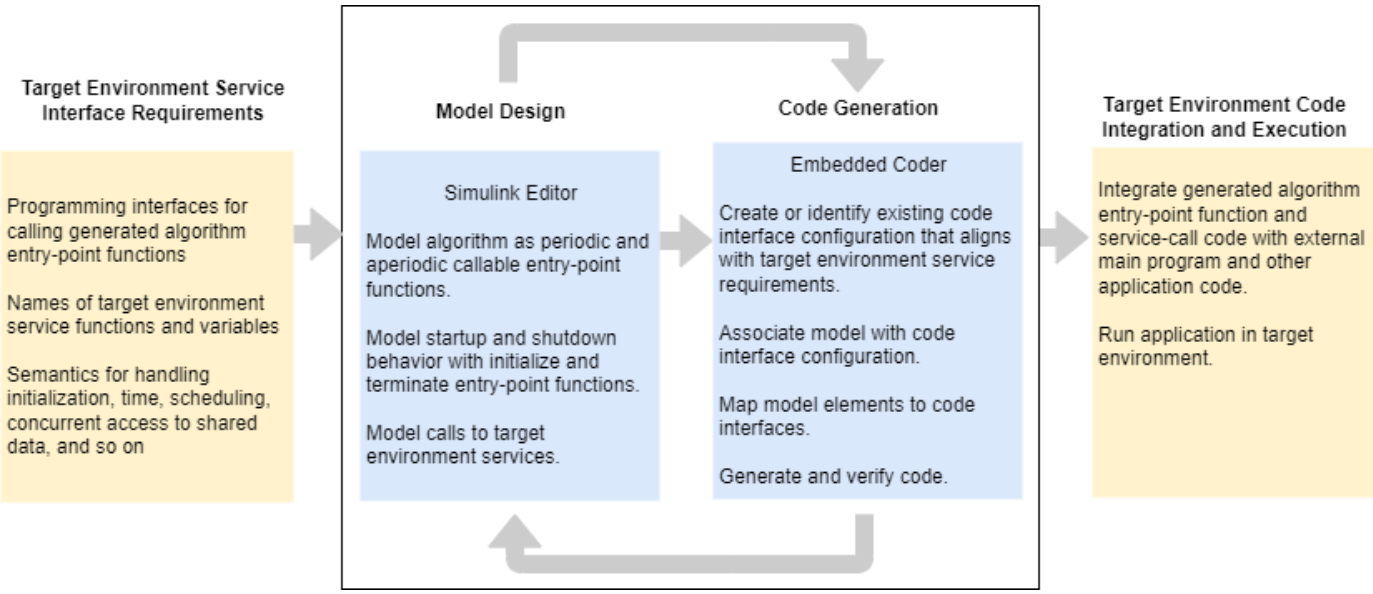
The arrows indicate the direction of function interface calls. The target function scheduler calls component entry-point functions: `Initialize`, `F1`, `F2`, and `Terminate`. The component calls target platform services to read input, write output, transfer data between callable functions, and get time values.

For the code generator to produce code that can interact with target environment services, you must make the code generator aware of service interface requirements by using a combination of:

- Model elements in your top model
- A code interface configuration
- Code mappings that associate the model elements with interfaces defined in the code interface configuration

The following figure shows the relationship among:

- Target environment service interface requirements (input to the deployment process)
- Code interface specification (how you use Simulink and Embedded Coder tooling to communicate the target environment interface requirements to the code generator)
- Target environment code integration and execution (what you do with the code generator output)



You inform the code generator about how to generate C or C++ interface code by mapping model elements that represent interfaces, such as functions and root-level inport and outports, to code interfaces that are defined in a code interface configuration file. For example, in the Simulink Editor, you can model an interface for reading data from a target platform receiver service by using a root-level Inport or In Bus Element block. For application deployment, you might also need to include custom blocks that handle data communication requirements. When designing a top model for component deployment, follow the recommended modeling guidelines in “Modeling Guidelines for Generated Code” on page 23-5.

The code generator supports two types of code interfaces: data and service. You set up the mappings in the Embedded Coder app by using the Code Mappings editor or by using the code mappings programming interface.

For C code interfaces:

- The code mappings reflect code interface choices that are defined in an Embedded Coder Dictionary. For a project or organization, someone defines a code interface configuration that aligns with interface requirements of a target execution environment. Then, for each model that you intend to deploy to that environment, you associate the models with that configuration.
- You can create and apply a data or service code interface configuration depending on your deployment objectives for your project models. For both types of code interfaces, you configure each type of callable entry-point interface as a function customization template, which defines how the code generator produces code for functions.

This table compares the two types of C code interfaces that you can define in an Embedded Coder Dictionary.

Data Interface	Service Interface
For application deployment.	For component deployment.

Data Interface	Service Interface
You configure each interface as a storage class, which specifies the appearance and placement of data elements in the generated code. Storage classes apply direct-access data communication.	For each interface, you specify function prototype information and data communication method requirements for the generated code. You can configure an interface for direct-access, outside-execution, or during-execution data communication, as required by the target platform.
	The code generator applies constraints when producing code so that it is easier to integrate the code for use in a specific target environment. For more information about the constraints, see “Service Interface Constraints and Limitations” on page 3-29.

If you need to use a data interface to configure a component for deployment, see the Component Deployment Guidelines for Embedded Coder Support Package. If you design and configure a component model according to the guidelines, the code generator produces code that aligns with interface requirements of a specific target platform.

For C++ code generation, you configure a model to use a data code interface. You do not create a code interface configuration. In the Embedded Coder app, you map functions that are represented in a top model to method prototypes. You map data elements that are represented in the top model to data visibility and member access function settings.

See Also

Related Examples

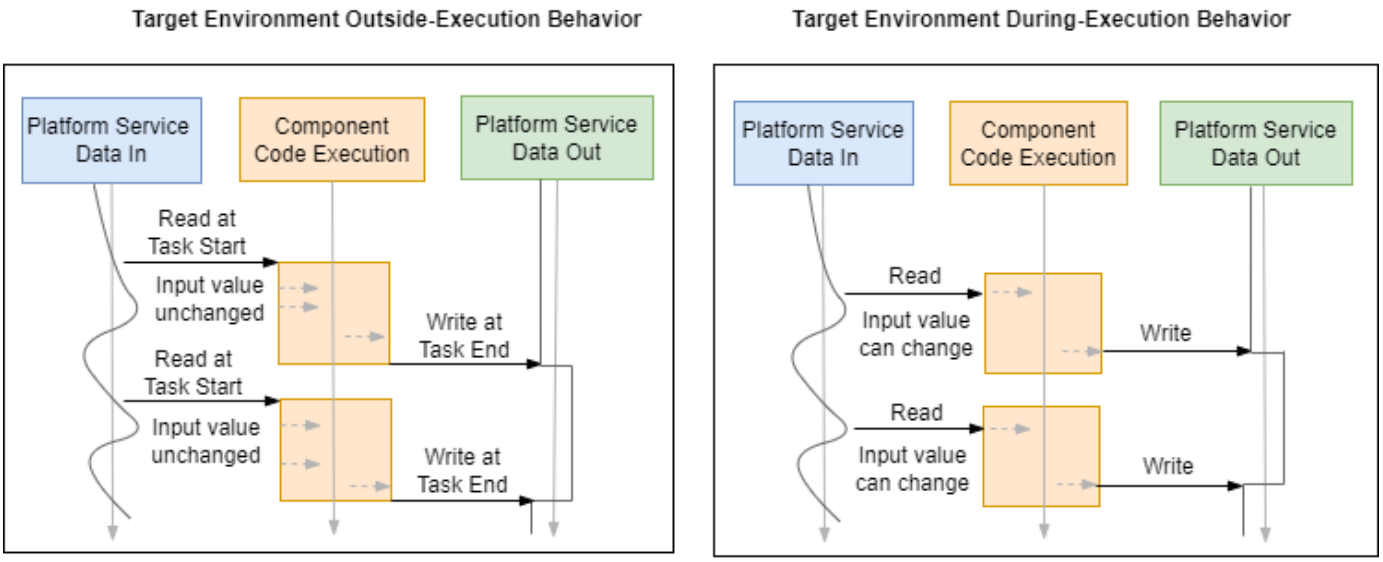
- “Model Code Interfaces” on page 3-2
- “Configure Code Interfaces for Code Generation” on page 3-4
- “Data Interfaces” on page 3-19
- “Service Interfaces” on page 3-21
- “Code Interface Configurations” on page 1-16
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Create a Service Interface Configuration” on page 32-11

Data Communication Methods

When designing an application or component model, for some target environment services, you must consider data communication methods that the services use. Data communication methods are direct-access, outside-execution, and during-execution. These methods coordinate the communication of data for tasks, including concurrent access to data. Generated function code must honor the data communication methods that target environment services use.

The direct-access method specifies that application or component code accesses data directly from memory provided by the target environment services.

This figure shows differences between the outside-execution and during-execution communication methods.



The gray dashed arrows show component code usage of communicated data. When communicating with a service that uses outside-execution data access, the values of input data remain unchanged as the component code executes. Each time the component code accesses the data, the code uses the same value. When communicating with a service that uses during-execution data access, the values of input data can change as the component code executes. Each time the component code accesses the data, the code might use a different value.

For each data communication method, this table describes the service behavior and the expected corresponding behavior of generated code that uses the service that applies the data communication method.

Data Communication Method	Target Environment Service Behavior	Generated Code Behavior
Direct access	• Provides functions that have direct access to data that is stored in target environment memory. Memory is persistent for the duration of a power cycle. • Assumes that generated functions do not access a memory buffer concurrently. Does not apply data concurrency safeguards. • Maximizes performance.	Communicates with other functions directly by using memory that the target environment manages.
Outside execution	• Applies data concurrency safeguards. • Communicates data to other functions outside (before and after) function execution. • Favors memory optimization over data freshness.	• Communicates with other functions indirectly. • Interacts with memory provided by the target environment.
During execution	• Applies data concurrency safeguards. • Communicates data to other functions immediately during function execution. • Favors data freshness over memory usage.	• Communicates with other functions indirectly. • Interacts with target environment services by using target environment local memory.

Design your model and configure model data elements based on the data communication methods that the target environment services apply.

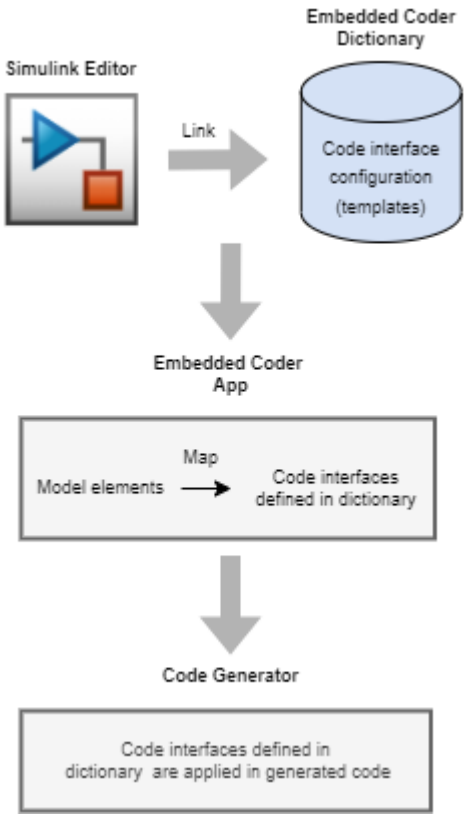
See Also

Related Examples

- “Service Interfaces” on page 3-21
- “Create a Service Interface Configuration” on page 32-11
- “Sample Target Environment Service Implementations of Data Communication Methods” on page 75-2

Code Interface Configurations

For C code generation, code interface configurations define templates that inform the code generator about requirements for the appearance, placement, and behavior of interface code. To generate interface code that aligns with specific target environment requirements, someone must define a code interface configuration for your project. A code interface configuration can be data or service oriented. Embedded Coder provides a default data interface configuration for generating code for application deployment. You can create custom code interface configurations for application and component deployment. Before generating code for a model, a modeler associates a code interface configuration that is defined in an Embedded Coder Dictionary with the model and, in the Embedded Coder app, maps model elements to code interfaces defined in the stored configuration. The code generator uses the mappings to substitute model-specific information into the interfaces defined in the code interface configuration.



You can create data or service interface configurations. This table compares the interfaces that you can define for each type of configuration.

Data Interface Configuration	Service Interface Configuration
Function customization template for initialize and terminate functions	Function customization template for initialize and terminate functions
Function customization template for periodic and aperiodic functions	Function customization template for periodic and aperiodic functions

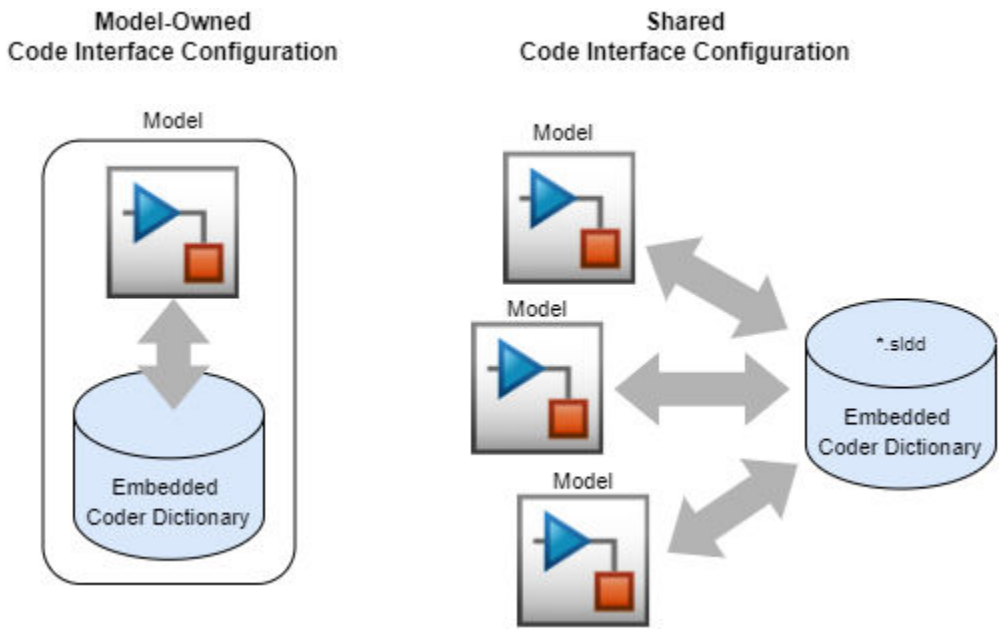
Data Interface Configuration	Service Interface Configuration
Storage classes for direct access to data in memory	Storage classes `SignalStruct` and `ParamStruct` and storage classes that you create for direct access to data in memory
Function memory section for placing callable entry-point function code in memory	Function memory section for placing callable entry-point function code in memory
Data memory section for direct access to data in memory	Data memory section for direct access to data in memory
	Receiver service interfaces
	Sender service interfaces
	Data transfer service interfaces
	Timer service interfaces
	Parameter tuning interfaces
	Parameter argument tuning interfaces
	Measurement interfaces

The code generator uses the code interface configuration to produce this interface code:

- Entry-point function prototypes for callable initialize and terminate functions
- Entry-point function prototypes for callable periodic or aperiodic functions
- Calls to target environment services
- Variable assignments
- Compiler `pragma` statements that identify areas of memory where functions and data are stored

Options for Code Interface Configuration Storage

The code interface configuration for a model resides in an Embedded Coder Dictionary. The coder dictionary can reside within the model (model-owned) or can be a separate `.sldd` file that models can share.



Model-Owned Embedded Coder Dictionary

By default, Simulink associates a model with a model-owned Embedded Coder Dictionary. A model-owned dictionary contains a code interface configuration that applies to the owner model only. You can customize the configuration. Use a model-owned dictionary for models of smaller scope and that do not share code interface requirements with other models. To customize the default model-owned dictionary, on the Embedded Coder app toolstrip, click **Code Interface > Embedded Coder Dictionary (Model)**.

Shared Embedded Coder Dictionary

A shared code Embedded Coder Dictionary:

- Resides in an `*.sldd`.
- Is independent of models that use it. A model and the dictionary that contains the code interface configuration are separate entities.
- Can be created from the Model Explorer.
- Can be attached to multiple models.

The separation of the model and dictionary enables you to:

- Standardize code generated from models designed across your organization for a specific target environment.
- Generate code that meets requirements of different target environments from a model.

Because the dictionary that contains a code interface configuration and the model are separate entities, modelers must attach the dictionary to models. Depending on how you create the dictionary, the attachment to a model might be set up automatically (see “Deploy Code Generation Definitions” on page 32-16). Once the dictionary is attached to the model, the modeler can map interface elements represented in the model to the configured interfaces (function customization templates, storage classes, and services).

To configure a model with a service interface, you must attach a shared coder dictionary that defines a service interface configuration to the model.

To create a shared Embedded Coder Dictionary, in the Model Explorer, click **New > Create Embedded Coder Dictionary** or in the Embedded Coder app, click **Code Interface > Set up shared Embedded Coder Dictionary**.

Code Interface Configurations for Data Objects

To create code interface configurations for data objects that you store in the base workspace or in a Simulink data dictionary, use the Custom Storage Class designer to create a configuration in a package. For these data objects, you apply code generation settings to the data object instead of applying them in the mapping of a model. When an item in a model uses the data object and its code generation settings, the code mapping for the item reflects the configuration of the data object.

If you want to use the same code interface configuration in model code mappings and in data objects, you do not need to store one copy of the configuration in an Embedded Coder Dictionary and another copy in a package. Instead, store the configuration in a package (by using the Custom Storage Class Designer). Then, configure one or more Embedded Coder Dictionaries to refer to the configuration in the package. When you want to make changes to the configuration, you make the changes in only one place, the package.

For more information, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.

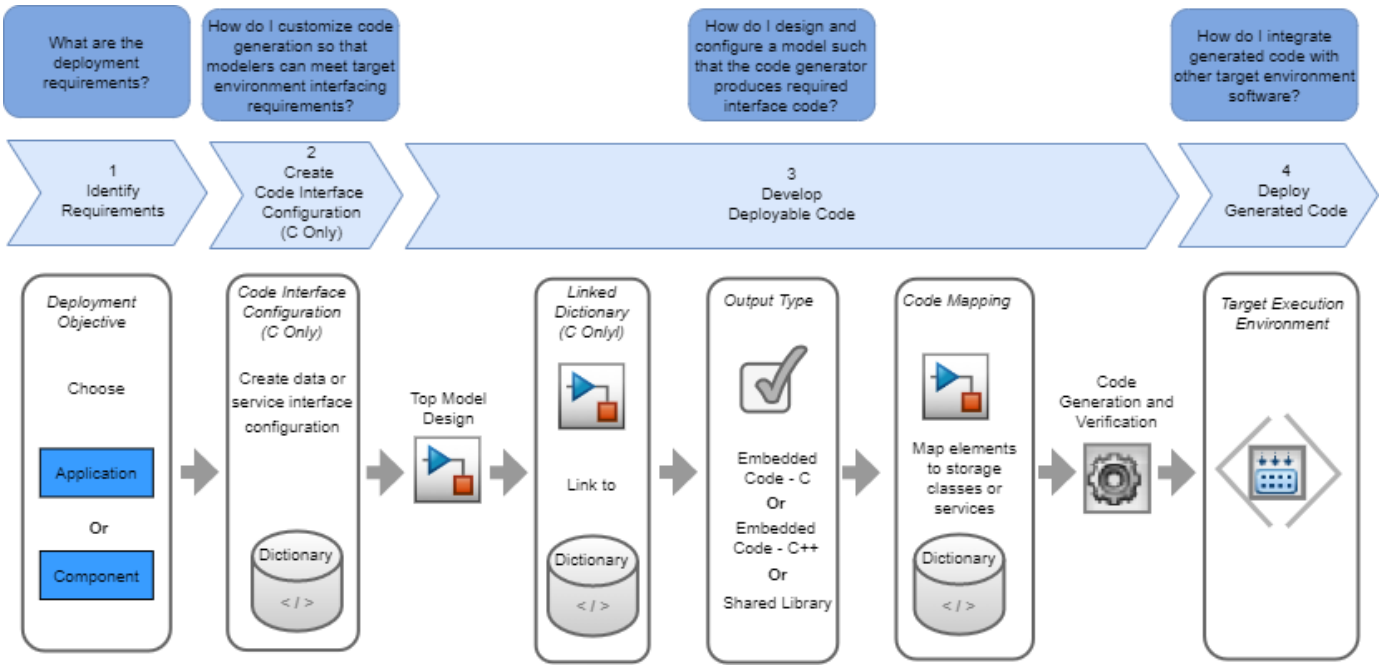
See Also

Related Examples

- “Code Interface Definitions”
- “Code Interfaces and Code Interface Specification” on page 1-11
- “Configure Code Interfaces for Code Generation” on page 3-4
- “Data Interfaces” on page 3-19
- “Service Interfaces” on page 3-21

Deploy Generated Software

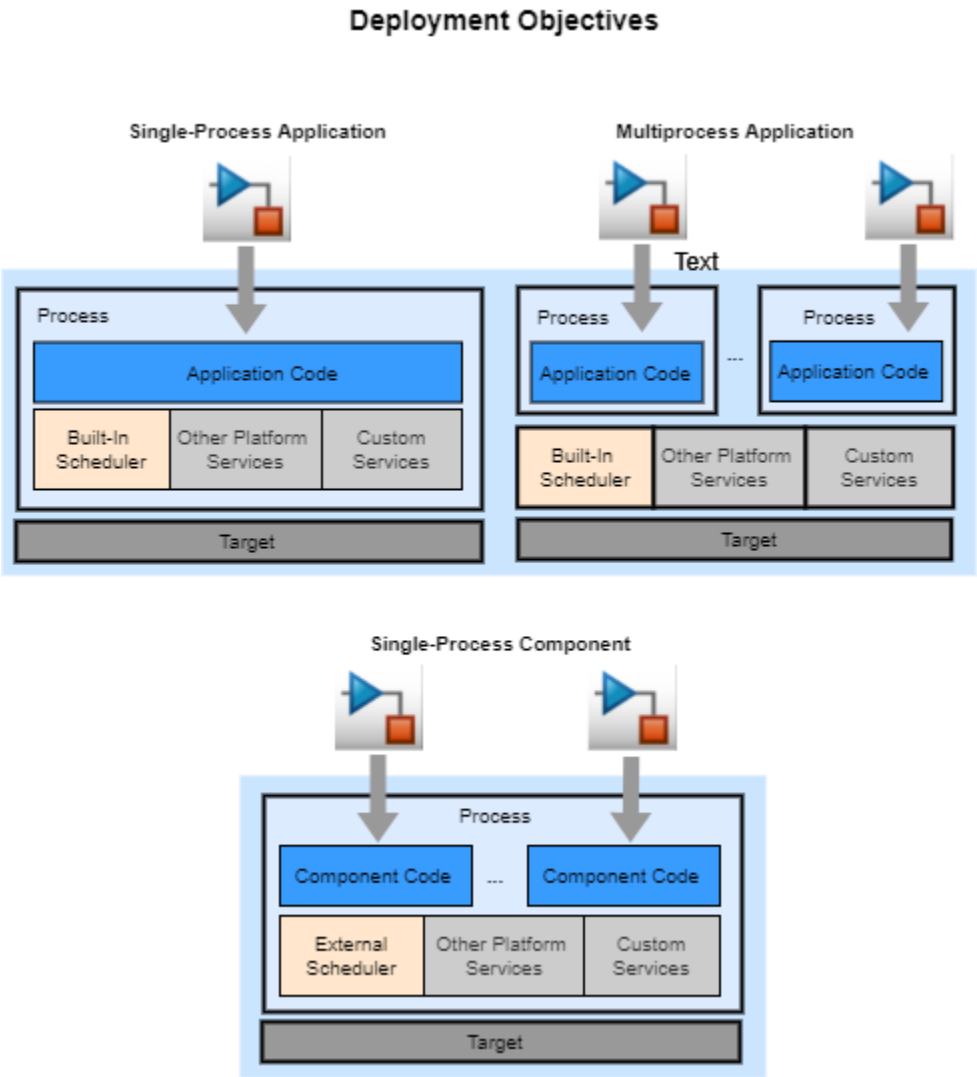
Simulink provides capabilities for designing top models and deploying them as generated code that behaves in a target execution environment like the model behaves during simulation. This figure and the following table shows the steps required for software deployment by using Simulink and Embedded Coder.



Step	Goal
Identify requirements	Assess project deployment requirements and determine whether your objective is to deploy an application or component for your project.
Create code interface configuration (C only)	Create a configuration that informs the code generator how to customize generated code so that modelers can meet target environment interfacing requirements.
Develop deployable code	Design a top model, attach an Embedded Coder Dictionary that defines a code interface configuration (C only) to the model, select an output type, map model elements to code interfaces, and generate and verify that the code generator produces required interface code.
Deploy generated code	Deploy application code that runs as a standalone executable program in a single-process or multiprocess target environment. Deploy component source code modules for integration with an external main program, other target environment software, and other components.

Identify Requirements

Identify the requirements for your project. The requirements help you determine whether to deploy your top model as an application or component. This figure shows how scheduling and processing architecture differ between the deployment objectives.



You will need to decide which of these deployment objectives meets scheduling and other target environment needs of your project. For application deployment, generated code includes an example main program that applies a single-tasking, rate-monotonic scheduling (RMS), or concurrent scheduling scheme. This scheme is configured for the model and determines how entry-point functions are assigned to tasks. For executing the algorithm code, you can use the generated scheduling and data transfer code as is, change it, or replace it. Meanwhile, component deployment requires integration of generated function code with an external function scheduler and other target environment software.

Code that you generate for application deployment can run on a single-process or multiprocess architecture as part of a real-time or non-real-time application, while code for component deployment can only run on a single-process architecture. For a single-process architecture, the generated code

is optimized for a specific target device and meets hard real-time requirements. Multiprocess deployment addresses requirements of high-throughput and distributed execution applications.

For a list of requirements and conditions that apply to each deployment objective, see “Software Deployment Objectives and Deployment Types” on page 1-2. Use that information to identify the deployment type that best aligns with your project requirements.

For more information about assessing project requirements, see “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3.

Create C Code Interface Configuration

After researching C code interface requirements, you create a code interface configuration by using the Embedded Coder Dictionary. You can create data and service code interface configurations, as “Code Interfaces and Code Interface Specification” on page 1-11 and “Code Interface Configurations” on page 1-16 describe. The interfaces that you create as part of a code interface configuration depend on your project requirements and deployment objective.

For details about creating code interface configurations, see “Code Interface Definitions”.

Research for Code Interface Requirements

Before you can define a custom code interface configuration that aligns with specific target environment requirements, you need to gather requirements. Research the required application or component interactions with your target environment software. Identify sources of input and required output. Determine what the code for interfacing with other target environment resources must look like and how it must behave. For example, does your model or generated code need to exchange data with target environment services? If so, map inputs, outputs, and parameters to corresponding service interfaces. Typical service interfaces involve function arguments and return values, global variables, and access functions.

The following questions can help you identify interface requirements that are relevant to application or component deployment. Answers that you gather serve as input when defining a code interface configuration.

Category	Questions
Target environment services	• Does the target environment use standard platform middleware, such as AUTOSAR or DDS? For information about AUTOSAR or DDS, see “AUTOSAR Blockset” and “DDS Blockset”. • What target environment software will the deployed code interact with? For example, does the target environment include an operating system? What target services are available? A code interface configuration defines interfaces for the services that you identify. • Does the target software expect the deployed code to handle startup or shutdown events? If so, the code interface configuration should define interfaces for initialize and terminate functions. • Does the target software have naming requirements for entry-point functions that it calls to initiate execution of deployed code? If so, the code interface configuration should specify function names or function naming rules that align with the requirements. • What are the names of the functions that the deployed code needs to call to use target services? The code interface configuration should specify function naming rules that align with the requirements.
Data communication	• What data communication methods do target services use for handling concurrent access to data when communicating with deployed code (direct access, outside execution, or during execution)? The code interface configuration should define interfaces that use data communication methods that align with those used by the target environment services.
Data types	• Does the target environment expect the input and output of deployed code to be represented as arrays, structures, or enumerated types? In the Simulink environment, you can represent these types as vectors, buses, and `IntEnum`, respectively. • Is fixed-point support required? If you use the Simulink fixed-point interface, you can scale and specify offsets. • Does the target environment software use organization-specific data types? If yes and you have Embedded Coder software, create alias types to represent those types. The code generator uses the alias types in the code that it produces. For example, once defined, you can specify an alias type in a function prototype, for a temporary variable, or for block output.

Category	Questions
Memory usage and latency	• Does the deployed code need to access shared memory? If yes, define and use memory sections in the code interface configuration. • Do you need to deploy code that minimizes latency or optimizes memory usage when sending, receiving or sharing data? Does the target environment require that deployed code optimize memory usage when sending data? Define and map model elements to code interfaces that use data communication methods that align with the requirements.
Parameter tuning and data measurement	• Does the target environment require access to variables that represent parameters for tuning or data for measurement purposes? If so, how does the target environment expect the data to be represented in the deployed code? Does the data need to reside in a specific header file? Define the interface to align with the requirements. • Does the application code running in the target environment consist of multiple deployed source code modules developed independently and integrated manually with a target function scheduler? Or, is the application code a generated executable program that runs standalone on the target device (single-processing) or as a collection of executable programs that run concurrently in a multiprocessing architecture?
Deployment objective	• Does the generated code running in the target environment consist of multiple deployed source code modules developed independently and integrated manually with a target function scheduler? Or, is the code a generated executable program that runs standalone on the target device (single-processing) or as a collection of executable programs that run concurrently in a multiprocessing architecture?

For some projects, these requirements are captured in an interface control document (ICD). An ICD provides the details about the interfaces between deployed units of code and target environment services.

Decide On a Type of Interface Configuration and Storage

Decide whether you are creating a data or service interface configuration and whether the configuration will be model-owned or shared.

Goal	Data Interface	Service Interface	Model-Owned with Data Interface	Shared
Code interfacing for application deployment.	X		X	
Code interfacing for component deployment	X	X	X	

Goal	Data Interface	Service Interface	Model-Owned with Data Interface	Shared
Enable mapping of model elements to storage classes and specific property settings (for example, to support legacy models).	X		X	
Customize interface code to include built-in safeguards for maintaining data coherence for outside-execution or during-execution data communication.		X		
Customize code for data transfers between functions outside of (before and after) function execution.		X		
Support accessing time values in tasks.		X		
Enable mapping of model elements to complete interface abstractions.		X		
For models of smaller scope or that do not share code interface requirements with other models.			X	
Enable multiple models to attach the same shared Embedded Coder Dictionary and use the same code interface configuration defined in that dictionary.				X
Standardize interface code generated from models designed across and organization for a specific target environment.				X
Enable code generation from a model that meets requirements of different target environments.				X

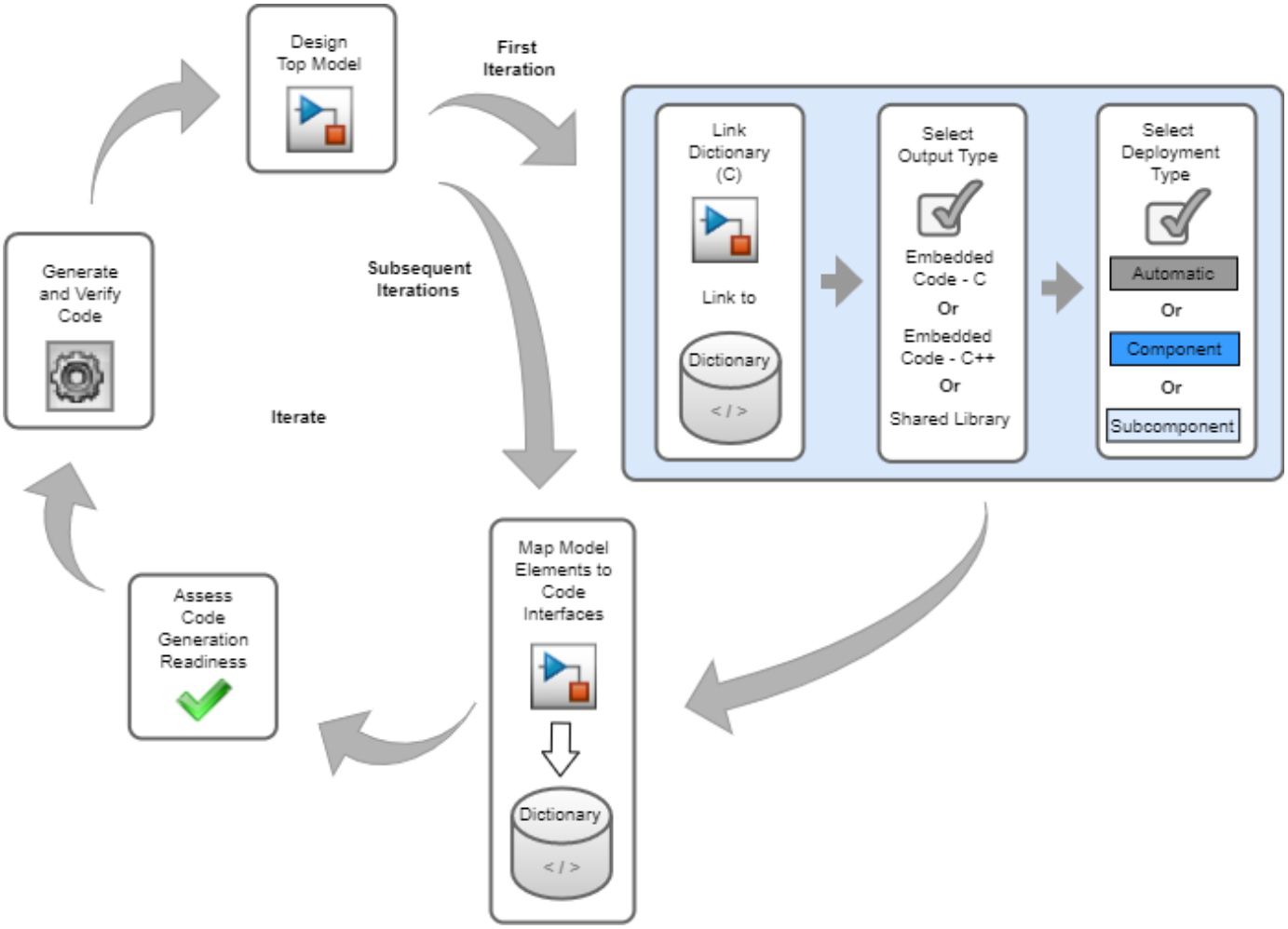
To customize the default model-owned dictionary, on the Embedded Coder app toolstrip, click **Code Interface > Embedded Coder Dictionary (Model)**.

To create a shared Embedded Coder Dictionary, in the Model Explorer, click **New > Create Embedded Coder Dictionary** or in the Embedded Coder app, click **Code Interface > Set up shared Embedded Coder Dictionary**.

For more information see “Code Interfaces and Code Interface Specification” on page 1-11, “Code Interface Configurations” on page 1-16, “Configure Code Interfaces for Code Generation” on page 3-4.

Develop Deployable Code

This figure shows the steps for developing a top model that you intend to deploy as generated code.



To deploy code from a top model, design your top model by decomposing the algorithm into subcomponents and identify relevant utilities. Choose and implement a modeling style and modeling patterns that align with how you intend to deploy the generated code. Apply relevant modeling guidelines.

When you are ready to generate code, complete this one-time code generation setup. You can optionally use a model template to create a model that is automatically configured according to this setup.

1. If relevant, attach a shared Embedded Coder Dictionary to the model. This step is required when you are deploying a component that uses a service code interface. This step is not relevant when you are deploying C++ code.

2. Select an output type for code generation. Depending on the deployment goals for a model, an output type specifies one or a combination of the system target file to apply, programming language of the output, or platform middleware, such as AUTOSAR or DDS. For information about AUTOSAR or DDS, see “AUTOSAR Blockset” and “DDS Blockset”.
3. Check whether the deployment type displayed in the Embedded Coder toolstrip aligns with your expectations. Typically, you can accept default deployment type settings.

Deployment Objective	Deployment Type Setting
C application deployment	`Automatic`. The code generator produces deployable application or component code based on the model contents and configuration. For example,for an export-function model that is configured with a data interface, the code generator produces deployable component code. The code generator automatically configures reference models as subcomponents.
C component deployment	`Component`. The code generator automatically configures reference models as subcomponents.
C++ code generation	Not applicable.

Then, iterate through these steps:

1. Map model elements to code interfaces. For example, for the C language, available code interfaces are defined in the model Embedded Coder Dictionary. The mappings inform the code generator about appearance, placement, and behavior requirements of the code for model elements.
2. Use the Model Advisor to check whether the model is ready for code generation. Use the Code Generation Advisor to check and configure a model for code generation application objectives, such as traceability or debugging.
3. Generate code and confirm that the output meets the code interface requirements.
4. If necessary, adjust your model design and repeat steps 1 to 3.

Modeling guidelines and tooling facilitate development of top models that you intend to deploy as:

- C function source code modules that are integrated with a target environment function scheduler and other target services that run in a single-processing environment.
- Standalone executable program that replicates the simulation scheduler and optimizes scheduling code for single-tasking, rate-monotonic multitasking, or concurrent execution, depending on the model configuration. The executable program runs as a single-process or multiprocess application. Code for a single-process application is intended to run on a specific target device.

When deploying a component, you develop each component in isolation. Then, you or someone else in your organization integrates the generated function code into a complete application program at the source code level.

For models that you deploy as components that use a data interface, see the Component Deployment Guidelines for Embedded Coder Support Package, which provides guidelines and tools to help produce code that aligns with these aspects of target platform services:

- Interface of generated callable entry-point functions
- Interface of calls to platform services
- Data communication methods

Design Model

You can design models for code generation where the code generator uses code interface mappings to produce code that aligns with target service requirements.

To start the design work for a deployment project, partition the project as an application or into components. For each application or component identify subcomponents and potential utilities.

High-level code interface requirements of an application or component model can include:

- Responses to startup and shutdown events
- Scheduling of periodic and aperiodic rates
- Communication service interfaces that safely handle access to data, including data that is shared by tasks or threads that execute concurrently
- Nonvolatile memory service interfaces
- Timer service interfaces
- Programming interface customizations for accessing data

The deployment type of a model determines how the model constructs map to interfaces for services provided by the target environment and the rules for peer interaction and scope for data and function accessibility for models in the hierarchy. Each model in the hierarchy gets associated with a deployment type based on the role of the model. For component deployment with a service interface configuration, the code generator sets the deployment type to `Component`. Otherwise, the code generator sets it to `Automatic`. In either case, for incremental development, you can change the setting to `Subcomponent`.

Choose and implement a modeling style and modeling patterns that align with how you intend to deploy the generated code. Apply relevant modeling guidelines. For information on modeling styles, see “Periodic and Aperiodic Function Interfaces” on page 3-6. For information on modeling guidelines, see “Standards, Guidelines, and Block Usage”.

The Component Deployment Guidelines for Embedded Coder Support Package provides component modeling guidelines for deployment components that use a data interface. The guidelines help you design component models from which the code generator produces code that can integrate and interface with platform services and other component code that runs on a target platform. The guidelines identify model elements and model element code configurations for execution and other service requirements. When you design and configure a component based on the component modeling guidelines:

- The code generator produces a collection of callable entry-point functions that align with target scheduler service calls.
- Within the generated function code, calls to target platform service functions align with the target service programming interfaces.
- The generated function code honors data communication methods that target platform services use.

For more information, see “Architecture and Component Design”.

Attach Dictionary

If you choose to use a shared Embedded Coder Dictionary, you must attach the dictionary to your model. When you are deploying a component that uses a service code interface, the code generator requires use of a shared Embedded Coder Dictionary. Embedded Coder Dictionaries are not relevant when you are deploying C++ code.

To attach an Embedded Coder Dictionary to a model, in the Embedded Coder app, click **Code Interface > Set up shared Embedded Coder Dictionary**. In the dialog box that appears, click **Open**, browse and select a dictionary (SLDD) file that contains the code interface configuration, and click **OK**. Alternatively, you can create a coder dictionary and attach it to the model.

For more information about creating and attaching Embedded Coder Dictionaries, see **Embedded Coder Dictionary**.

Select Output Type

When using the Embedded Coder app, before you generate code for a loaded model, you must select an output type. Available output types are listed in the toolstrip **Output** menu. For more information, see “Code Generation Output”.

Check Deployment Types

For C code generation configurations, in the Embedded Coder app toolstrip, the **Deployment Type** field shows the deployment type of the currently loaded model. Typically, you do not need to adjust the deployment type setting. The code generator sets the deployment type based on the model style and code interface configuration. For models configured with a data interface configuration, the code generator sets the field to `Automatic`. If you want to generate code for a subcomponent of a model, you can change the setting to `Subcomponent`.

If the code generator determines that you are deploying a component that uses a service code interface, for models in the component model hierarchy, you can configure the deployment types. By default, the code generator sets the deployment type of the loaded model to `Component` and models referenced by that model to `Subcomponent`. For each model in the model hierarchy, you can select a deployment type based on the role of the model.

Role of Model	Deployment Type
Publish code interfaces according to the code interface configuration associated with the model. Code generated from the model can interact with target environment software directly.	Component
Referenced by and runs as part of a parent component model. Execution and communication interfaces are accessible within the context of the parent model only. Generated code is not accessible to target environment software.	Subcomponent

To review or set the deployment types for models throughout the model hierarchy, from the **Deployment Type** menu, select **Set Up Deployment Type for Model Hierarchy**. For more information, see “Configure C Code Deployment Types for Model Hierarchy” on page 43-29.

Map Model Elements to Code Interfaces

In the Embedded Coder app, use the Code Mappings editor to map model interface elements to code interface elements. Alternatively, you can use the code mappings programming interface. The code

interface elements to which you can map model elements depend on the type of code interface configuration that is associated with the model (data or service). For C code generation, available code interface elements are defined in an Embedded Coder Dictionary that is associated with the model.

For more information, see:

- "Configure Code Interfaces for Code Generation" on page 3-4
- "C Code Generation Configuration for Model Interface Elements" on page 33-2
- **Code Mappings editor**
- `coder.mapping.api.CodeMapping`
- "Interactively Configure C++ Interface" on page 44-2
- "Programmatically Configure C++ Interface" on page 44-25
- `coder.mapping.api.CodeMappingCPP`

Assess Code Generation Readiness

Verify that your model is ready for code generation by using the Model Advisor and Code Generation Advisor. The Model Advisor checks to assess readiness to generate code. General Embedded Coder checks and checks for component deployment of models configured with a service interface configuration are available in Simulink Check™. Checks for component deployment of models configured with a data interface configuration are available as part of the Component Deployment Guidelines for Embedded Coder Support Package.

The Code Generation Advisor checks your model for code generation application objectives such as traceability or debugging.

For information about using Model Advisor, see:

- "Check Your Model Using the Model Advisor"
- "Embedded Coder Checks"
- Component Deployment Guidelines for Embedded Coder Support Package

For information about using the Code Generation Advisor, see "Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 49-2.

Generate and Verify Code

You have different options for generating code depending on whether you are generating code for an application or component model. For an application model, by default, the code generator produces source code for algorithms represented in your model and a `main` function that handles function scheduling. Then, the code generator compiles and links the generated code into an executable program. Alternatively, you can generate source code and suppress compiling and linking by selecting model configuration parameter **Generate code only** and customizing the process for producing an executable program.

For a component model, you can generate source code and suppress compilation and linking or you can generate a component model library. The code generator produces build artifacts, including a header file `services.h`, in folder *`codeGenerationFolder/modelBuildFolder/`*`services`, and a MAT-file `buildInfo.mat`, in folder *`codeGenerationFolder/modelBuildFolder/`*`services/lib`. The `services.h` file, which you can rename in the Embedded Coder Dictionary, specifies function prototypes for target platform services. The `buildInfo.mat` file stores an `RTW.BuildInfo`

object that contains information for compiling the generated code and producing a library. By default, the code generator produces source code and does not perform compilation and linking. To generate a library, clear model configuration parameter **Generate code only**. If you configure a component model for library generation or to run as a software-in-the-loop (SIL) or processor -in-the-loop (PIL) simulation, the code generator builds a component model library according to the model toolchain configuration. SIL or PIL simulations verify the code that the library contains.

When you generate source code without compiling and linking, you can build an executable program for an application or a component model library outside of the MATLAB® environment by using a CMake workflow. You can create a:

- CMake configuration (`CMakeLists.txt`) file by using the `codebuild` function
- ZIP file for transferring files to another development environment by setting model configuration parameter **Package code and artifacts** and using the `packNGo` function

For an example, see “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24.

To verify whether the generated code aligns with code interface requirements, use the code view pane in the coder app and the generated Code Interface Report. Several other tools are available for other types of verification, such as profiling code execution, testing for numeric equivalence between simulations and code execution, and checking for unintended functionality.

To explore verification of code generated from component models, you can use the test harness example models provided in “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89 and “Deploy Rate-Based Component Configured for C Service Interface Code Generation” on page 3-103.

For more information, see:

- “Source Code Generation”
- “Generated Code Compilation”
- “Verification, Testing, and Certification”

Deploy Generated Code

After you generate and verify code, deploy it. How you deploy the generated code depends on whether you are deploying an application or component.

Deployment Type	Deployment Goals
Application	• Run generated executable program in the target environment as a standalone application. • Integrate generated executable program to run with other code in the target environment. • Customize generated source code to integrate with other code that runs in the target environment. In this case, you might use the generated example `main` as a starting point for developing a custom `main` program.
Component	Integrate generated component source code or component model library with other code that runs in the target environment.

Application Deployment Workflow Options

For application deployment, by default, the code generator compiles and links generated source code into an executable program that you can download to a target execution environment to run. If you choose to configure the code generator to produce source code without compiling and linking, you can:

1. Customize or replace the generated `main` function. You must take care to align the code interfaces with the interfaces generated for the model algorithm code.
2. Produce an executable program by using the `codebuild` command to compile and link the generated code.

See “Manage Build Process Folders” on page 52-30, “Approaches for Building Code Generated from Simulink Models” on page 59-11, and “Compile Code in Another Development Environment” on page 52-99.

Component Deployment Workflow Options

For component deployment, you have multiple workflow options. In each case, you must integrate your generated code with an external `main` program that handles function scheduling.

You can:

- Integrate generated source code with an external `main` function and other target environment source code and build an executable program.
- Link a generated component model library with an external `main` function and other target environment code to create an executable program.
- Use the `codebuild` command to build a component model library from generated source code. Then, within or outside of the MATLAB environment, link the library with an external `main` function and other target environment code to create an executable program.

See “Manage Build Process Folders” on page 52-30, “Configure CMake Build Process” on page 59-13, “Approaches for Building Code Generated from Simulink Models” on page 59-11, and “Compile Code in Another Development Environment” on page 52-99.

Integrate Generated Component Code with Target Environment Code

For an embedded system, the `main` function:

- Defines function interfaces (function prototypes).
- Includes files that declare external data.
- Defines `extern` data.
- Initializes data.
- Uses simulated or actual hardware.
- Calls algorithmic functions.

To integrate generated component code with a `main` function and other target environment code, you must:

- Match the data and function interfaces of the generated code with other interfaces of existing system code.

- Connect input data.
- Connect output data.
- Access other data, such as block state values, local parameters, and time.

Design and configure the component model such that generated code aligns with the code interfaces of the target environment. For example, if your target environment software defines a global variable for storing output data and the generated code reads that data as input, you can configure the corresponding inport so that the generated code interacts with the existing variable. The target environment software that calls the entry-point functions must provide input signal data and, depending on your application, scheduling information. Also, suppose that the generated algorithm calculates output data that the calling environment can use. The target environment software and the algorithm can exchange this data by using global variables or formal parameters (arguments). The set of input and output data and data access mechanisms constitute the interfaces of the entry-point functions.

When you understand the code interfaces and how to control them, you can:

- Generate code that can be called by other component code and target services.
- Write code that calls the generated code.
- Generate reusable (reentrant) code that you can call multiple times in a single application.
- Integrate the generated code with other, external code in your application.

In a model, root-level inports and outports represent the primary inputs and outputs of the component algorithm. By default, the code generator aggregates this input and output data into standard structures.

For examples, see “Configure Generated Code According to Interface Control Document Specifications” on page 40-54 and “Integrate External Application Code with Code Generated from PID Controller” on page 40-45.

For more information, see “Deployment, Integration, and Supported Hardware”.

Review Generated Interface Code

Confirm that the code generator produces expected interface code by reviewing the Code Interface Report. The report describes the generated code interface, including interface information for:

- Each component model entry-point function (function name, prototype, timing information, arguments, return value, and header file)
- Each model root-level inport and root-level outport (block name, code identifier, data type, scaling information, and dimensions)
- Tunable parameters that are associated with the model (data store source, code identifier, data type, scaling information, and dimensions)
- Global data stores and data stores with non-Auto storage that are associated with the model

For models configured for component deployment with service interfaces, the report includes information for:

- Each periodic and aperiodic callable entry-point function (prototype, header file, simulation rate, and service call prototypes)
- Initialize and terminate callable entry-point functions (prototype and header file)

- Each type of service (function prototypes, data communication method, function names, data types, dimensions, and header file)

Configure the code generator to produce a Code Interface Report as part of the generated HTML report.

For more information about the report, see “Analyze Generated Data Code Interface” on page 54-13.

Get Meta Information About Generated Code

By default, the code generator creates a code descriptor file (`codedescriptor.dmr`) in the build folder. That file contains meta information about the generated code, including:

- Data interfaces (inports, outports, parameters, data stores, and internal data)
- Function interfaces (initialize, output, update, and terminate)
- Execution information for the data and function interfaces, such as timing requirements
- Model hierarchy information and information for referenced models

You can use the code descriptor programming interface to get access to the contents of the code descriptor file and use the results to confirm that generated interfaces meet integration requirements. You can also use the programming interface to provide input to tools that generate interfaces for target platform services.

For more information, see “Get Code Description of Generated Code” on page 54-44 and `coder.codedescriptor.CodeDescriptor` class.

Identify and Package Files for Deployment

For component deployment, after you review and, if necessary, refine the code interfaces generated for a model, identify files required to build the integrated code. You can generate code for a component model and compile a component model library for deployment by clearing model configuration parameter **Generate code only** and initiating a model build. If code for the component model is already generated, you can build a component model library by using the `codebuild` command with the path *`codeGenerationFolder/modelBuildFolder`*`/services/lib`. You can link the generated component model library with a `main` function and other target environment code to create an executable program.

When you generate source code without compiling and linking, you can build a component model library outside of the MATLAB environment by using a CMake workflow. You can create a:

- CMake configuration (`CMakeLists.txt`) file by using the `codebuild` function. See “Configure CMake Build Process” on page 59-13.
- ZIP file by setting model configuration parameter **Package code and artifacts** and using the `packNGo` function. The `packNGo` utility finds and packages files that you need to build an executable image. Then, you can relocate the ZIP file to another development environment, such as a dedicated application build system. See “Relocate or Share Generated Code” on page 59-56.

Build Integrated Code Into Executable Program

Embedded Coder provides extensive capabilities for customizing the build process. For example:

- Choose between a CMake, toolchain, or template makefile approach. See “Approaches for Building Code Generated from Simulink Models” on page 59-11.

- Compile a component model library that you can link with a `main` function and other target environment code to create an executable program. See “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24.
- Control compiler optimizations. See “Control Compiler Optimizations” on page 76-6.
- Reduce build time by building referenced models in parallel. See “Reduce Build Time for Referenced Models by Using Parallel Builds” on page 59-51.
- Customize build processing that occurs after code generation to include third-party tools. See “Customize Post-Code-Generation Build Processing” on page 93-7.
- Compile code in another development environment. See “Compile Code in Another Development Environment” on page 52-99.

See Also

Related Examples

- “Software Component Modeling”
- “Code Interface Definitions”
- “Source Code Generation”
- “Code Generation Output”
- “C Data and Function Interfaces”
- “C++ Data and Function Interfaces”
- “Deployment, Integration, and Supported Hardware”

Code Generation Goal Summary

Use this table to find Simulink Coder™ Embedded Coder information based on your project goals.

Goals	Related Information
Get familiar with the product by using tutorials and learning about concepts.	• "Get Started with Embedded Coder" • "Embedded Coder Fundamentals"
Avoid or minimize project rework by considering code generation from the beginning of the model design process • Divide software problem into parts and map aspects of your system architecture to Simulink modeling environment elements. • Model software interfaces . • Modularize design by using Simulink component-based modeling options and reuse techniques. • Import external MATLAB®, C, or C++ code. • Design timer and scheduling support. • Design models based on recommended guidelines and for generating code that complies with industry standards. • Use blocks for embedded processors and operating systems.	"Architecture and Component Design"
Generate code from model design. • Configure code interfaces. • Check whether model is ready for code generation. • Generate source code. • Compile generated source code and link object code. • Generate shared libraries and reusable utility code. • Generate reports of code generation results.	"Code Generation"

Goals	Related Information
Deploy generated code. • Integrate generated callable function code into a main program. • Adapt a generated program to run without an external real-time executive or operating system on target hardware. • Integrate a generated program with an operating system or board support package. • Relocate static and generated code files for a model to another development environment. • Use a generated shared library in application code. • Package and share artifacts generated for a model with a third party without revealing intellectual property. • Run accelerated simulations and deploy standalone simulations outside of the MATLAB and Simulink environment.	“Deployment, Integration, and Supported Hardware”
Optimize generated code by minimizing memory requirements and speeding up execution. • Optimize use of RAM, ROM, and stack space by generated code. • Improve execution speed of generated code. • Reduce data copies. • Apply code replacement libraries. • Make tradeoffs when generating optimized code. • Generate code for high-throughput computing applications.	“Code Efficiency”
Customize generated code and code generation tools for project or organization. • Customize data representation to ease integration with external code and support interactivity and instrumentation. • Create code interface definitions for standardization and software architecture conformance. • Create and maintain code replacement libraries. • Customize how the code generator produces code. • Define code generation behavior for custom blocks. • Use the Target Language Compiler to customize generated code, inline code generated for custom blocks, and generate additional or different types of files. • Capture target hardware information for MathWorks® software. • Provide the code generator with information about a target platform and required code features. • Customize the compilation stage of the build process. • Create software for deploying MATLAB and Simulink designs to hardware platforms.	“Code and Tool Customization”

Goals	Related Information
Verify and test generated code. • Monitor signals and tune model parameters in your target application. • Profile code execution to determine whether generated code meets performance requirements. • Test numeric equivalence. • Test integrated component software in the target environment. • Check for unintended functionality by comparing model and code coverage. • Verify generated code by using code analysis tools. • Qualify Embedded Coder for IEC certification.	“Verification, Testing, and Certification”
Integrate generated code from previous releases with generated code from the current release.	“Cross-Release Support”
Generate C/C++ code from MATLAB code for embedded systems. • Control the appearance of identifiers, code style, comments, and function and file banners in generated code. • Control the representation of global variables. • Verify numerical equivalence between MATLAB code and generated code. • Improve efficiency of generated code. • Create and maintain code replacement libraries.	“Code Generation from MATLAB Code”

Customize Code Generation Environment

If you are responsible for customizing the code generation environment for a deployment project or for code deployment across an organization, consider these options.

Customization Goal	Description	More Information
Data representation	Customize how the code generator stores data, including matrices and arrays. Control the naming and placement of data types in generated code.	“Data Representation in Generated Code”
Code interface configurations	Create code interface configurations for standardization and software architecture conformance.	“Code Interface Definitions”
Code replacement	Create and maintain code replacement libraries that control how and when the code generator replaces code that it generates for functions and operators.	“Code Replacement Customization”
Model configuration settings	Establish standard model configuration parameter settings that define how the code generator produces and compiles code from models.	“Model Configuration Set Customization”
Custom blocks	Define code generation behavior for custom blocks by developing S-functions.	“Code Generation for Custom Blocks”
Target Language Compiler	Use the Target Language Compiler to customize generated code, inline code generated for S-function blocks, and generate additional or different types of files.	“Target Language Compiler”
Target platform devices	For a target platform device, select a system target file that specifies required target platform environment and code features, use the `target` package to extend the range of hardware that the code generator supports, and configure the code generator to use code replacement libraries.	“Target Platform Device Customization”
Code compilation	Customize the compilation stage of the build process by using a toolchain or template makefile approach.	“Code Compilation Customization”

Customization Goal	Description	More Information
Custom software for target hardware	Create software that supports the deployment of MATLAB and Simulink designs to hardware platforms. Register custom toolchains for the build process, customize XCP server software for target hardware, and create a target hardware application that checks the validity of code generation assumptions. Use the target package to extend the range of supported hardware and provide connectivity between Simulink and your target hardware for external mode and processor-in-the-loop (PIL) simulations.	“Custom Software for Target Hardware”

Embedded Coder Supported Hardware

Get Support Package Now

Embedded Coder generates ANSI/ISO C and C++ code that can be compiled and executed on a processor by manually integrating the generated code with the RTOS, I/O devices, and build tools for your processor.

Embedded Coder provides support packages that help to automate integration, execution, and verification of generated code for the processors and devices in this table.

Support Package	Vendor	Earliest Release Available	Last Release Available
Analog Devices DSPs	Analog Devices®	R2013a	R2015b
ARM Cortex-A Processors	ARM®	R2014a	Current
ARM Cortex-M Processors	ARM	R2013b	Current
ARM Cortex-R Processors	ARM	R2016b	Current
AUTOSAR Standard	AUTOSAR (AUTomotive Open System ARchitecture) development partnership	R2014b	R2018b
BeagleBone Black Hardware	BeagleBoard	R2014b	Current
Green Hills MULTI	Green Hills® Software	R2012b	R2014a
Intel SoC Devices	Intel®	R2014b	Current
PX4 Autopilots	Dronecode	R2018b	R2020a **Note:** Starting in R2020b, see Supported Hardware (UAV Toolbox)
STMicroelectronics STM32 Processors	STMicroelectronics®	R2013b	Current
Texas Instruments C2000 Processors	Texas Instruments®	R2013b	Current

Support Package	Vendor	Earliest Release Available	Last Release Available
Texas Instruments C2000 F28M3x Concerto Processors	Texas Instruments	R2014b	R2022a **Note:** Starting in release R2022b, see Texas Instruments C2000 Processors
Texas Instruments C6000 DSPs	Texas Instruments	R2014a	R2016a
Wind River VxWorks RTOS	Wind River®	R2013b	R2017a
Xilinx Zynq Platform	Xilinx®	R2013a	Current

For a complete list of hardware support packages, see Hardware Support.

See Also

Related Examples

- “Software Component Modeling”

Model Architecture and Design

Design Preparation

- “Divide Software Problem” on page 2-2
- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3

Divide Software Problem

A step in project assessment process is to break down your software problem into smaller parts. You can represent each part as a separate application or component.

Smaller components:

- Facilitate separation of concerns.
- Are easier to work with.
- Enable parallel development.
- Promote reuse.

When assessing a component for reuse, consider whether the component is:

- Reusable as is.
- Reusable with slight variations.
- Expected to be volatile.

If you have access to System Composer™, you can use that software to graphically represent software components as you decompose the software problem and design a software architecture. For more information, see “Author Architecture Models” (System Composer).

As part of the decomposition process, identify data and execution dependencies, such as ports, function-call interfaces, and parameters that require tuning and signal data that must be measured during program execution.

See Also

Related Examples

- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3

Map Embedded System Architecture to Simulink Modeling Environment

Map elements of your system architecture to elements of the Simulink modeling environment. Consider these aspects of the model design.

“Modeling Algorithms” on page 2-3	Given initial state and input, a set of tasks or instructions that efficiently produce the results that you want.
“Modeling Interfaces” on page 2-4	Mechanisms that enable algorithm components to communicate and exchange information across component boundaries.
“Modeling Systems” on page 2-8	Collection of algorithm components that achieve a higher-level, domain-specific goal or result. Components often share resources.
“Modeling Target Platform Environments” on page 2-9	Framework that handles scheduling of system algorithm resources and execution.

The following sections map questions to consider regarding an embedded system architecture to corresponding modeling capabilities and links to related information. Use the information as a guide for mapping your architecture details to the Simulink modeling environment. Designing a model architecture with your specific embedded system architecture in mind can help you avoid rework and future conversion and maintenance costs.

Modeling Algorithms

Architecture Considerations	Modeling Considerations	Related Information
What is the system domain?	Product prerequisites (based on domains of components)	• “Use Products and Blocks Supported for Code Generation” • “Simulink Control Design” • “Signal Processing” • “Signal Generation, Manipulation, and Analysis” (DSP System Toolbox)
Does the system involve physical domains, such as mechanical, electrical, or hydraulic domains?	Physical systems	• “Physical Modeling” • “Basic Principles of Modeling Physical Networks” (Simscape) • “Essential Physical Modeling Techniques” (Simscape)
Are you deploying an application or component?	Modeling styles and code interfaces	“Software Deployment Objectives and Deployment Types” on page 1-2

Architecture Considerations	Modeling Considerations	Related Information
What aspects of your algorithm can you represent with blocks provided by MathWorks products? What blocks do you need to create?	Block usage, creation, and customization	• "Use Products and Blocks Supported for Code Generation" • "Code Generation for Custom Blocks" • "Target Language Compiler"
Do you want to import existing external MATLAB, C, or C++ code into a model for inclusion in your generated code?	External code import	"External Code Import"
Does the architecture include state machine components?	Event-driven system	"Model Finite State Machines by Using Stateflow Charts" (Stateflow)
Do you want to apply modeling guidelines that are relevant to code generation intent? Do models and generated code need to comply with industry standards?	Standards and modeling guidelines	"Standards, Guidelines, and Block Usage"
Is there a need to standardize code that the code generator produces from multiple models?	Custom code interface configurations	• "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2 • **Embedded Coder Dictionary**

Modeling Interfaces

Architecture Considerations	Modeling Considerations	Related Information
What type of code interface are you going to use (data or service)?	Code interfaces	"Configure Code Interfaces for Code Generation" on page 3-4
For a data code interface: • What data must you represent in the generated code? • How do you need to represent input and output—data type, dimension, complexity? • Do the algorithms use floating-point or fixed-point arithmetic? • How will the data change?	Data representation	• "Data Representation in Generated Code" • "Define Interfaces of Model Components" • "Modeling Patterns for C Code Constructs" • "Fixed-Point Designer"

Architecture Considerations	Modeling Considerations	Related Information
• Where and how is data pulled into the system and pulled within the system? • Will the deployed code use a target environment sender service? • Will you represent input by using Inport or In Bus Element blocks? • Will the deployed code read data from nonvolatile memory?	Input, service code interface	• “Comparison of Signal Loading Techniques” • “Target Environment Services” on page 1-9 • “Code Interfaces and Code Interface Specification” on page 1-11 • “Service Interfaces” on page 3-21 • **Embedded Coder Dictionary**Embedded Coder Dictionary • **Code Mappings editor**
• Where and how is data pushed within the system and out of the system? • What external triggers are required? • Will the deployed code use a target environment receiver service? • Will you represent output by using Outport or Out Bus Element blocks? • Will the deployed code write data to nonvolatile memory?	Output, service code interface	• “View Data in the Simulation Data Inspector” • “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67 • “Target Environment Services” on page 1-9 • “Code Interfaces and Code Interface Specification” on page 1-11 • “Service Interfaces” on page 3-21 • **Embedded Coder Dictionary**Embedded Coder Dictionary • **Code Mappings editor**
• What callable entry-point functions do you need to define? • What is the prototype for each entry-point function?	Functions and function calls	• “Create Export-Function Model” • “Simulink Functions Overview” • “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114 • “Interactively Configure C++ Interface” on page 44-2 • “Generate Component Source Code for Export to External Code Base” on page 58-40

Architecture Considerations	Modeling Considerations	Related Information
For a data code interface, can you benefit from setting up default code generation configurations for categories of data and functions?	Data code interface	• “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15 • “Code Interfaces and Code Interface Specification” on page 1-11 • **Embedded Coder Dictionary**Embedded Coder Dictionary • **Code Mappings editor**
Do you need to export functions that are invoked by controlling logic that is outside the model?	Function export	• “Export-Function Models Overview” • “Generate Component Source Code for Export to External Code Base” on page 58-40
Will internal functions exchange data?	Service code interface	• “Target Environment Services” on page 1-9 • “Code Interfaces and Code Interface Specification” on page 1-11 • “Service Interfaces” on page 3-21 • **Embedded Coder Dictionary**Embedded Coder Dictionary • **Code Mappings editor**
Do you need to access target environment timer data?	Service code interface	• “Target Environment Services” on page 1-9 • “Code Interfaces and Code Interface Specification” on page 1-11 • “Service Interfaces” on page 3-21 • **Embedded Coder Dictionary**Embedded Coder Dictionary • **Code Mappings editor**

Architecture Considerations	Modeling Considerations	Related Information
Does the system monitor signals or log data (for example, for calibration)?	C API and ASAP2 data exchange interfaces	• "Configure Parameters for C Code Generation" on page 33-61 • "Configure Signal Data for C Code Generation" on page 33-50 • "Target Environment Services" on page 1-9 • "Service Interfaces" on page 3-21 • "External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling" on page 60-2 • "Exchange Data Between Generated and External Code Using C API" • "Export ASAP2 File for Data Measurement and Calibration"
Do you need to replace code generated for functions or operators, for example, to optimize the code for specific hardware?	Code replacement	• "What Is Code Replacement?" • "What Is Code Replacement Customization?" on page 72-3
Do you need to control the placement of data or functions in memory?	Memory sections	"Control Data and Function Placement in Memory by Inserting Pragmas" on page 45-2
Is there a requirement for elaboration and future considerations?	Elaboration and future considerations	"Define Interfaces of Model Components"

Modeling Systems

Architecture Considerations	Modeling Considerations	Related Information
• What is the scope of the system? Controller? External environment or plant? Test harness? • How is the system partitioned into algorithm components (chunks of logic)? • Which components can you represent in Simulink? • Can you design components for reuse? What is the motivation for reuse (for example, division of labor or plug-n-play)?	Componentization	• "Define Interfaces of Model Components" • "Component-Based Modeling Guidelines" • "Compare Capabilities of Model Components" • "Create Custom Library" • "Export-Function Models Overview" • "Integrate Basic Algorithms Using MATLAB Function Block" • "Generate Subsystem Code as Separate Function and Files" on page 4-7 • "Generate Code for Model Reference Hierarchy" • "Code Generation of Stateflow Blocks"
• Do aspects of the system require unit testing? • Is a team of people collaborating on the project? • Do you need to protect intellectual property?	Model referencing	• "Model Reference Basics" • "Compare Capabilities of Model Components" • "Generate Code for Model Reference Hierarchy" • "Generate Code from Atomic Subcharts"
Are you modeling a client-server architecture?	Simulink Function and Caller blocks	• "Client-Server Architecture" • "Simulink Functions Overview"
Is relevant legacy or custom code available that you can include in a component model?	Code import	"External Code Import"
Can you apply a reference architecture or reference components?	Model and project templates	• "Create Template from Model" • "Create a New Project Using Templates"
Do you need to export functions that are invoked by controlling logic that is outside a model?	Export-function models	"Export-Function Models Overview"
Is there a need to package the source code for a component as a shared object library to simplify distribution or sharing?	Shared object libraries (dynamic link libraries)	"Package Generated Code as Shared Libraries" on page 69-2

Architecture Considerations	Modeling Considerations	Related Information
Can you reuse functions?	Function reuse	• "Generate Reusable Code from Library Subsystems Shared Across Models" • "Generate Reentrant Code from Top Models" • "Generate Reentrant Code from Subsystems"
• Do components need to share access to global data? • Within the system, do state changes occur? In each case, how does the result get communicated? • Are there identifier (naming) issues to consider?	Shared data	• "Local and Global Data Stores" • "Data Structures in the Generated Code" on page 36-31 • "Storage Classes for Parameters and Signals Used in Model Blocks" • "How Generated Code Stores Internal Signal, State, and Parameter Data" • "Data Stores in Generated Code" • "Customize Generated Identifier Naming Rules" on page 55-10 • "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2 • "Code Interface Configuration"
Do you need to control placement of data or functions in memory?	Memory sections	"Control Data and Function Placement in Memory by Inserting Pragmas" on page 45-2
Are you required to apply the AUTOSAR standard? If yes, what aspects of the architecture involve AUTOSAR?	AUTOSAR	"AUTOSAR Blockset"
Does your system need to meet other standards or guidelines?	Standards and guidelines	"Support for Standards and Guidelines" on page 23-2

Modeling Target Platform Environments

Architecture Considerations	Modeling Considerations	Related Information
• What level of control over target platform interfacing does your application require? • How much of your system can you represent in a model?	Target platform interfacing	• "Execution of Code Generated from a Model" • "Modeling Interfaces" on page 2-4
Is the system decomposed into concurrent components to maximize parallelism? Which components?	Concurrency	"Multicore Processor Targets"

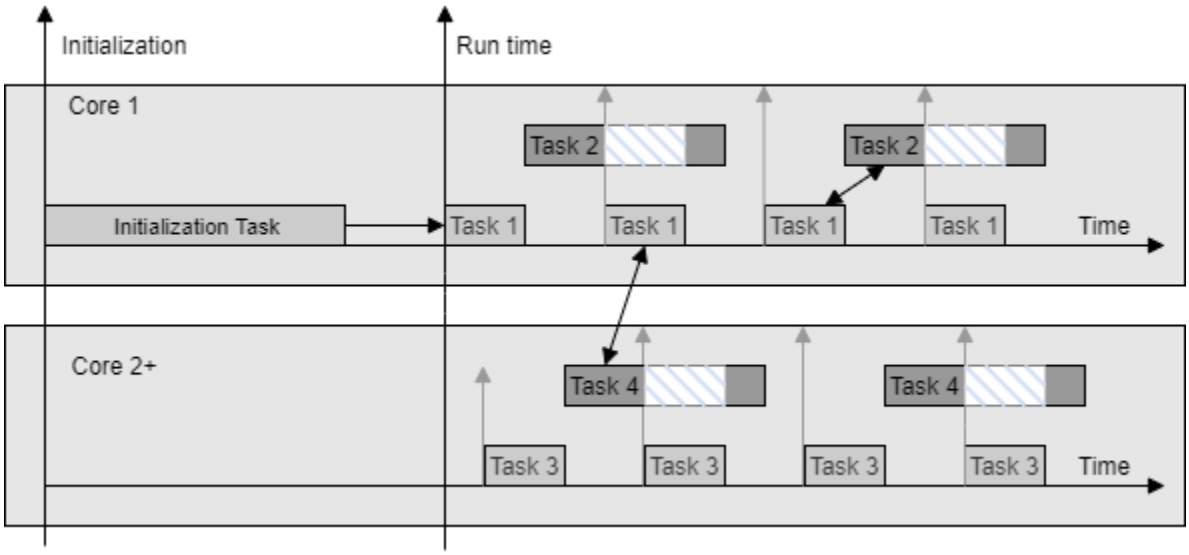
Architecture Considerations	Modeling Considerations	Related Information
• Are components in the system driven by clocks? • What clock rates do system components use? • Do components use a single rate or multiple rates? • What are the priorities of system tasks and functions?	Clocks, clock, rates, and time-based scheduling	• “Define Interfaces of Model Components” • “Service Interfaces” on page 3-21 • “Timer Service Interfaces” on page 3-25 • “Absolute and Elapsed Time Computation” • “Time-Based Scheduling”
• Are components in the system driven by events (interrupts)? • What are the priorities of system tasks and functions?	Event-based scheduling	• “Absolute and Elapsed Time Computation” • “Event-Based Scheduling” • “Model Finite State Machines by Using Stateflow Charts” (Stateflow)
Does the system need to handle initialization, reset, or terminate events during system startup or shutdown?	Initialization, reset, termination	• “Using Initialize, Reinitialize, Reset, and Terminate Functions” “Startup, Reset, and Shutdown Function Interfaces” on page 3-10
• Is the system a single-tasking or multitasking system? • Are components required to execute in real time? • What are the execution order dependencies (sequencing) between components? • What are the time constraints for task and function execution?	Task execution	• “Deployment Objectives and Deployment Types” on page 1-3 • “Execution of Code Generated from a Model” • “Modeling for Single-Tasking Execution” • “Modeling for Multitasking Execution”
• If you know the processing platform, what is it? • Will the system run on a single-core or multicore processor? • Is the system a distributed system? • Is the processing platform hybrid or heterogeneous? • Does the architecture employ symmetric or asymmetric multiprocessing? If asymmetric, how is the platform software partitioned across CPUs?	Processing platforms	• “Deployment Objectives and Deployment Types” on page 1-3 • “Multicore Processor Targets”

Architecture Considerations	Modeling Considerations	Related Information
• Do you want to generate and run a standalone executable that does not require an external real-time kernel or operating system? • Is a real-time operation system (RTOS) required? If yes, what RTOS?	Kernel, operating system	• "Deployment Objectives and Deployment Types" on page 1-3 • "Deploy Applications to Target Hardware" on page 71-2 • "Deploy Generated Component Software to Application Target Platforms" on page 71-34

For multicore scenarios, the code generator assumes that:

- Target hardware includes two or more cores.
- System initialization occurs on one core.
- Services assign tasks to a core statically. Tasks do not switch between cores.
- Tasks can communicate within and across cores during run time.

This figure illustrates these assumptions. The tasks in the figure represent instances of entry-point functions running in a multi-core target platform environment.



Application and Component Interfaces

Model Code Interfaces

To produce code that can interact with target platform services, the code generator must be aware of target platform interface requirements. The first step for doing that is to represent required interfaces in your top model by including specific model elements. For example, you can model an interface for reading data from a target platform receiver service by using a root-level In Bus Element or Inport block.

This table lists typical code interfaces and ways to represent them in a top model.

Type of Code Interface	Model Element or Pattern
Callable rate-based entry point	Block in a rate-based model that is configured for single-tasking or multitasking execution or that is set up as a partition in the Simulink Schedule Editor
Callable exported-function entry point	Exported-function model that defines functions by using Function-Call subsystem, function-call Model, Simulink Function, Message Triggered Subsystem, or S-Function blocks
Callable initialize function entry point	Initialize Function block
Callable terminate function entry point	Terminate Function block
Call to target platform receiver service	In Bus Element or Inport block
Call to target platform sender service	Out Bus Element or Outport block
Call to target platform data transfer service	Signal line that connects callable functions that exchange data
Access target platform nonvolatile memory	Initialize Function and Terminate Function blocks or Simulink Function and Function Caller blocks
Reuse memory between a state variable and root-level output variable	Function loopback pattern that models state variable as a signal or Delay block that models state variable
Call to target platform timer service to get an absolute or elapsed time value	Scoped Simulink Function block (to represent service) and library block that includes a Function Caller block (to call the service) or, for a model configured with a service interface (requires), Discrete Time Integrator or Weighted Sample Time block.
Message communication	Send block directly connected to a root-level Outport block and Receive block directly connected to a root-level Inport block
Client-server communication	Simulink Function and Function Caller blocks
Tunable parameters and parameter arguments	Parameter objects stored in the model workspace
Data for measurement	Signals (output port of a block), states (states on a block), and Datastore Memory blocks
Custom service interface	Custom blocks

These topics provide more information for the different types of interfaces:

- “Periodic and Aperiodic Function Interfaces” on page 3-6
- “Startup, Reset, and Shutdown Function Interfaces”
- “Data Interfaces” on page 3-19
- “Service Interfaces” on page 3-21
- “Message Communication Interfaces” on page 3-33
- “Client-Server Communication Interfaces” on page 3-65
- “Device Driver I/O Interfaces” on page 3-75

Modeling guidelines can be helpful depending on your deployment objective and how you configure code interfaces for the code generator. In some cases, you might need to create custom representations that involve creating custom blocks.

For information on modeling guidelines, see “Modeling Guidelines and Block Usage”“Modeling Guidelines”, “Modeling Guidelines for Generated Code” on page 23-5, and Component Deployment Guidelines for Embedded Coder Support Package.

For information on creating custom blocks, see “Code Generation for Custom Blocks”.

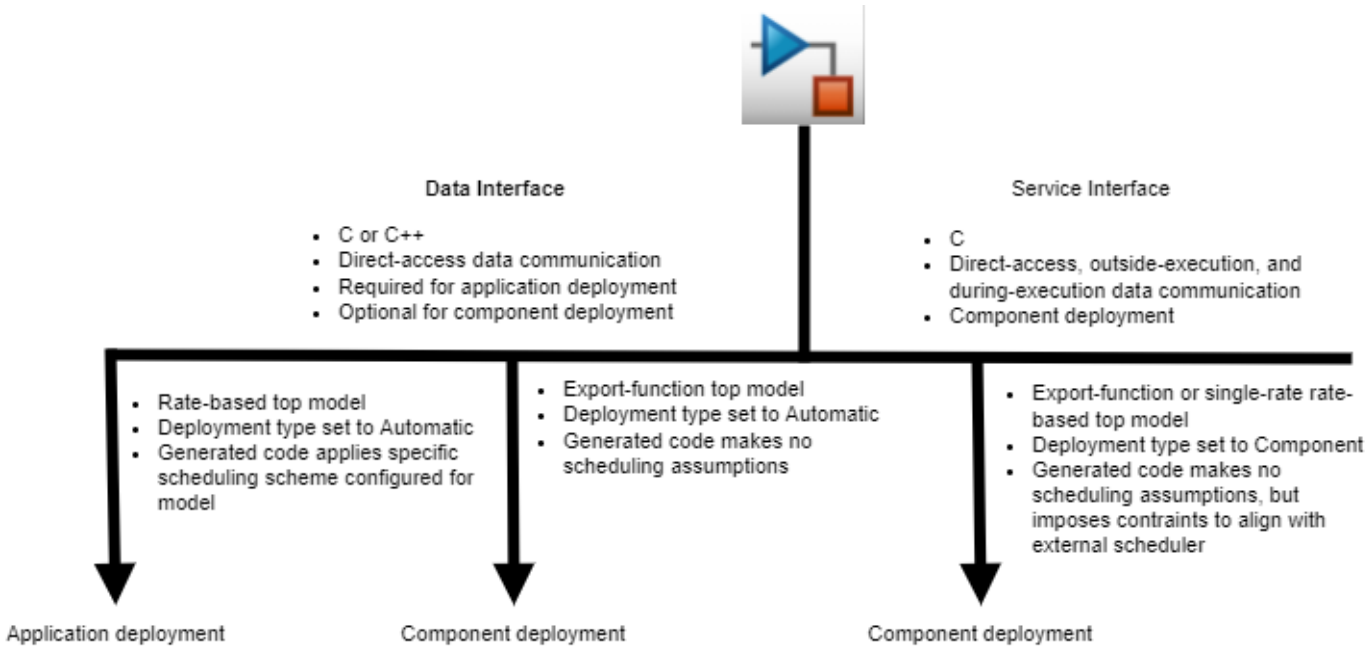
Configure Code Interfaces for Code Generation

In addition to representing code interfaces as elements in your model design, for some elements, you must provide the code generator with mappings of those elements to code interfaces. The code generator uses the mappings to produce code that can interact with target platform services. You configure a model with a data or service interface, depending on your project requirements, deployment objective, and the code generation product that you are using. Service interfaces require Embedded Coder.

As the following figure shows, you must configure a model to use a data code interface for application deployment or if you are using Embedded Coder and generating C++ class code. Code generated for application deployment is optimized to a specific scheduler (single-tasking, rate-monotonic, or concurrent) based on your model configuration.

For component deployment, a service code interface offers benefits such as:

- Customized interface code that includes built-in safeguards for maintaining data coherence for outside-execution and during-execution data communication
- Customized code for data transfers between functions outside of (before and after) function execution
- Support for accessing time values in tasks
- Flexible scheduling
- Mapping of model elements to complete interface abstractions rather than to storage classes and specific property settings



For more information, see:

- “Software Deployment Objectives and Deployment Types” on page 1-2
- “Target Environment Services” on page 1-9

- “Rate-Based Top Models” on page 3-6
- “Export-Function Top Models” on page 3-7
- “Data Communication Methods” on page 1-14
- “Data Interfaces” on page 3-19
- “Service Interfaces” on page 3-21

Periodic and Aperiodic Function Interfaces

Target environment software initiates execution of algorithm code generated for a top model (application or component) by calling generated execution entry-point functions. The execution entry-point functions that implement the top model behavior can be periodic or aperiodic. The entry-point function code that the code generator produces for a top model depends on the modeling styles and periodic rates represented in the model. You can represent periodic and aperiodic entry-point functions in a model by using a combination of two modeling styles: rate-based and, if you are using Embedded Coder, export-function.

This table lists conditions that might prompt you to choose each modeling style.

Rate-Based Modeling Style Conditions	Export-Function Modeling Style Conditions
• Assumes rate monotonic scheduling. • Code generator produces a unique entry-point function for each rate represented in the model. • Sample time is design data. It is not an implementation consideration. • Can generate reusable code from multiple instances of a model.	• You are using Embedded Coder. • A function represents the entry-point function for a server. • A function exchanges data with multiple other functions. • A function represents a periodic entry-point function that has a rate that is not a multiple of the fastest rate used by the top model. • A function represents a periodic entry-point function that other events also trigger. • Multiple functions represent periodic entry-point functions that are triggered at the same period.

If you want a top model to support multiple instantiation, you cannot model periodic functions as function-call subsystems. Use rate-based modeling.

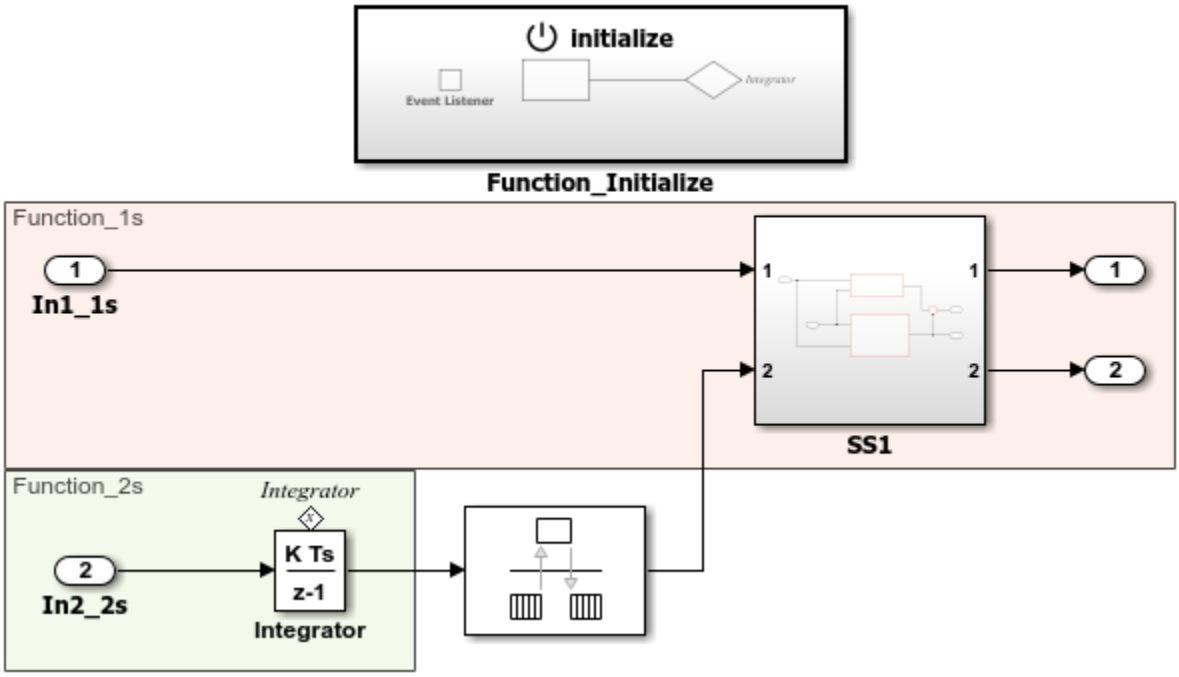
For more information about modeling execution behavior, see “Rate-Based Top Models” on page 3-6 and “Export-Function Top Models” on page 3-7. For guidelines on modeling execution behavior for component models, see “cgsl_0401: Modeling styles for component deployment”.

For options to configure the function prototype for generated periodic and aperiodic execution entry-point functions, see “Periodic and Aperiodic Function Interface Configuration” on page 3-9.

Rate-Based Top Models

A rate-based top model represents periodic entry-point functions by using unique periodic rates.

This example model shows a rate-based implementation of a component top model. The model implements periodic entry-point functions by using the rates 1 second and 2 seconds. An Initialize Function block initializes the component.



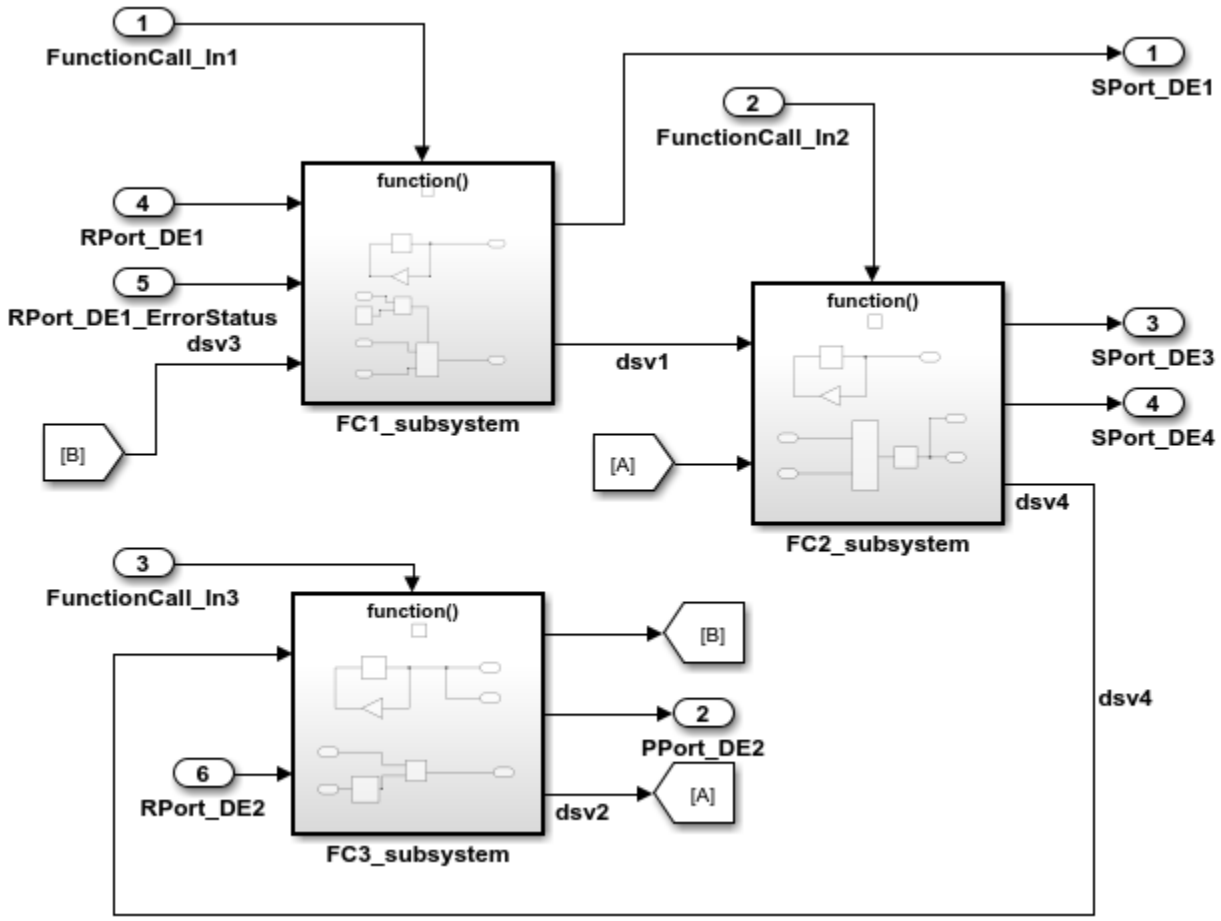
When you generate code for a rate-based top model, the C code contains rate-grouped step entry-point functions, one function for each discrete rate in the model. The target environment function scheduler calls the generated periodic step functions.

You can design a rate-based top model to represent periodic and aperiodic entry-point functions. For example, in the JMAAB-B type beta architecture, an asynchronous trigger entry-point function interacts with periodic rate-based entry-point functions.

Export-Function Top Models

If you are using Embedded Coder, for export-function models, the code generator produces code for independent functions that execute at a periodic rate or aperiodically in response to events. You can integrate the independent function code with other target environment software, including the function scheduler and other platform and custom services.

This example top model shows an export-function implementation of a component. The model uses three Function-Call Subsystem blocks, FC1_subsystem, FC1_subsystem, and FC3_subsystem, which have periodic rates to implement three periodic entry-point functions. Signal lines that connect the Function-Call Subsystem blocks, `dsv1`, `dsv2`, `dsv3`, and `dsv4`, represent data transfers.



When you generate code for an export-function implementation, the code includes a callable model entry-point function for each function represented in the top model. Unless you define and configure the interface otherwise, the code generator uses the name of the function-call subsystem input port as the name of the entry-point function. You can override the default naming by defining a function customization template in an Embedded Coder Dictionary and configuring the function name in the model code mappings.

You can control how the code generator packages the code for an export-function implementation. For example, by using subsystem block parameter **Function packaging**, you can specify whether the code generator inlines the function code for a subsystem or produces separate function code that passes I/O as global data. When inlined, the code generator uses the name of the function-call subsystem input port to name the entry-point function. When producing separate function code, the code generator names the function based on the settings of other parameters, such as **File name options**.

For more information, see “Generate Component Source Code for Export to External Code Base” on page 58-40, “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114, and “Interactively Configure C++ Interface” on page 44-2.

Periodic and Aperiodic Function Interface Configuration

The code generator produces several types of entry-point functions for model functions (see "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114). As an example, for a component model that includes two periodic function-call subsystems and an aperiodic function-call subsystem, the code generator produces three callable entry-point functions for initiating component execution:

- *model*`_step0` - function for the first periodic rate
- *model*`_step1` - function for the second periodic rate
- *function* - aperiodic function, where *function* is the name of the Inport block for a function-call subsystem or label of the signal line connecting the Inport and subsystem

If you are using Embedded Coder, you can customize the entry-point function prototypes to align with function calls initiated by the target environment function scheduler by applying one of these approaches:

- For the model, in the Code Mappings editor or by using the code mappings programming interface, configure the function name and arguments for each model execution function individually.
- Define a function customization template for execution functions in the Embedded Coder Dictionary (see "Code Interface Definitions"). Then, for the model, in the Code Mappings editor or by using the code mappings programming interface, configure the code generator to use that template for model execution functions.

For more information, see "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114.

See Also

Related Examples

- "Schedule Components to Avoid Data Dependency Issues"
- "Export-Function Models Overview"
- "Create Rate-Based Model"
- "Create an Export-Function Model"
- "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114
- "Configure C++ Class Interface for Rate-Based Models" on page 44-46
- "Configure C++ Class Interface for Export-Function Models" on page 44-50
- "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2

Startup, Reset, and Shutdown Function Interfaces

Applications sometimes require logic that executes during system initialization, reset, and termination sequences. To model startup, reset, and shutdown processing in a model, use the Initialize Function, Reset Function (data interface configurations only), and Terminate Function blocks. You can use the blocks anywhere in a model hierarchy.

The Initialize Function, Reset Function, and Terminate Function blocks can control execution of an application or component in response to initialize, reset, or terminate events. Examples of when to generate code that responds to these events include:

- Starting and stopping an application or component.
- Calculating initial conditions.
- Saving and restoring state from nonvolatile memory.
- Generating reset entry-point functions that respond to external events.

You can place the blocks at different levels of a model hierarchy. Each nonvirtual subsystem can have its own set of initialize, reset, and terminate functions. In a lower-level model, Simulink aggregates the content of the functions with corresponding instances in the parent model.

The Initialize Function and Terminate Function blocks contain an Event Listener block. To specify the event type of the function—`Initialize`, `Reset`, or `Terminate`—use the **Event type** parameter of the Event Listener block. The function block reads or writes the state of conditions for other blocks. By default, the Initialize Function block initializes block state with the State Writer block. The Terminate Function block saves the block state by using the State Reader block. When the function is triggered by an initialize or terminate event, the function writes the value of the state variable or reads the value from the specified block.

Application and component models can use the Initialize Function, Reset Function (data interface configurations only), and Terminate Function blocks to model complex startup, reset, and shutdown sequences. The subsystems work with various modeling styles. Software-in-the-loop (SIL) simulation of initialize, reset, or terminate functions works with export-function modeling only.

The code generator produces initialize and terminate code differently than reset code. For initialize and terminate code, the code generator includes your initialize and terminate code in the default entry-point functions, *`model`*`_initialize` and *`model`*`_terminate`. The code generator produces reset code only if you model reset behavior.

For example, consider a component model that includes these elements:

- An Initialize Function block
- Two Reset Function blocks that specify event names `tooHighReset` and `tooLowReset`
- A Terminate Function block

For this scenario, the code generator produces four entry-point functions that respond to target platform function calls triggered by events:

- *`model`*`_initialize` - initialization function
- *`model`*`_tooHighReset` - reset function for `tooHighReset` event
- *`model`*`_tooLowReset` - reset function, for `tooLowReset` event
- *`model`*`_terminate` - terminate function

If you are using Embedded Coder, you can customize the entry-point function names to align with function calls initiated by the target environment function scheduler by applying one of these approaches:

- For the model, in the Code Mappings editor or by using the code mappings programming interface, configure the function names for model initialize, reset, and terminate functions individually.
- Define function customization templates in the Embedded Coder Dictionary. Then, for the model, in the Code Mappings editor or by using the code mappings programming interface, configure the code generator to use the templates for the model event-triggered functions.

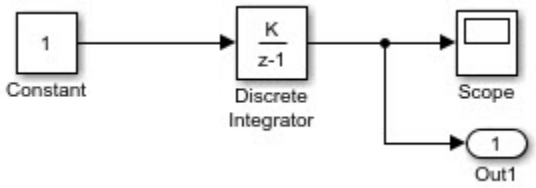
For more information about modeling startup, reset, and shutdown behavior, see “Using Initialize, Reinitialize, Reset, and Terminate Functions”. For guidelines on modeling startup and shutdown behavior for component models, see “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment”.

Generate Code for Initialize and Terminate Events

When you generate code for a component that includes Initialize Function and Terminate Function blocks, the code generator:

- Includes initialize event code with default initialize code in entry-point function `model_initialize`.
- Includes terminate event code with default terminate code in entry-point function `model_terminate`.

Consider the model rtwdemo_irt_base.



For this model, the code generator produces initialize and terminate entry-point functions that other code can interface with.

```
void rtwdemo_irt_base_initialize(void)
void rtwdemo_irt_base_terminate(void)
```

This code appears in the generated file `rtwdemo_irt_base.c`. The initialize function, `rtwdemo_irt_base_initialize`:

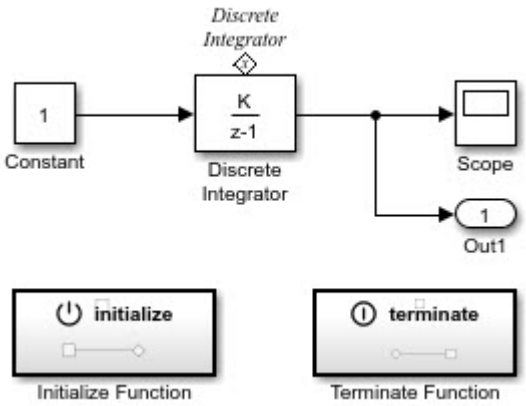
- Initializes an error status.
- Allocates memory for block I/O and state parameters.
- Sets the output value
- Sets the initial condition for the discrete integrator.

The terminate function, ,`rtwdemo_irt_base_terminate`, requires no code.

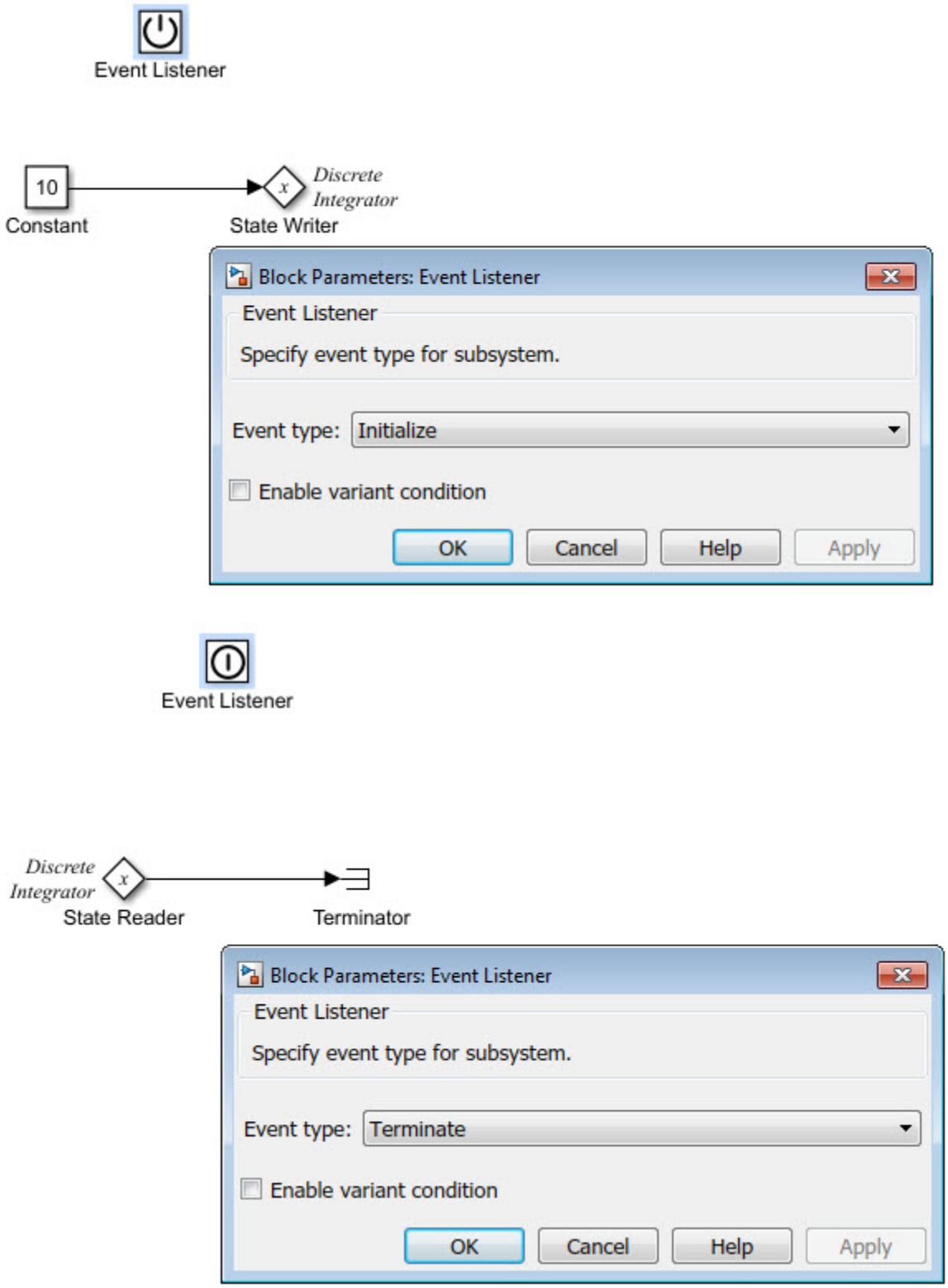
This code assumes that support for nonfinite numbers and MAT-file logging is disabled.

```
void rtwdemo_irt_base_initialize(void)
{
  rtmSetErrorStatus(rtwdemo_irt_base_M, (NULL));

  (void) memset((void *)&rtwdemo_irt_base_DW, 0,
                sizeof(DW_rtwdemo_irt_base_T));

  rtwdemo_irt_base_Y.Out1 = 0.0;

  rtwdemo_irt_base_DW.DiscreteIntegrator_DSTATE = 0.0;
}

void rtwdemo_irt_base_terminate(void)
{
  /* (no terminate code required) */
}
```

Add Initialize Function and Terminate Function blocks to the model (see `rtwdemo_irt_initterm`). The Initialize Function block uses the State Writer block to set the initial condition of a Discrete Integrator block. The Terminate Function block includes a State Reader block, which reads the state of the Discrete Integrator block.



Parameter **Event type** for the Event Listener block for the initialize and terminate functions is set to `Initialize` and `Terminate`, respectively. The initialize function uses the State Writer block to initialize the input value for the Discrete Integrator block to 10. The terminate function uses the State Reader block to read the state of the Discrete Integrator block.



The code generator includes the event code that it produces for the Initialize Function and Terminate Function blocks with standard initialize and terminate code in entry-point functions

`rtwdemo_irt_initterm_initialize` and `rtwdemo_irt_initterm_terminate`. This code assumes that support for nonfinite numbers and MAT-file logging is disabled.

```
void rtwdemo_irt_initterm_initialize(void)
{
  rtmSetErrorStatus(rtwdemo_irt_initterm_M, (NULL));

  (void) memset((void *)&rtwdemo_irt__initterm_DW, 0,
              sizeof(DW_rtwdemo_irt__initterm_T));

  rtwdemo_irt_initterm_Y.Out1 = 0.0;

  rtwdemo_irt_initterm_DW.DiscreteIntegrator_DSTATE = 10.0;
}

void rtwdemo_irt__initterm_terminate(void)
{
  /* (no terminate code required) */
}
```

For guidelines on modeling startup and shutdown behavior for component models, see “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment”.

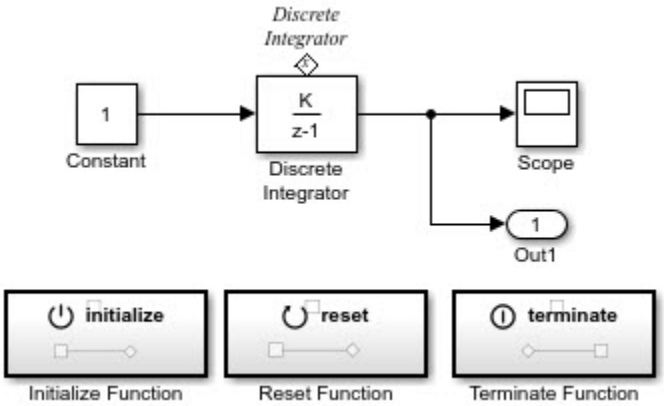
Generate Code for Reset Events

For application models and component models that are configured with a data interface, you can generate code that responds to a reset event by including an Initialize Function or Terminate Function block. Configure the block for a reset by setting the **Event type** parameter of its Event Listener block to `Reset`. Also set the **Event name** parameter. The default name is `reset`.

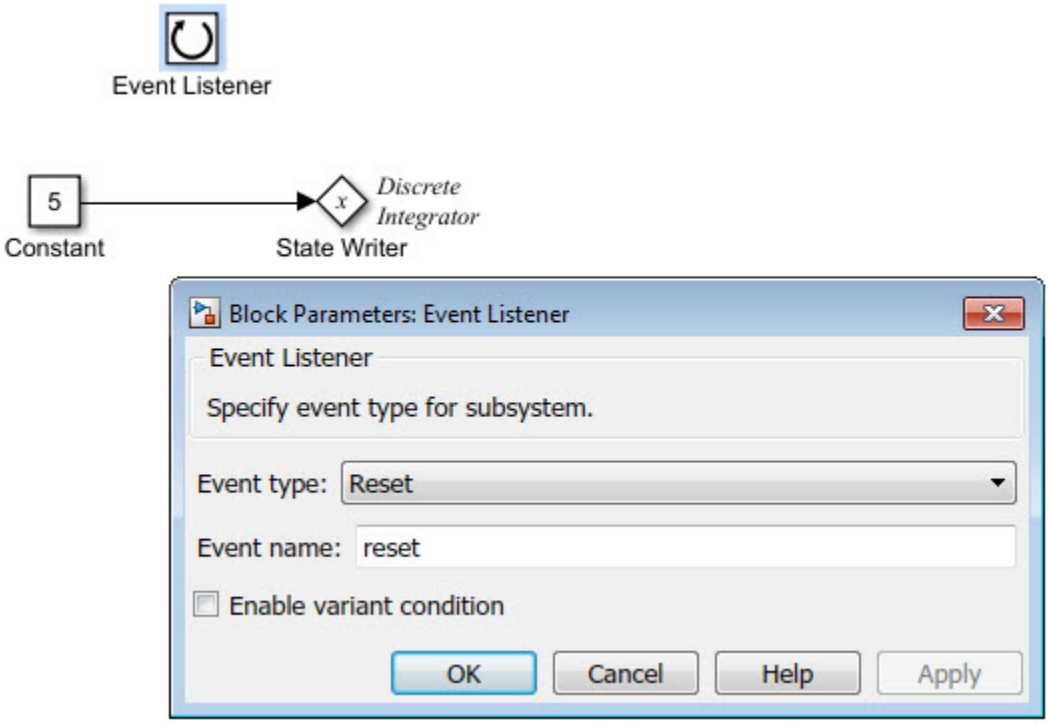
The code generator produces a reset entry-point function *only* if you model reset behavior. If a component contains multiple reset specifications, the code that the code generator produces depends on whether reset functions share an event name. For a given component hierarchy:

- For reset functions with unique event names, the code generator produces a separate entry-point function for each named event. The name of each function is the name of the corresponding event.
- For reset functions that share an event name, the code generator aggregates the reset code into one entry-point function. The code for the reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The name of the function is `model_reset`. For more information, see “Event Names and Code Aggregation” on page 3-16.

Consider the model `rtwdemo_irt_reset`, which includes a Reset Function block derived from an Initialize Function block.



The **Event type** and **Event name** parameters of the Event Listener block are set to `Reset` and `reset`, respectively. The function uses the State Writer block to reset the input value for the Discrete Integrator block to 5.



The code generator produces reset function `rtwdemo_irt_reset_reset`.

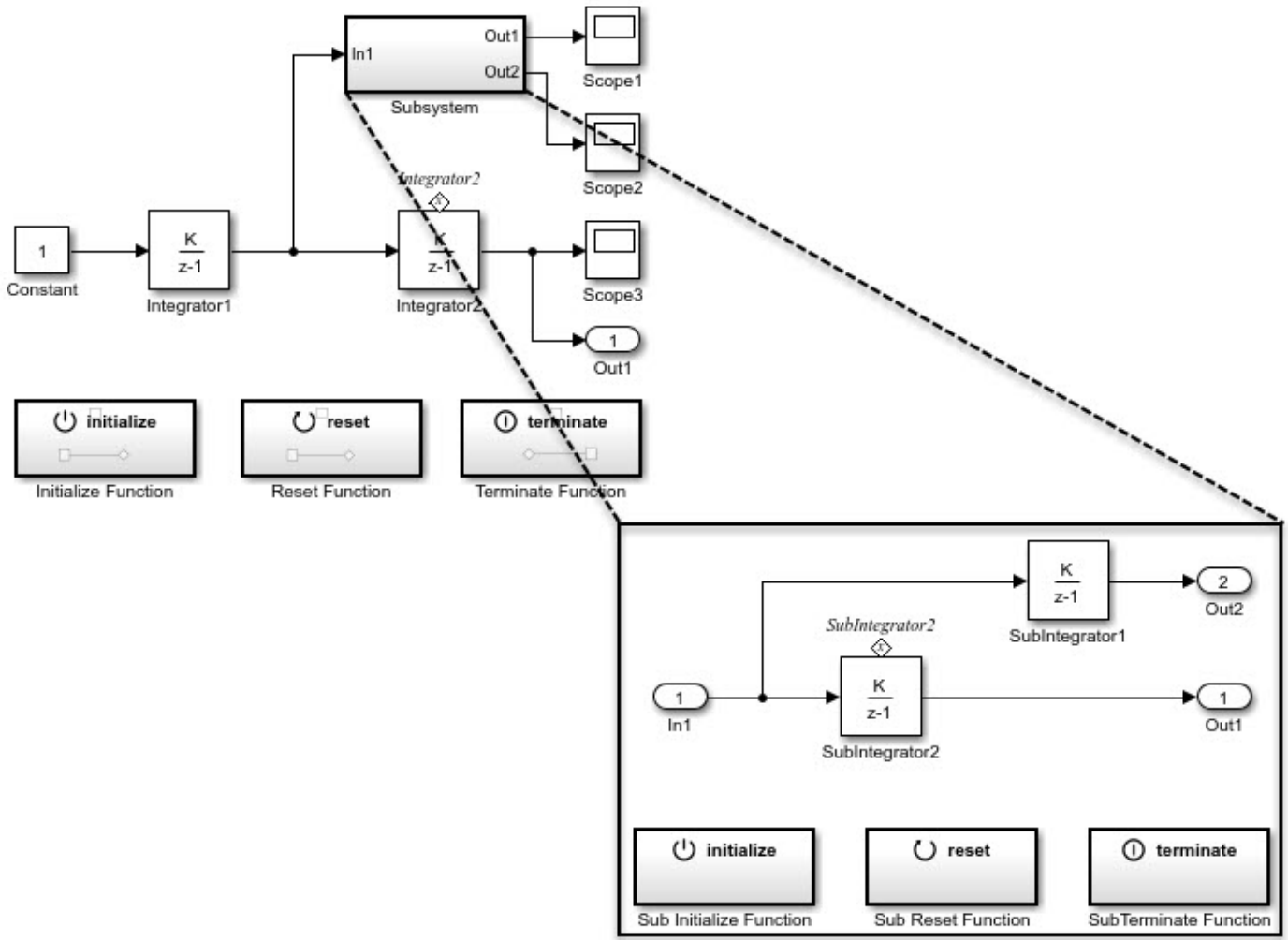
```
void rtwdemo_irt_reset_reset(void)

{
  rtwdemo_irt_reset_DW.DiscreteIntegrator_DSTATE = 5.0;
}
```

Event Names and Code Aggregation

Use the Initialize Function and Terminate Function blocks to define multiple initialize, reset, and terminate functions for a component hierarchy. Define only one initialize function and one terminate function per hierarchy level. You can define multiple reset functions for a hierarchy level. The event names that you configure for the functions at a given level must be unique.

When producing code, the code generator aggregates code for functions that have a given event name across the entire component hierarchy into one entry-point function. The code for reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The code generator uses the event name to name the function.

For example, the model `rtwdemo_irt_shared` includes a subsystem that replicates the initialize, reset, and terminate functions that are in the parent model.



Although the model includes multiple copies of the initialize, reset, and terminate functions, the code generator produces one entry-point function for reset (`rtwdemo_irt_shared_reset`), one for initialize (`rtwdemo_irt_shared_initialize`), and one for terminate (`rtwdemo_irt_shared_terminate`). Within each entry-point function, after listing code for blocks

configured with an initial condition (*model*_P.*block*_IC), the code generator orders code for components, starting with the lowest level of the hierarchy and ending with the root.

```
.
.
.
void rtwdemo_irt_shared_reset(void)
{
  rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 5.0;

  rtwdemo_irt_shared_DW.Integrator2_DSTATE = 5.0;
}
.
.
.
void rtwdemo_irt_shared_initialize(void)
{
  rtmSetErrorStatus(rtwdemo_irt_shared_M, (NULL));

  (void) memset(((void *)&rtwdemo_irt_shared_DW), 0,
               sizeof(DW_rtwdemo_irt_shared_T));

  rtwdemo_irt_shared_Y.Out1 = 0.0;

  rtwdemo_irt_shared_DW.Integrator1_DSTATE = 0.0;

  rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 2.0;

  rtwdemo_irt_shared_DW.Integrator2_DSTATE = 10.0;
.
.
.
void rtwdemo_irt_shared_terminate(void)
{
   /* (no terminate code required) */
   }
```

If you rename the event configured for the subsystem reset function to `reset_02`, the code generator produces two reset entry-point functions, `rtwdemo_irt_shared_reset` and `rtwdemo_irt_shared_reset_02`.

```
void rtwdemo_irt_shared_reset(void)
{
      rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 5.0;
}

void rtwdemo_irt_shared_reset_02(void)
{
   rtwdemo_irt_shared_DW.Integrator2_DSTATE = 5.0;
}
```

Limitation

- You cannot generate code from a harness model—a root model that contains a Model block, which exposes initialize, reset, or terminate function ports.

- You cannot model reset processing in a component model that is configured with a service code interface.

See Also

Initialize Function | Reset Function | Terminate Function

Related Examples

- “Using Initialize, Reinitialize, Reset, and Terminate Functions”
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Initialization of Signal, State, and Parameter Data in the Generated Code” on page 36-187

Data Interfaces

If you are deploying a top model as an application or generating C++ code, configure the model to use a data code interface. Service code interface configurations are recommended for deploying component C code (see “Service Interfaces” on page 3-21). If you are using Embedded Coder and you need to use a data code interface for deployed C component code, see Component Deployment Guidelines for Embedded Coder Support Package.

“Model Code Interfaces” on page 3-2 lists modeling elements that you can use to represent interfaces for interacting with target environment resources. To configure the modeling elements for code generation, in the **Code Mappings editor** or by using the code mappings programming interface, you map the interface elements represented in the model to data code interfaces. What you map the elements to depends on whether you are generating C or C++ code.

C Data Interfaces

When generating C code from a top model for application deployment, you map model data interface elements to storage classes. A storage class specifies the appearance and placement of data elements in the generated code. Storage classes apply direct-access data communication. The list of storage classes available for mapping is determined by the Embedded Coder Dictionary that is associated with the model. The dictionary can define default settings, which can simplify the mapping process when the default mappings apply for your model.

To represent data interface elements in a model and configure them for C code generation:

1. In the model, include data interface elements as recommended in “Model Code Interfaces” on page 3-2.
2. Open the **Embedded Coder** app.
3. If you are using a shared Embedded Coder Dictionary, link your model to that dictionary. Select **Code Interface** > **Set up shared Embedded Coder Dictionary**. In the Set Up Embedded Coder Dictionary dialog box, specify an existing coder dictionary file or create a new one.
4. If you want to set up default mappings for categories of model elements, select **Code Interface** > **Default Code Mappings**. In the **Code Mappings editor**, on the **Data Defaults** and **Function Defaults** tabs, configure default mappings.
5. Select **Code Interface** > **Individual Element Code Mappings**.
6. In the Code Mappings editor, on each element category tab, inspect the mappings and property settings for individual interface elements. If necessary, adjust the settings.
7. Save your interface configuration changes by saving the model.

If you prefer, you can map data elements to storage classes by using the code mappings programming interface (see `coder.mapping.api.CodeMapping`).

For an example, see “Configure Model Data Interface for C Code Generation” on page 33-96 .

For more information, see **Code Mappings editor**, “C Data and Function Interfaces”, and “Code Interface Definitions”.

C++ Data Interfaces

When generating C++ class code for deployment, you:

- Map functions that are represented in a model to method prototypes.
- Map model data interface elements to data visibility and member access function settings.

Data visibility can be private or public. Optionally, you can configure access members for model data as get or set methods as a structure. For inports and outports, you can configure individual access functions and set up data access to be direct (by value) or by pointer. For configurable data types, inline member function definitions are supported.

To represent data interface elements in a model and configure them for C++ code generation:

1. In the model, include data interface elements as recommended in “Model Code Interfaces” on page 3-2.
2. Open the **Embedded Coder** app.
3. Select **Code Interface** > **Code Mappings**.
4. In the **Code Mappings editor**, on the **Data** and **Functions** tabs, inspect the mappings for the interface elements. If necessary, adjust the settings.
5. Save your interface configuration changes by saving the model.

If you prefer, you can configure C++ data interfaces by using the code mappings programming interface (see `coder.mapping.api.CodeMappingCPP`).

For examples, see “Configure C++ Class Interface for Rate-Based Models” on page 44-46 and “Configure C++ Class Interface for Export-Function Models” on page 44-50.

For more information, see **Code Mappings editor** and “C++ Data and Function Interfaces”.

Service Interfaces

When generating C code from a top model for application deployment, you map model data elements to storage classes. A storage class specifies the appearance and placement of data elements in the generated code. Storage classes apply direct-access data communication. The list of storage classes available for mapping is determined by the Embedded Coder Dictionary that is associated with the model. The dictionary can define default settings, which can simplify the mapping process when the default mappings apply for your model.

When generating code from a component model that uses a C service code interface configuration, you map model data elements to service interfaces. A service interface specifies function prototype information and a data communication method (direct-access, outside-execution, or during-execution) for the generated code as required by the target platform services. The code generator applies constraints when producing service interface code so that it is easier to integrate the code for use in a specific target environment. The list of service interfaces available for mapping is determined by interfaces defined in the shared Embedded Coder Dictionary that is linked to the model. If dictionary default service interfaces align with your target platform service requirements, you need not make mapping changes and can skip to generating and reviewing the component interface code.

If you need to set up a component model to use a data code interface configuration, see the Component Deployment Guidelines for Embedded Coder Support Package.

Receiver and Sender Service Interfaces

Within a target environment, a component receives data from and sends data to other components by calling the target platform receiver and sender services. To represent calls to the receiver and sender services in a component model and configure the relevant data elements for code generation:

1. In the component model:
   - Represent a data receive request by using a root-level inport (In Bus Element or Inport block).
   - Represent a data send request by using a root-level outport (Out Bus Element or Outport block).
2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. Select **Code Interface** > **Component Interface**.
4. In the **Code Mappings editor**, on the **Inports** and **Outports** tabs, inspect the mapping for each root-level inport and outport. If you need to use a service other than the default, select an alternative. For example, a service interface might be available for each data communication method.
5. Save your interface configuration changes by saving the model.

If you prefer, you can map the root-level inports and outports to receiver and sender service interfaces by using the code mappings programming interface. See `coder.mapping.api.CodeMapping`.

A best practice is to represent communication I/O ports at the root-level of an application or component model by using In Bus Element and Out Bus Element blocks.

- As a named signal interface, these blocks reduce wiring in the model and avoid issues with port reordering.

- As a structured data type, these blocks enable use of arrays of buses.

To represent a bus as a signal interface, use an In Bus Element or Out Bus Element block that has the **Bus virtuality** attribute set to `virtual`. Virtual buses provide a way to group signals so that you can access them as a whole or select specific signals. A virtual bus consisting of one signal is equivalent to using an Inport or Outport block.

To enforce the composition of a bus interface, use a bus object. A bus object specifies the architectural properties of a bus, such as element names, hierarchy, order, and data types. A bus object is analogous to a structure definition in C. The object defines members of the bus but does not create the bus.

To reduce the number of function arguments that are passed to a reference model in the generated code, choose one of these options for configuring an In Bus Element:

- Set the block **Bus virtuality** attribute to `nonvirtual` and, in the model code mappings, for the inport, set the storage class.
- Set the block **Bus virtuality** attribute to `virtual` or `inherited` and, in the model code mappings, for the inport, set the storage class to `Default`.

If you have a requirement to represent root-level I/O in the generated code as a structured data type, model the inport or outport as a nonvirtual bus. The data type of a nonvirtual bus must be a `Simulink.Bus` object that is defined in the model base workspace or data dictionary.

Note For an export-function model, you cannot configure a root-level inport or outport to use a service interface if the port is connected to a signal that is configured with a storage class.

For an example, see “Generate Sender and Receiver C Interface Code for Component Deployment” on page 34-2.

For more information, see:

- “Configure Sender and Receiver Service Interfaces for Model Inports and Outports” on page 35-2
- In Bus Element, Out Bus Element, Inport, and Outport
- “cgsl_0402: Signal interfaces for component deployment”
- “cgsl_0405: Data receive for component deployment”
- “cgsl_0406: Data send for component deployment”
- “cgsl_0408: Partial data send for component deployment”
- “Data Communication Methods” on page 1-14
- “Composite Interface Guidelines”
- “Group Signals or Messages into Virtual Buses”
- “Create Nonvirtual Buses”

Data Transfer Service Interfaces

Within a target environment, callable functions that are within a component exchange data by calling the target platform data transfer service. To represent calls to a platform data transfer service and configure the relevant data elements for code generation:

1. In the component model, represent a data transfer by using a signal line that connects the communicating functions.
2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. Select **Code Interface** > **Component Interface**.
4. In the **Code Mappings editor**, on the **Data Transfers** tab, inspect the mapping for data transfer signals. If you need to use a service other than the default, select an alternative. For example, a service interface might be available for each data communication method.
5. Save your interface configuration changes by saving the model.

If you prefer, you can map the signal lines that represent data transfers to data transfer service interfaces by using the code mappings programming interface. See `coder.mapping.api.CodeMapping`.

For an example, see “Generate C Data Transfer Service Interface Code for Component Deployment” on page 34-6.

For more information, see:

- “Configure Data Transfer Service Interfaces for Data Transfer Signals” on page 35-5
- “Signal Basics”
- “Data Communication Methods” on page 1-14
- “cgsl_0409: Data transfer for component deployment”
- “Configure Signal Data for C Code Generation” on page 33-50
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75

Nonvolatile Memory Interfaces

You can choose between two approaches to model interfacing for target environment nonvolatile memory. You can use I/O ports with initialize and terminate functions or a client-server interface.

Requirement	I/O Ports with Initialize and Terminate Functions	Client-Server Interface
Robust handling of memory access by functions that execute concurrently	X	
Multiple instances of a component	X	
Use target platform service		X

I/O Ports with Initialize and Terminate Functions

For robust handling of memory access by functions that execute concurrently or if you are using multiple instances of a component, model and configure your nonvolatile memory interface as follows:

1. In the component model:
   - Represent a direct memory read with an inport (Inport or In Bus Element block) connected to an Initialize Function block.

- Represent a direct memory write with an outport (Outport or Out Bus Element block) connected to a Terminate Function block.

2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. Select **Code Interface** > **Component Interface**.
4. In the **Code Mappings editor**, configure the code interfaces for the Inport and Outport blocks that are connected to the Initialize Function and Terminate Function blocks. Map the Inport and Outport blocks to a service interface that uses the direct-access data communication method.
5. In the Code Mappings editor, on the **Functions** tab, inspect the mapping for each initialize and terminate entry-point function. If necessary, adjust the default function prototypes. For more information, see "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114.
6. Save your interface configuration changes by saving the model.

For more information, see:

- In Bus Element, Out Bus Element, Inport, Outport, Initialize Function, and Terminate Function
- "Data Communication Methods" on page 1-14
- "cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks"
- "Configure Root-Level Inport Blocks for C Code Generation" on page 33-30
- "Configure Root-Level Outport Blocks for C Code Generation" on page 33-41
- "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
- "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114

Client-Server Interface

Use a centralized client-server interface to access the nonvolatile memory of a target environment when the target environment provides a platform service. To use the client-server approach:

1. In the model:
   - Represent the target environment service that provides access to nonvolatile memory by using a Simulink Function block.
   - Access the service by using a Function Caller block.
2. If you have not already done so, link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the Embedded Coder app.
3. Configure the function prototype generated for the blocks to align with target platform requirements by using the **Code Mappings editor**. See "Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks" on page 42-2.
4. Save your interface configuration changes by saving the model.

For more information, see:

- Simulink Function and Function Caller
- "Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks" on page 42-2

Timer Service Interfaces

For component model functions, you can generate code that accesses the function clock tick used by the target environment. A function clock tick value is the number of time ticks from the start of program execution to the present time for a function. Generated code for a function can access the target platform function clock tick value by calling a function provided by the platform timer service.

You can configure and generate a call to a timer service function for:

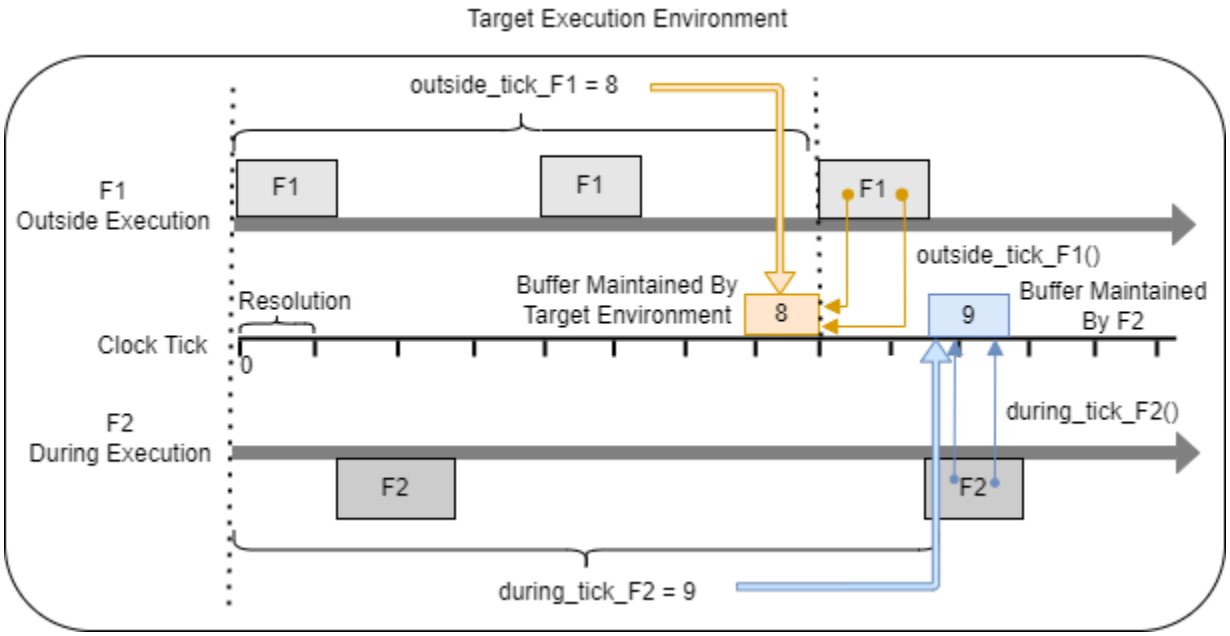
- Aperiodic entry-point functions generated from models that use blocks that rely on an elapsed time value (Discrete Time Integrator and Weighted Sample Time blocks)
- Periodic entry-point functions generated from models that use blocks that rely on elapsed time in an aperiodic context. An example of an aperiodic context is when the **Sample time type** parameter of the Trigger Port block of the model or subsystem that includes the time-based block is set to `triggered`.
- Periodic entry-point functions generated from models that use blocks that rely on absolute time values, such as Sine Wave and Pulse Generator blocks

You represent requests for the function clock tick in a model implicitly when you include blocks that rely on elapsed or absolute time values. To configure the code generator to produce code that calls a timer service function, use the Code Mappings editor or code mappings programming interface to map functions that use time-based blocks to a timer service interface defined in the Embedded Coder Dictionary that is linked to the model.

The code generator assumes that:

- The target platform timer service maintains integrity of the function clock tick value, including during concurrent access, for single- and multiprocess target environments and supports the outside-execution and during-execution data communication methods.
- The clock resolution is specified in seconds with model configuration parameter **Clock resolution** (`ClockResolution`). If **Clock resolution** is set to `-1`, the clock resolution is inherited based on scheduling properties for the model style and type of function (periodic or aperiodic). For example, the code generator sets the inherited clock resolution for an aperiodic function to the model fixed-step size (fundamental sample time). For periodic functions, the inherited clock resolution is the function sample time.

This timing diagram shows how the function clock tick service for a target platform communicates data by using the outside-execution and during-execution data communication methods.



In the diagram, functions `F1` and `F2` represent generated callable functions that are configured to interface with timer service functions. `F1`, the faster of the two functions, is configured to interface with service function `outside_tick_F1`, which uses outside-execution data access. Before `F1` executes, the service function writes clock tick value eight to a buffer that the target environment maintains. The service holds that value constant until execution of `F1` is complete. The diagram shows `F1` accessing the clock tick value twice by calling `outside_tick_F1`. Each function call returns the tick value eight.

`F2` is configured to interface with service function `during_tick_F2`, which uses during-execution data access. Function `during_tick_F2` returns the current clock tick value at the time of the service call. When the model uses during-execution data access, the timer service does not save the value in a locked buffer. The value that the service function returns can change as the `F2` executes. To maintain data coherency, the first time that `F2` calls service function `during_tick_task2`, `F2` creates a buffer to store clock tick value nine and locks it. Subsequent calls to `during_tick_F2` access the tick value from the locked buffer maintained by the F2. The diagram shows a second call to the service function, which returns tick value nine.

To represent calls to a platform data transfer service and configure the relevant data elements for code generation:

1. In the component model, add blocks that rely on time as needed.
2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. Select **Code Interface** > **Component Interface**.
4. In the **Code Mappings editor**, on the **Functions** tab, inspect the timer service mappings

   component model functions. Select the row for a function. Click the pencil icon . In the dialog box that appears, check the setting of the **Timer Service**. If you need to use a service interface other than the default, set the property to the name of the required timer service interface. For example, a service interface might be available for each data communication method.

5 Save your interface configuration changes by saving the model.

If you prefer, you can associate a timer service interface with an entry-point function by using the code mappings programming interface. See `coder.mapping.api.CodeMapping`.

For an example, see “Generate C Timer Service Interface Code for Component Deployment” on page 34-13.

For more information, see:

- “Configure Timer Service Interfaces” on page 35-9
- Discrete-Time Integrator
- Weighted Sample Time
- “Timers”

Parameter and Parameter Argument Tuning Service Interfaces

Some components require interface support for tunable parameters and parameter arguments. When a parameter or parameter argument is tunable, you can change the value that is stored in memory during program execution. For example, you can use a tunable parameter to determine an optimal value by changing the value and monitoring the results or to adapt an algorithm to different execution conditions. An example of a block parameter value that you might want to be tunable is the **Gain** parameter of the Gain block.

To represent a parameter or parameter argument in a component model and configure the parameter to be tunable in generated code:

1. Set up the parameter data for the model by defining the data as parameter object in the model workspace. For parameter argument data, select the **Argument** property.
2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. Select **Code Interface** > **Component Interface**.
4. In the **Code Mappings editor**, on the **Parameters** tab, inspect the parameter mappings. If you need to use a service other than the default, change the service setting. If you do not need to tune a parameter, select `Not tunable`.
5. Optionally, for each parameter and parameter argument that you want to tune, you can specify an identifier that informs the code generator how to represent the variable in the generated code. In the row for a parameter or parameter argument, select the pencil icon . In the dialog box that appears, specify a value for the **Identifier** property.
6. Save your interface configuration changes by saving the model.

If you prefer, you can map parameters and parameter arguments to parameter tuning and parameter argument tuning service interfaces, by using the code mappings programming interface. See `coder.mapping.api.CodeMapping`.

After you configure parameter and parameter argument interfaces, you can:

- Set up and run external mode simulations that use a communication channel between Simulink on your development computer (host) and your generated executable program that runs on the target platform by using the **Run on Custom Hardware** app.

- Generate and customize an ASAP2 file and a Calibration Data Format (CDF) file by using the **Generate Calibration Files** tool.
- Use the C application programming interface (API) or Target Language Compiler (TLC) programming interface to tune parameters and parameter arguments.

For an example, see “Generate C Parameter Tuning Service Interface Code for Component Deployment” on page 34-20.

For more information, see:

- “Manage Design Data”
- “Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments” on page 35-13
- “Configure Parameters for C Code Generation” on page 33-61
- “Exchange Data Between Generated and External Code Using C API” on page 62-2
- “Parameter Functions” on page 22-23“External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2
- “Generate ASAP2 and CDF Calibration Files” on page 64-2

Measurement Service Interfaces

Some components require interface support for viewing and analyzing model data (signal, state, and datastore data) during program execution. For example, you can use the Simulation Data Inspector to produce plots and maps of the data for comparing run results that show model behavior.

To represent model data in a component model and configure the data elements for measurement in generated code:

1. Set up the data elements in the model design.

Data Element	Representation in Model
Signal	Output port of block
State	State on block
Datastore	Datastore Memory block

2. Link your model to a shared Embedded Coder Dictionary that defines a service code interface configuration and open the **Embedded Coder** app.
3. In the **Embedded Coder** app, open the **Code Mappings editor**. In the **C Code** tab, select **Code Interface > Individual Code Mappings**.
4. For signals, add the signal lines that you want to measure to the model code mappings.
   a. In the component model, select a signal.
   b. Pause on the ellipsis that appears above or below the signal line to open the action bar. Click the **Add Signal** button.

      In the Code Mappings editor, on the **Signals/States** tab, the Signals node expands and lists the signal that you added.
5. If you need to use a service interface other than the dictionary default for measurement data:

- On the **Signals/States** tab, select an alternative measurement service for a selected signal or state. For signals and states that you do not want to measure, select `Not measurable`.
- On the **Data Stores** tab, select an alternative measurement service for a selected data store. For data stores that you do not want to measure, select `Not measurable`.

6 For each signal and state, select the pencil icon . In the dialog box that appears, specify a value for the **Identifier** property. The identifier informs the code generator how to represent the variable in the generated code.

7 Save your interface configuration changes by saving the model.

If you prefer, you can map signals, states, and datastores to measurement service interfaces, by using the code mappings programming interface. See `coder.mapping.api.CodeMapping`.

After you configure measurement interfaces, you can:

- Set up and run external mode simulations that use a communication channel between Simulink on your development computer (host) and your generated executable program that runs on the target platform by using the **Run on Custom Hardware** app.
- Generate and customize an ASAP2 file and a Calibration Data Format (CDF) file by using the **Generate Calibration Files** tool.
- Use the C application programming interface (API) or Target Language Compiler (TLC) programming interface to visualize and analyze signal, state, and data store data without using external mode.

For an example, see “Generate C Measurement Service Interface Code for Component Deployment” on page 34-24.

For more information, see:

- “Manage Design Data”
- “Configure Measurement Service Interfaces for Signals, States, and Data Stores” on page 35-16
- “Configure Signal Data for C Code Generation” on page 33-50
- “Configure Block States for C Code Generation” on page 33-74
- “Exchange Data Between Generated and External Code Using C API” on page 62-2
- “Input Signal Functions” on page 22-7
- “Output Signal Functions” on page 22-17
- “Block State and Work Vector Functions” on page 22-29
- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2

Service Interface Constraints and Limitations

When designing and configuring a component model that uses a service code interface configuration, consider constraints that the code generator applies and release-specific limitations.

Constraints

Service interface constraints are product features that are not relevant and are not supported when you are deploying software that uses a service interface configuration.

Modeling styles and patterns that are not supported:

- Blocks that use continuous state
- S-function blocks that specify option `SS_OPTION_ASYNCHRONOUS`, `SS_OPTION_ASYNCHRONOUS_INTERRUPT`, or `SS_OPTION_ASYNCHRONOUS_CUSTOM` to change slower task IDs to faster task IDs
- S-function outports that have a mix of data and function-call elements
- Use of state ports
- `Classic` initialization mode for conditionally executed subsystem initialization values
- Multirate global datastores
- Multirate enabled subsystems
- Multirate S-functions

Model configuration constraints:

- Cannot select **Support continuous time** (`SupportContinuousTime`)
- Cannot select **Support non-inlined S-functions** (`SupportNonInlinedSFcns`)
- Cannot select **Single output/update function** (`CombineOutputUpdateFcns`)
- Cannot specify external model for parameter tuning and signal monitoring by using XCP transport layer
- Cannot select **Generate an example main program** (`GenerateSampleERTMain`)
- Cannot select **MAT-file logging** (`MATFileLogging`)
- Cannot select **Invalid input data access in chart initialization** (`SFInvalidInputDataAccessInChartInitDiag`)
- Must select **Remove root level I/O zero initialization** (`ZeroExternalMemoryAtStartup = 'off'`)
- Cannot clear **Combine signal/state structures** (`CombineSignalStateStructs`)
- Cannot select **Classic call interface** (`GRTInterface`)
- Cannot select **Target supports function prototype control** (`ModelStepFunctionPrototypeControlCompliant`)
- Cannot specify **TLC command line options** (`TLCOptions`) `- -aInlineSetEventsForThisBaseRateFcn=TLC_FALSE`, `-aSuppressMultiTaskScheduler=TLC_FALSE`, `-aRateBasedStepFcn=TLCFALSE, -` (wrapper function)

Receiver and sender service interface constraints:

- Export-function models that include a signal that is configured with a storage class and is connected to a root-level inport or outport

Other capabilities that are not supported:

- Function prototype control for Initialize Function, Reset Function, and Terminate Function blocks, component step functions, and private Simulink Function blocks
- Automatic switching between simulation and code generation variants
- Subsystem builds

- Use of external mode for parameter tuning and signal monitoring

R2023a Limitations

In R2023a, these limitations apply when you deploy software that uses a service interface configuration.

Modeling styles and patterns that are not supported for R2023a:

- Multirate, rate-based models
- Rate Transition and Reinitialize Function blocks
- Use of function ports to call Simulink functions in a reference model from another reference model
- Use of State Reader and State Writer blocks connected to outports or used in functions other than initialize, reset, or terminate functions
- Message-based communication involving signal connections between root-level inports and outports and Send and Receive blocks
- Global datastores and local datastores that are configured with a global storage class, such as `ExportedGlobal`
- Simulink Function blocks that have the **Function visibility** parameter set to `global`
- Component models that have model configuration parameter **Code interface packaging** set to `Reusable function`
- Component models with a root-level Inport block meeting the following conditions:
  - The Inport block is mapped to a receiver service interface with the data communication method set to `During Execution`
  - One branch of the Inport block is driving a subsystem whose function interface is not set to **Allow arguments (Match graphical interface)**
- Component models that include an S-Function block that relies on elapsed time.

Model configuration R2023a limitations:

- Cannot set **Language** (`TargetLang`) to `C++` and **Code interface packaging** (`CodeInterfacePackaging`) to `C++ class`
- Must set **Default parameter behavior** (`DefaultParameterBehavior`) to `Inlined`
- Cannot select **Support variable-size signals** (`SupportVariableSizeSignals`)
- Cannot select **Support non-inlined S-functions** (`SupportNoninlinedSFcns`)
- Cannot set **Code interface packaging** (`CodeInterfacePackaging`) to `Reusable function`
- To prevent an overflow condition, set **Application lifespan (days)** (`LifeSpan`) to a value that does not make the clock tick value exceed 32 bits.

R2023a limitations for specific service interfaces:

- Merge blocks connected to root-level inports or outports, where the inports or outports meet one of the following conditions, and the model is configured with a service interface are not supported:
  - Use during-execution data communication.
  - Use outside-execution data communication and the signal has a scalar data type.

- Receiver and sender service interfaces that apply to multiple function-call inports connected to a Mux block that feeds the trigger for a function-call subsystem are not supported.
- Data transfer service interface for a mix of signal types from different exported functions is not supported.
- Timer service interfaces support 32-bit clock tick values.

Message Communication Interfaces

Within a platform environment, code generated from a Simulink model can communicate with platform software or with other code generated from Simulink models by using message-based communication. Generate C++ component code for message-based communication with platform software. Generate C++ or C component code for message-based communication with other models.

Generate C++ Messages to Communicate Data Between Simulink and an Operating System or Middleware

To generate C++ code that supports message communication between Simulink top models and external applications, use the Simulink Messages & Events Library Send and Receive blocks. Generating code from a top model to facilitate messages passed outside of the Simulink environment enables your modeled application to communicate in a distributed system that uses an external message protocol service, commonly referred to as an operating system or middleware (for example, DDS, ROS, SOMEIP, or POSIX messages).

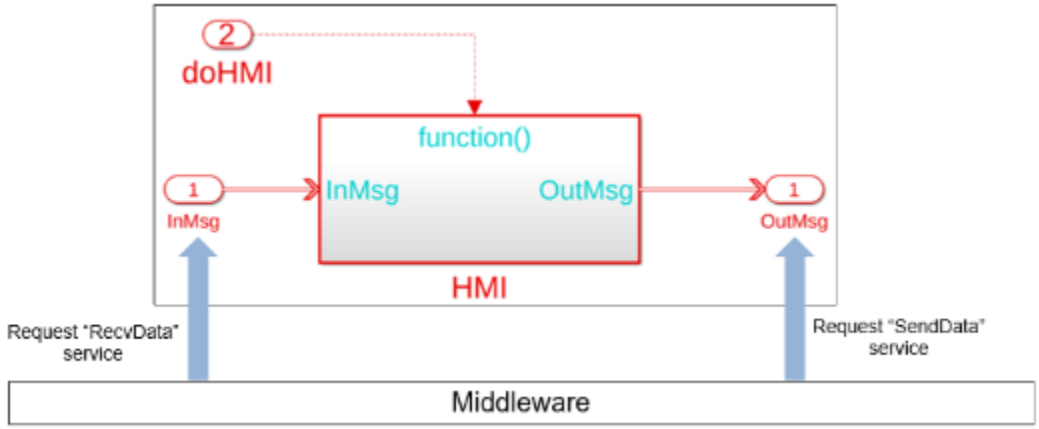
Simulink top models pass messages by:

- Top models contain message blocks to communicate outside the Simulink environment. If a top model contains a Send block directly connected to a root Outport block, the block converts its signals into messages and passes them outside the Simulink environment. If a top model contains a Receive block directly connected to a root Inport block, the block converts the received messages into signals.
- The external message protocol manages the message communication according to its own standard, that is, policies that control capacity, order of delivery, and other quality of service (QoS) behavior.

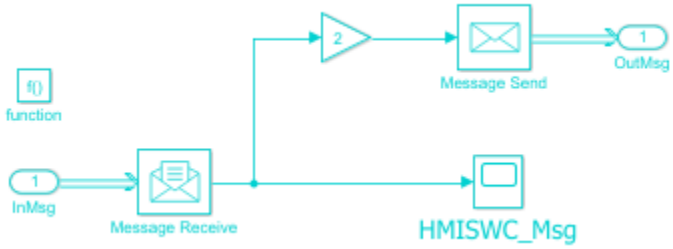
To generate C++ messages to communicate between Simulink top models and an operating system or middleware, prepare your model, generate code, and integrate that code with your selected operating system or middleware.

Prepare Model

To set up your model so that it can pass messages with an operating system or middleware, configure the model as a top model that has at least one message block, either a Send block connected to a root Outport block or a Receive block connected to a root Inport block. The top model can then connect to your selected operating system or middleware through the message blocks:



Within a model, the message ports are connected as shown:



Generate Code

To generate C++ code from a model:

1. In the Apps gallery, click **Embedded Coder**.
2. In the Configuration Parameters dialog box, set these parameters:
   - In the **Code Generation** pane, set the **Language** to `C++`.
   - In the **Interface** pane, set **Code interface packaging** to `C++ class`.
   - In the **Templates** pane, select `Generate an example main program`.
3. Generate code. On the **C++ Code** tab, click **Build**.
4. View the generated code. On the **C++ Code** tab, click **View Code**.

Integrate Code

To integrate the generated C++ code from your model with your selected operating system or middleware use handwritten code to implement send and receive message classes and application code that uses those classes to pass messages. Specifically:

1. If you use your own main, create concrete subclasses from the generated abstract classes shown in `RecvData_<T>.h` and `SendData<T>.h`. If you use the generated example main program, concrete subclasses are provided in the file.
2. Implement the class functions `SendData` and `RecvData` to call into your selected operating system or middleware to send and receive messages.

3. Create an instance of your implemented send and receive message classes (send and receive objects).
4. Create an instance of the model class by using the instances of each message class (send and receive objects) as arguments in the model constructor.
5. Send and receive messages as required by your application. Manage the lifetime of the message queues.The queues must be in a state ready to accept messages prior to the first step of the model.

To integrate the generated code with a platform main program:

1. Open the generated example main program or create your own. If you use the generated example main program, the concrete subclasses `RecvData_real_T` and `SendData_real_T` are in the file. If you create your own main, create concrete subclasses in your application code:

```
class mHMIHandlerRecvData_real_T : public RecvData_real_T{
 public:
  void RecvData(real_T* data, int32_T length, int32_T* status)
  {
    // Add receive data logic here
  }
};

static mHMIHandlerRecvData_real_T InMsgRecvData_arg;
class mHMIHandlerSendData_real_T : public SendData_real_T{
 public:
  void SendData(const real_T* data, int32_T length, int32_T* status)
  {
    // Add send data logic here
  }
};

static mHMIHandlerSendData_real_T OutMsgSendData_arg;
static mHMIHandler mHMI_Obj(InMsgRecvData_arg, OutMsgSendData_arg);

int_T main(int_T argc, const char *argv[])
{
  // Unused arguments
  (void)(argc);
  (void)(argv);

  // Initialize model
  mHMI_Obj.initialize();
  while (rtmGetErrorStatus(mHMI_Obj.getRTM()) == (NULL)) {
    //  Perform application tasks here.
  }

  // Terminate model
  mHMI_Obj.terminate();
  return 0;
}
```

2. To receive messages, manually write the implementation of the generated receive class. Implement the class function `RecvData` to call into your selected operating system or middleware to receive messages.

   An example implementation for POSIX is:

```
class mHMIHandlerRecvData_real_T: public RecvData_real_T {
    public:
    void RecvData(real_T* data, int32_T length, int32_T* status)
    {
        // Use POSIX API mq_receive to receive messages
        unsigned int priority = 1;
        *status = mq_receive(msgQueue, (char *)data, length, &priority);
    }
};
```

Create a receive object.

```
static mHMIHandlerRecvData_real_T InMsgRecvData_arg;
```

3 To send messages, manually write the implementation of the generated send class. Implement the class function `SendData` to call into your selected operating system or middleware to send messages.

An example implementation for POSIX is:

```
class mHMIHandlerSendData_real_T : public SendData_real_T {
    public:
    void SendData(const real_T* data, int32_T length, int32_T* status)
    {
        // Use the POSIX API mq_send to send messages
        unsigned int priority = 1;
        *status = mq_send(msgQueue, (char*)data, length, priority);
    }
};
```

Create a send object.

```
static mHMIHandlerSendData_real_T OutMesgSendData_arg;
```

Create an instance of the model class by using your send and receive objects as arguments in the model constructor.

```
static mHMIHandler mHMI_Obj(InMsgRecvData_arg, OutMsgSendData_arg);
```

4 Send and receive messages as required by your application and maintain the lifetime of the message queue.

An example implementation for POSIX is:

```
int_T main(int_T argc, const char *argv[])
{
    // Unused arguments
    (void)(argc);
    (void)(argv);

    //Initialize model
    mHMI_obj.initialize();

    // Open POSIX queue
     mqd_t msgQueue = mq_open("/PosixMQ_Example", O_RDONLY);
     if (msgQueue == -1)
     {
         printf("mq_open failed\n");
         exit(1);
```

```
        }

       // Send and Receive messages
        while (rtmGetErrorStatus(mHMI_Obj.getRTM()) == (NULL)) {
            //perform application tasks here.
            rt_OneStep();
        }

       // Close POSIX queue
        mq_close(msgQueue);

       // Terminate model
        mHMI_Obj.terminat():
        return 0;
    }
```

For a more complex POSIX integration example, see “Model Message-Based Communication Integrated with POSIX Message Queues” on page 3-47

Considerations and Limitations

- Handwritten code is the only supported integration technique.
- You must select the model configuration parameter **Generate an example main program**. Applications that require a static main are not supported.
- You cannot configure function prototype control (FPC) for a top model that has root message ports.

Generate C++ Messages to Communicate Data Between Simulink Components

To generate C++ code that supports message-based communication between model components in the Simulink environment, use the Simulink Messages & Events Library blocks Send and Receive. To customize the communication, use the Queue block (from the same library) to set parameters for capacity, sorting policy (LIFO, FIFO, and priority), and overwriting policy (behavior when the queue exceeds capacity). You can generate C++ code for GRT-based system target files by using Simulink Coder or for ERT-based system target files by using Embedded Coder.

Messages are an effective communication technique for distributed and complex systems that you can model within Simulink.

How to Prepare Models in Simulink for Message-Based Communication

In Simulink, model message-based communication between model components:

1. Create a model that contains a Send block (referenced model).
2. Create a model that contains a Receive block (referenced model).
3. Create a model that has two Model blocks (top model).
   - Set the first Model block to the model that contains the Send block (model from step 1).
   - Set the second Model block to the model that contains the Receive block (model from step 2).
4. When you run the model, a queue generates in the top model above the message line. A queue, explicit or implicit, controls the message communication. You can use the generated queue or you can add a Queue block in the top model to explicitly specify communication parameters.

How Model Components Pass Messages and How the Code Implements the Behavior

Referenced models pass messages:

1. In the model that contains the Send block, the Send block converts signals into messages.
2. The top model that contains the queue manages messages according to parameters that define capacity, order of delivery, and other quality of service (QoS) metrics.
3. In the model that contains the Receive block, the Receive block converts messages back to signals.

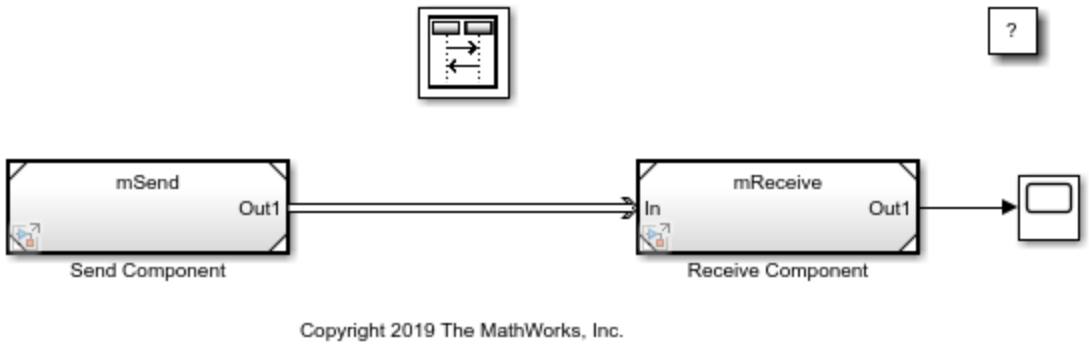
In the generated C++ code, the top model facilitates the connection between the send and receive referenced models by establishing a set interface that the referenced models can access independently of one another.

The generated C++ code implements message behavior:

1. A service is created at each model boundary. A service contains a reference to the top model and an entry point function, referred to as a service function, for referenced models to use to pass messages to the top model.
2. The top model initializes each service to create a connection to each referenced model.
3. The referenced models invoke service functions to pass messages to the top model.

C++ Code Generation Example

This example generates and examines C++ code from the model provided in “Establish Message Send and Receive Interfaces Between Software Components”.



Generate C++ Code:

1. Open the model.
2. In the Apps gallery, click **Embedded Coder**.
3. For each model (top and both referenced models), in the Configuration Parameters dialog box, set these parameters:

   - In the Code Generation pane, set **Language** to `C++`.
   - In the Interface pane, set **Code interface packaging** to `C++ class`.

   Save the model.
4. Generate code. On the **C++ Code** tab, click **Build**.

5 View the generated code. On the **C++ Code** tab, click **View Code**.

Examine C++ Code:

1 A service is created at each model boundary. In C++, services are represented as objects that hold an instance of the top model and a service function that referenced models invoke to pass messages.

   a To view the creation of the services, open the top model C++ file `MessageSendReceiveDefaultBufferModel.cpp`. View the constructor method.

```
// Constructor
MessageSendReceiveDefaultBufferModelModelClass::
  MessageSendReceiveDefaultBufferModelModelClass():
  MessageSendReceiveDefaultBuff_B()
  ,MessageSendReceiveDefaultBuf_DW()
  ,ReceiveComponentRecvData(*this)
  ,SendComponentSendData(*this)
  ,Receive_ComponentMDLOBJ0(get_ReceiveComponentRecvData())
  ,Send_ComponentMDLOBJ1(get_SendComponentSendData())
  ,MessageSendReceiveDefaultBuf_M()
{
  // Currently there is no constructor body generated.
}
```

   The constructor methods, `ReceiveComponentRecvData(*this)` and `SendComponentSendData(*this)`, create the receive and send service objects respectively by taking as an argument a reference to the instance of the top model. Each service object saves a reference to the top model and defines the message interface (the service functions `RecvData` and `SendData`).

   b To view the receive and send service classes, open the top model header file `MessageSendReceiveDefaultBufferModel.h`, and view the following section.

```
// Forward declaration
class MessageSendReceiveDefaultBufferModelModelClass;
class
  MessageSendReceiveDefaultBufferModelModelClassMessa_ReceiveComponent_RecvDataT
  : public RecvData_real_T
{
 private:
  MessageSendReceiveDefaultBufferModelModelClass & aProvider;
 public:
  MessageSendReceiveDefaultBufferModelModelClassMessa_ReceiveComponent_RecvDataT
    (MessageSendReceiveDefaultBufferModelModelClass & aProvider);
  virtual void RecvData(real_T *data, int32_T length, int32_T *status);
};

class
  MessageSendReceiveDefaultBufferModelModelClassMessa_ReceiveComponent_SendDataT
  : public SendData_real_T
{
 private:
  MessageSendReceiveDefaultBufferModelModelClass & aProvider;
 public:
  MessageSendReceiveDefaultBufferModelModelClassMessa_ReceiveComponent_SendDataT
    (MessageSendReceiveDefaultBufferModelModelClass & aProvider);
  virtual void SendData(const real_T *data, int32_T length, int32_T *status);
};
```

2. The top model initializes each service to create a connection to each referenced model. To view the initialization, open the top model C++ file `MessageSendReceiveDefaultBufferModel.cpp`. View the constructor method.

```
// Constructor
MessageSendReceiveDefaultBufferModelModelClass::
  MessageSendReceiveDefaultBufferModelModelClass():
  MessageSendReceiveDefaultBuff_B()
  ,MessageSendReceiveDefaultBuf_DW()
  ,ReceiveComponentRecvData(*this)
  ,SendComponentSendData(*this)
  ,Receive_ComponentMDLOBJ0(get_ReceiveComponentRecvData())
  ,Send_ComponentMDLOBJ1(get_SendComponentSendData())
  ,MessageSendReceiveDefaultBuf_M()
{
  // Currently there is no constructor body generated.
}
```

   The constructor method `Receive_ComponentMDLOBJ0(get_ReceiveComponentRecvData())`, passes a reference to the receive service to the receive referenced model. The constructor method, `Send_ComponentMDLOBJ1(get_SendComponentSendData())`, passes a reference to the send service to the send referenced model.

3. The referenced models invoke service functions to pass messages to the top model. In C++, referenced models invoke the top model (common ancestor, if in a hierarchy) service functions to send or receive messages (specifically, a referenced model invokes the abstract service method (`RecvData` or `SendData`) from the interface created in step 2).The abstract interface classes are emitted to a shared folder. The implementation of the service functions in each service is defined in the top model C++ file.

a To view the abstract interface class to send messages, open from the shared folder the header file `SendData_real_T.h`.

```
class SendData_real_T
{
 public:
  virtual void SendData(const real_T *data, int32_T length, int32_T *status) = 0;
};
```

b To view the implementation of the service function to send messages, open the top model C++ file `MessageSendReceiveDefaultBufferModel.cpp`.

```
// Model step function
void MessageSendReceiveDefaultBufferModelModelClass::
  Messa_ReceiveComponent_SendData(const real_T *data, int32_T length, int32_T
  *status_ptr)                        // Explicit Task: ExplicitTask0
{
  UNUSED_PARAMETER(length);
```

c To view how the send referenced model invokes the service function, open its C++ file `mSend.cpp`. In the step function, the model invokes the service function to send messages to the top model and receives back a return status.

```
// Send: '<Root>/Send'
Out1SendData.SendData(&rtb_SineWave, sizeof(real_T), &status);
```

d To view the abstract interface class to receive messages, open from the shared folder the header file `RecvData_real_T.h`.

```
class RecvData_real_T
{
 public:
  virtual void RecvData(real_T *data, int32_T length, int32_T *status) = 0;
};
```

e To view the implementation of the service function to receive messages, open top model C++ file `MessageSendReceiveDefaultBufferModel.cpp`.

```
// Model step function
void MessageSendReceiveDefaultBufferModelModelClass::
  Messa_ReceiveComponent_RecvData(real_T *data, int32_T length, int32_T
  *status_ptr)                        // Explicit Task: ExplicitTask0
{
  UNUSED_PARAMETER(length);
```

f To view how the receive referenced model invokes the service function, open its C++ file `mRecieve.cpp`. In the step function, the model invokes the service function to receive messages and a status from the top model.

```
// Output and update for referenced model: 'mReceive'
void mReceiveModelClass::step(real_T *rty_Out1)
{
  int32_T status = 1;

  // Receive: '<Root>/Receive'
  status = 1;
  InRecvData.RecvData(rty_Out1, sizeof(real_T), &status);
}
```

Considerations and Limitations

- C++ code support is available for GRT-based system target files by using the Simulink Coder App.
- C++ code support is available for ERT-based system target files by using the Embedded Coder App.
- To generate code, for top and referenced models. select the same language (C++) and the same system target file.
- You cannot configure function prototype control (FPC) for a top model that has root message ports.
- External models and variant models are not supported.
- Software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations are not supported.

Generate C Messages to Communicate Data Between Simulink Components

To generate C code that supports message-based communication between model components in the Simulink environment, use the Simulink Messages & Events Library blocks Send and Receive. To customize the communication, use the Queue block (from the same library) to set parameters for capacity, sorting policy (LIFO, FIFO, and priority), and overwriting policy (behavior when the queue exceeds capacity). You can generate C code for GRT-based system target files by using Simulink Coder or for ERT-based system target files by using Embedded Coder.

Messages are an effective communication technique for distributed and complex systems that you can model within Simulink.

How to Prepare Models in Simulink for Message-Based Communication

In Simulink, you can model message-based communication between model components:

1. Create a model that contains a Send block (referenced model).
2. Create a model that contains a Receive block (referenced model).
3. Create a model that has two Model blocks (top model).
   - Set the first Model block to the model that contains the Send block (model from step 1).
   - Set the second Model block to the model that contains the Receive block (model from step 2).
4. When you run the model, a queue generates in the top model above the message line. A queue, explicit or implicit, controls the message communication. You can use the generated queue or you can add a Queue block in the top model to explicitly specify communication parameters.

How Model Components Pass Messages and How the Code Implements the Behavior

Referenced models pass messages:

1. In the model that contains the Send block, the Send block converts signals into messages.
2. The top model that contains the queue manages messages according to parameters that define capacity, order of delivery, and other quality of service (QoS) metrics.
3. In the model that contains the Receive block, the Receive block converts messages back to signals.

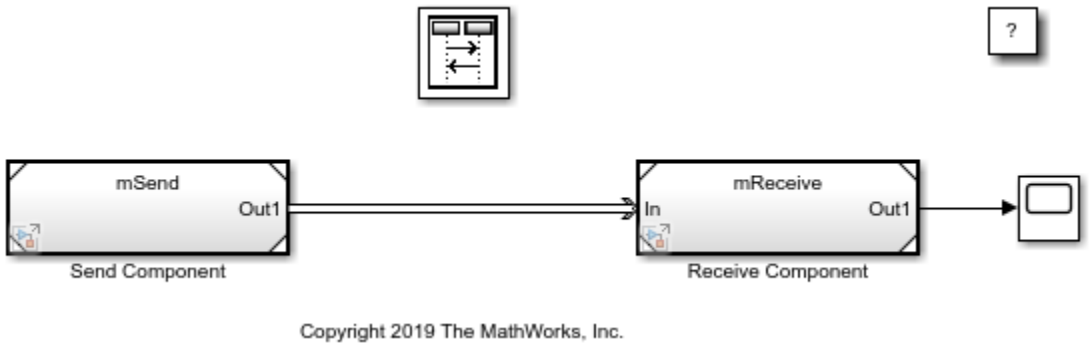
In the generated C code, the top model facilitates the connection between the send and receive referenced models by establishing a set interface that the referenced models can access independently of one another.

The generated C code implements message behavior:

1. A service is created at each model boundary. A service contains a reference to the top model and an entry point function, referred to as a service function, for referenced models to use to pass messages to the top model.
2. The top model initializes each service to create a connection to each referenced model.
3. The referenced models invoke service functions to pass messages to the top model.

C Code Generation Example

This example generates and examines C code from the model provided in “Establish Message Send and Receive Interfaces Between Software Components”.



Generate C Code:

1. Open the model.
2. In the Apps gallery, click **Embedded Coder**.
3. For each model (top and both referenced models), in the Configuration Parameters dialog box, in the Code Generation pane, set **Language** to `C` and save the model.
4. Generate code. On the **C Code** tab, click **Build**.
5. View the generated code. On the **C Code** tab, click **View Code**.

Examine C Code:

1. A service is created at each model boundary. In C, referenced models represent services as DWork. The service provides a pointer to an instance of the top model and a service function that referenced models invoke to pass messages.

   a. To view the service to send messages, open the send referenced model header file `mSend.h`. View the DWork allocation.

```
/* Block states (default storage) for model 'mSend' */
typedef struct {
  SendData_real_T Out1_SendData;       /* synthesized block */
  real_T lastSin;                      /* '<S1>/Sine Wave' */
  real_T lastCos;                      /* '<S1>/Sine Wave' */
  int32_T systemEnable;                /* '<S1>/Sine Wave' */
} DW_mSend_f_T;
```

   b. To view the send message data type, if the data type is shareable (for example, built-in data types, imported bus types, or exported bus types that have a specified data type) the information is located in a shared header file. If the data type is not shareable, the information is located the model header file `mSend.h`. For this example, view the shareable data type by opening the shared header file `SendData_real_T.h`.

```
typedef struct {
  void *host;
  void (*SendData)(void *, const real_T *, int32_T, int32_T *);
} SendData_real_T;
```

   c. To view the service to receive messages, open the receive referenced model header file, `mReceive.h`. View the DWork allocation.

```
/* Block states (default storage) for model 'mReceive' */
typedef struct {
  RecvData_real_T In_RecvData;         /* synthesized block */
} DW_mReceive_f_T;
```

   d. To view the received messages data type, if the data type is shareable (for example,built-in data types, imported bus types, or exported bus types that have a specified data type) the information is located in a shared header file. If the data type is not shareable the information is located the model header file, `mReceive.h`. For this example, view the shareable data type by opening the shared header file `RecvData_real_T.h`.

```
typedef struct {
  void *host;
  void (*RecvData)(void *, real_T *, int32_T, int32_T *);
} RecvData_real_T;
```

2. The top model initializes each service to create a connection to each referenced model. In C, the top model initializes each referenced model DWork. To view the initialization, open the top model C file `MessageSendReceiveDefaultBufferModel.c`.

```
/* Model initialize function */
void MessageSendReceiveDefaultBufferModel_initialize(void)
{
  /* Registration code */

  /* states (dwork) */
  MessageSendReceiveDefaultBuf_DW.SendComponent_InstanceData.rtdw.Out1_SendData.SendData
    = Messa_ReceiveComponent_SendData;
  MessageSendReceiveDefaultBuf_DW.SendComponent_InstanceData.rtdw.Out1_SendData.host
    = MessageSendReceiveDefaultBuf_M;
  MessageSendReceiveDefaultBuf_DW.ReceiveComponent_InstanceData.rtdw.In_RecvData.RecvData
    = Messa_ReceiveComponent_RecvData;
  MessageSendReceiveDefaultBuf_DW.ReceiveComponent_InstanceData.rtdw.In_RecvData.host
    = MessageSendReceiveDefaultBuf_M;
```

3. The referenced models invoke service functions to pass messages to the top model. In C, a referenced model invokes a service function by dereferencing the service function pointer and passing a pointer to the instance of the top model. You can view the prototypes of the service functions in the top model header file. You can view the referenced model invocations of those service functions in the referenced model C files.

   a. To view the prototype of the service function to send messages, open the top model header file, `MessageSendReceiveDefaultBufferModel.h`.

```
/* Exported entry point function */
extern void Messa_ReceiveComponent_SendData(void *host, const real_T *data,
  int32_T length, int32_T *status_ptr);
```

   b. To view the implementation of the service function to send messages, open the top model C file, `MessageSendReceiveDefaultBufferModel.c`.

```
/* Model step function */
void Messa_ReceiveComponent_SendData(void *host, const real_T *data, int32_T
  length, int32_T *status_ptr)         /* Explicit Task: ExplicitTask0 */
{
  UNUSED_PARAMETER(host);
  UNUSED_PARAMETER(length);

  /* DiscreteEventSubgraph generated from: '<Root>/Receive Component' */
  *status_ptr = Messa_ReceiveComponent_SendDa_l(data);
}
```

```
static int32_T Messa_ReceiveComponent_SendDa_l(const real_T *data)
{
  int32_T status = 1;
  Msg_real_T msg;

  /* DiscreteEventSubgraph generated from: '<Root>/Receive Component' */
  msg = MessageSendReceiveDefaultBufferModel_createMsg(data);

  /* Queue generated from: '<Root>/Receive Component' incorporates:
   *  DiscreteEventSubgraph generated from: '<Root>/Receive Component'
   */
  if (MessageSendReceiveDefaultBuff_l
      (&MessageSendReceiveDefaultBuf_DW.Queue_InsertedFor_ReceiveCompon, &msg))
  {
    status = 0;
  }

  /* End of Queue generated from: '<Root>/Receive Component' */
  return status;
}
```

c To view how the send referenced model invokes the service function, open the C file for the model `mSend.c`. In the step function, the model invokes the service to send instance data and a message to the top model and receives back a return status.

```
/* Send: '<Root>/Send' */
if (localDW->Out1_SendData.SendData != NULL) {
  localDW->Out1_SendData.SendData(localDW->Out1_SendData.host, &rtb_SineWave,
    sizeof(real_T), &status);
}
```

d To view the prototype of the service function to receive messages, open the top model header file `MessageSendReceiveDefaultBufferModel.h`.

```
/* Exported entry point function */
extern void Messa_ReceiveComponent_RecvData(void *host, real_T *data, int32_T
  length, int32_T *status_ptr);
```

e To view the implementation of the service function to receive messages, open the top model C file `MessageSendReceiveDefaultBufferModel.c`.

```
/* Model step function */
void Messa_ReceiveComponent_RecvData(void *host, real_T *data, int32_T length,
  int32_T *status_ptr)                /* Explicit Task: ExplicitTask0 */
{
  UNUSED_PARAMETER(host);
  UNUSED_PARAMETER(length);

  /* DiscreteEventSubgraph generated from: '<Root>/Receive Component' */
  *status_ptr = Messa_ReceiveComponent_RecvDa_l(data);
}
```

```
static int32_T Messa_ReceiveComponent_RecvDa_l(real_T *data)
{
  int32_T status = 1;
  Msg_real_T msg;

  /* Queue generated from: '<Root>/Receive Component' incorporates:
   *  DiscreteEventSubgraph generated from: '<Root>/Receive Component'
   */
  if (MessageSendReceiveDefaultBuf_lh
      (&MessageSendReceiveDefaultBuf_DW.Queue_InsertedFor_ReceiveCompon, &msg))
  {
    status = 0;
    *data = msg.fData;
  }

  /* End of Queue generated from: '<Root>/Receive Component' */
  return status;
}
```

f To view how the receive referenced model invokes the service function, open the C file for the model `mReceive.c`. In the step function, the model invokes the service to receive a message payload and a return status.

```
/* Output and update for referenced model: 'mReceive' */
void mReceive(real_T *rty_Out1, DW_mReceive_f_T *localDW)
{
  int32_T status = 1;

  /* Receive: '<Root>/Receive' */
  status = 1;
  if (localDW->In_RecvData.RecvData != NULL) {
    localDW->In_RecvData.RecvData(localDW->In_RecvData.host, rty_Out1, sizeof
      (real_T), &status);
  }

  /* End of Receive: '<Root>/Receive' */
}
```

Considerations and Limitations

- C code support is available for GRT-based system target files by using the Simulink Coder App.
- C code support is available for ERT-based system target files by using the Embedded Coder App.
- To generate code, for top and referenced models, select the same language (C) and the same system target file.
- You cannot configure function prototype control (FPC) for a top model that has root message ports.
- External models and variant models are not supported.
- Software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations are not supported.

Model Message-Based Communication Integrated with POSIX Message Queues

This example shows how to integrate generated C++ code with POSIX to facilitate message communication. The example application is a multisensor positioning system designed to estimate the

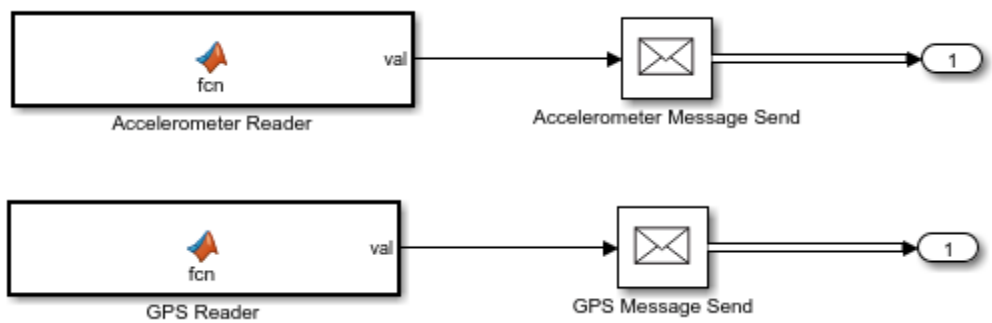
position of a vehicle. The multisensor positioning system is composed of two components: sensors (multiple models) and an estimation model. The outlined workflow shows how to prepare each component, generate code, and integrate the generated code to pass messages between system components. This example assumes you are familiar with using messages in the Simulink environment. For more information, see “Generate C++ Messages to Communicate Data Between Simulink and an Operating System or Middleware” on page 3-33.

Sensor Models

The first component in the multisensor positioning system is the sensors. The positioning system uses an accelometer and a GPS sensor to estimate the vehicle position. Each sensor model is composed of a Function block, Send block, and Output port. Each sensor model sends messages to the estimation model by using POSIX messages.

Prepare Models

Open the sensor models `AccelerometerSendMessages` and `GPSSendMessages`.



For each model, perform these steps:

1. In the Apps Gallery, click **Embedded Coder**.
2. On the toolstrip, set the **Stop time** to `100` seconds. The stop time defines the duration of the data collection when estimating the vehicle position.
3. In the model, open the Block Parameters for the Function block and set the **Sample time** to `0.01`.

Generate Code

1. For each sensor model, in the Configuration Parameters dialog box, set these parameters:

- On the **Code Generation** pane, set **Language** to `C++`.
- On the **Interface** pane , set **Code interface packaging** to `C++ class` and **Multi-instance code error diagnostic** to `none`. Under Advanced parameters, select `MAT-file logging`.

2. Generate code. On the **C++ Code** tab, click **Build**.

Integrate Code

To integrate the generated C++ code from each sensor model with POSIX, hand write the implementation of the send class and generated main program (or create your own). This example provides a customized main program with an implemented send class. To integrate the code from each sensor model, replace the generated main file with the customized main file.

Review the customized main file for each sensor:

1. The customized main file for each sensor is implemented in the same way. As an example, open the customized main file `AccelerometerSendMessages_ert_main_customized.cpp`.

2. Review the implemented send class:

- The send class function, `SendData`, calls into the POSIX API (`mq_send`) to send messages.
- To provide access to a message queue, a POSIX message queue is added as a class variable and a set method is implemented to set the queue.

```
class accelSendModelClassSendData_real_T : public SendData_real_T{
     mqd_t msgQueue;

     public:
     // Use POSIX to send messages
     void SendData(const real_T* data, int32_T length, int32_T* status)
     {
         unsigned int priority = 1;
         *status = mq_send(msgQueue, (char*)data, length, priority);
     }

     // Set method for class POSIX queue
     void SetMQ(mqd_t mq)
     {
         msgQueue = mq;
     }
 };
```

After the send class implementation, an instance of the send class (a send object) is created. The send object is then attached to the sensor model to enable the sensor to send messages to the estimate model.

```
// Create send object
static accelSendModelClassSendData_real_T Out1SendData_arg;

// Attach send object to sensor model to send messages
static accelSendModelClass rtObj( Out1SendData_arg);
```

3. Review the added POSIX queue behavior functions. The main function uses these functions to open and close the POSIX queue.

```
mqd_t openMQ(const char *name, int flags, int maxmsg, int msgsize)
 {
     struct mq_attr attr;
     attr.mq_flags = 0;
     attr.mq_maxmsg = maxmsg;
     attr.mq_msgsize = msgsize;

     mqd_t msgQueue = mq_open(name, flags, 0664, &attr);

     if (msgQueue < 0) {
         printf("mq_open failed\n");
         exit(1);
     }

     return msgQueue;
 }
```

```
void closeMQ(mqd_t msgQueue)
 {
     mq_close(msgQueue);
 }
```

4. Review the main function. The implemented main function uses the previously defined class and queue functions to send messages and to manage the POSIX message queue. The main function performs the following:

- Opens a POSIX message queue.
- Sends messages.
- Closes the POSIX queue.

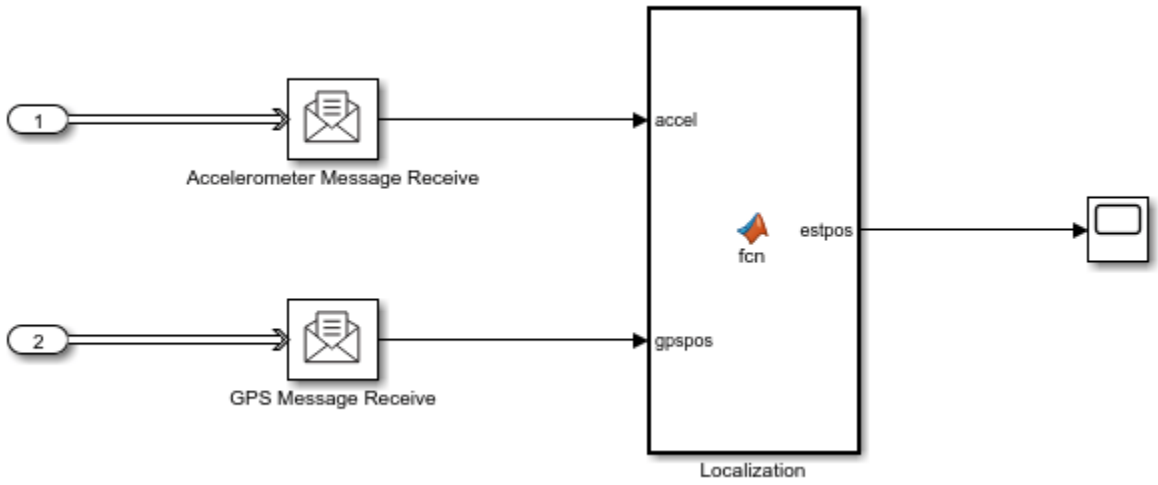
```
int_T main(int_T argc, const char *argv[])
 {
     // Unused arguments
     (void)(argc);
     (void)(argv);

     // Initialize model
     rtObj.initialize();

     // Open POSIX message queue
     mqd_t msgQueue = openMQ("/PosixMQ_accel", O_CREAT | O_WRONLY, 2, 16);
     Out1SendData_arg.SetMQ(msgQueue);

     // Send messages
     while (rtmGetErrorStatus(rtObj.getRTM()) == (NULL)) {
     rt_OneStep();
     }

     // Matfile logging
     rt_StopDataLogging(MATFILE, rtObj.getRTM()->rtwLogInfo);

     // Disable rt_OneStep() here

     // Close POSIX message queue
     closeMQ(msgQueue);

     return 0;
 }
```

Estimate Model

The second component is the estimate model. The estimate model receives messages that have input data from the sensor models and calculates the estimated position of the vehicle.

Prepare Model

Open the model `PositionEstimateMessages`.



1. In the Apps Gallery, click **Embedded Coder**.
2. On the toolstrip, set the **Stop time** to `100` seconds.
3. In the model, double-click to open each inport. In the Block Parameters, set **Data type** to `double`, **Port dimensions** to `3`, and **Sample time** to `0.01`.
4. Open each Message Receive block. Clear the **Use internal queue** parameter to enable the model to use a POSIX message queue.
5. Open the Function block and view the implementation of the estimation algorithm.

Generate Code

1. In the Configuration Parameters dialog box, set these parameters:

- On the **Code Generation** pane, set **Language** to `C++`.
- On the **Interface** pane, set **Code interface packaging** to `C++ class` and set **Multi-instance code error diagnostic** to `none`. Under Advanced parameters, select `MAT-file logging`.

2. Generate code. On the **C++ Code** tab, click **Build**.

Integrate Code

To integrate the generated C++ code from the estimate model with POSIX, hand write the implementation of the receive class and generated main program (or create your own). The estimate model receives messages from the sensor models and calculates an estimated position of the vehicle. This examples provides a customized main program with an implemented receive class. To integrate the code from the estimate model, replace the generated main file with the customized main file.

Review the customized main file:

1. Open the customized main file `PositionEstimateMessages_ert_main_customized.cpp`.

2. Review the implemented receive class:

- The receive class function, `RecvData`, calls into the POSIX API (`mq_receive`) to receive messages.

- To provide access to a message queue, a POSIX message queue is added as a class variable and a set method is implemented to set the queue.

```
class positionEstimateModelClassRecvData_real_T : public RecvData_real_T{
     mqd_t msgQueue;

     public:
     // Uses POSIX API mq_receive to receive messages
     void RecvData(real_T* data, int32_T length, int32_T* status)
     {
     // Add receive data logic here
     unsigned int priority = 1;
     *status = mq_receive(msgQueue, (char *)data, length, &priority);
     }

     // Set method for class POSIX queue
     void SetMQ(mqd_t mq)
     {
         msgQueue = mq;
     }
  };
```

After the receive class implementation, an instance of the receive class (a receive object) is created for each sensor model. The receive objects are then attached to the estimate model to enable the model to receive messages from each sensor.

```
// Create receive objects
static positionEstimateModelClassRecvData_real_T In1RecvData_arg;
static positionEstimateModelClassRecvData_real_T In2RecvData_arg;

// Attach receive objects to estimate model to receive messages
static positionEstimateModelClass rtObj( In1RecvData_arg, In2RecvData_arg);
```

3. Review the added POSIX queue behavior functions. The main function uses these functions to open, close, and unlink the POSIX queue.

```
mqd_t openMQ(const char *name, int flags, int maxmsg, int msgsize)
 {
     struct mq_attr attr;
     attr.mq_flags = 0;
     attr.mq_maxmsg = maxmsg;
     attr.mq_msgsize = msgsize;

     mqd_t msgQueue = mq_open(name, flags, 0664, &attr);

     if (msgQueue < 0) {
         printf("mq_open failed\n");
         exit(1);
     }

     return msgQueue;
 }

void closeMQ(mqd_t msgQueue)
 {
     mq_close(msgQueue);
 }

void unlinkMQ(const char *name)
 {
```

```
    if (mq_unlink(name) < 0) {
        printf("mq_unlink failed\n");
        exit(1);
    }
 }
```

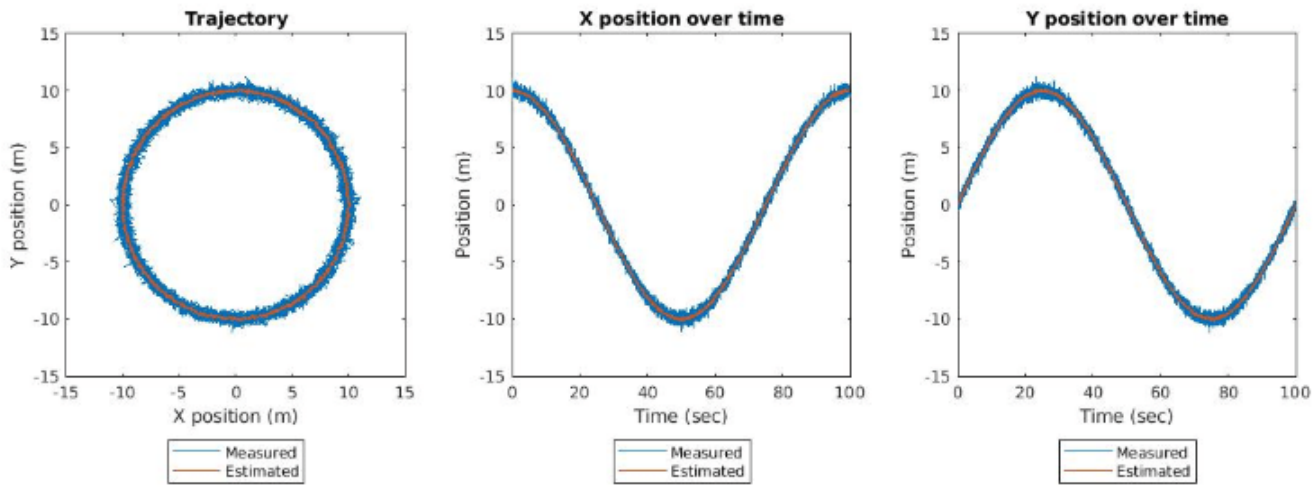
4. Review the main function. The implemented main function uses the previously defined class and queue functions to receive messages and to manage the POSIX queue. The main function performs the following:

- Opens a POSIX message queue.
- Receives messages.
- Closes the POSIX queue.

```
int_T main(int_T argc, const char *argv[])
 {
    // Unused arguments
    (void)(argc);
    (void)(argv);

    // Initialize model
    rtObj.initialize();

    // Open POSIX message queues
    mqd_t msgQueueAccel = openMQ("/PosixMQ_accel", O_CREAT | O_RDONLY, 2, 16);
    mqd_t msgQueueGPS = openMQ("/PosixMQ_gpspos", O_CREAT | O_RDONLY, 2, 16);

    In1RecvData_arg.SetMQ(msgQueueAccel);
    In2RecvData_arg.SetMQ(msgQueueGPS);

    // Receive messages
    while (rtmGetErrorStatus(rtObj.getRTM()) == (NULL)) {
        rt_OneStep();
    }

    // Matfile logging
    rt_StopDataLogging(MATFILE, rtObj.getRTM()->rtwLogInfo);

    // Disable rt_OneStep() here

    // Close POSIX message queues
     closeMQ(msgQueueAccel);
     closeMQ(msgQueueGPS);

     unlinkMQ("/PosixMQ_accel");
     unlinkMQ("/PosixMQ_gpspos");

    return 0;
 }
```

Execute Multisensor Positioning System

To execute the multisensor positioning system, rebuild and run the executables for all models. To view results, run the provided build script `positioning_system_script`. An example of results is shown in this figure.



Generate Code for Wireless Sensor in Tire Pressure Monitoring Messaging System

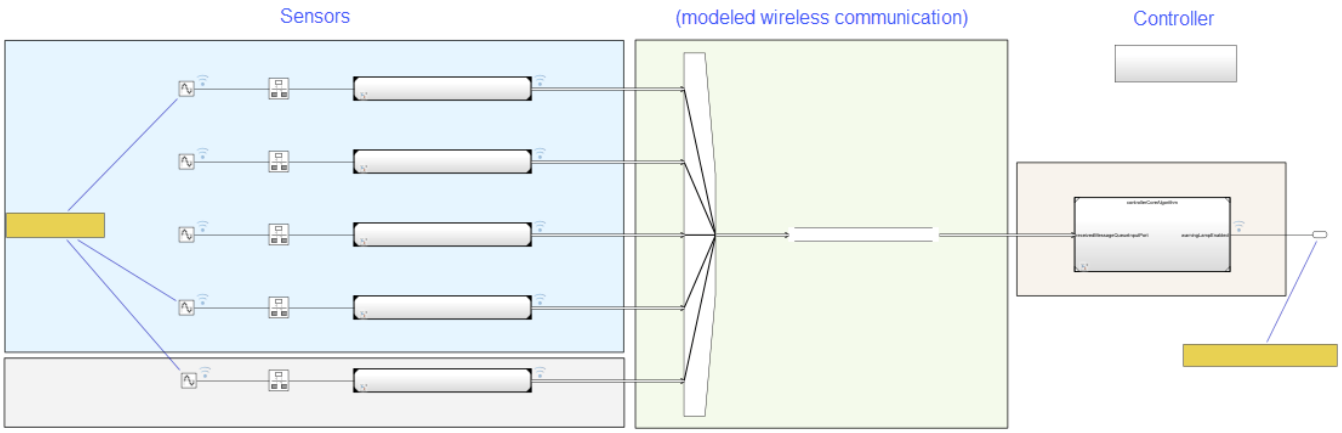
This example shows how to generate code for individual embedded components in a distributed monitoring system. It also describes integrating target-specific external code for message protocol services, data acquisition from a physical environment, and nonvolatile memory (NVM) management on an embedded target.

The example focuses on how to generate C code for a sensor in a tire pressure monitoring system and follows this general workflow:

1. Review the distributed messaging system.
2. Access models and other project files.
3. Open the wrapper model for the sensor.
4. Locate Simulink® blocks that contain hooks for inserting external code.
5. Integrate custom target-specific driver and middleware code.
6. Build and generate C code from the sensor wrapper.
7. Inspect the generated makefile and build information.
8. Inspect the generated C code for the sensor.

For more information about the system-level modeling and core algorithms in the sensor and component models, see “Wireless Tire Pressure Monitoring System with Fault Logging”.

Review Distributed Messaging System



The messaging system highlighted in this example represents an automotive tire pressure monitoring system that uses identical pressure sensors at each wheel.

Sensors communicate wirelessly with the controller in the vehicle and publish periodic messages consisting of acquired pressure data and a unique device ID. The sensor device ID is stored in nonvolatile memory on the sensor and serves as the message topic ID. Device IDs are used to uniquely identify which sensors are associated with the tires installed in the vehicle, as the vehicle might be among many other cars in traffic and erroneous messages must be identified and rejected.

The controller receives the messages and inspects the sender device ID against a list of the relevant sensor IDs to determine which communications to examine and which ones to reject. The controller then evaluates payload pressure data from the relevant sensors for fault conditions. If the controller detects a fault condition, it indicates an error and aggregates the fault condition in a log that is stored in controller NVM.

The system-level model `tirePressureMonitoringSystem` contains multiple identical tire pressure sensors modeled by multi-instanced Model blocks that are routed to the controller. Messages are sent by using the Message Send block, which reads the value of an input signal and converts it to a message that carries the signal value. Since all four sensors in the vehicle execute the same algorithm, you deploy the same code to each sensor. The only difference among the sensors is the unique sensor ID, which is burned into the NVM of each sensor before code execution.

The models and deployment steps in this example support a manufacturing process similar to the following:

1. Engineers update the design of the core algorithm of the sensor (`SensorCoreAlgorithm`) as needed and verify their design at the system level by simulating the system model `tirePressureMonitoringSystem.slx`.
2. Engineers isolate the sensor functionality for code generation by referencing the sensor core algorithm model in a wrapper model (`SensorCodeGenerationWrapper.slx`).
3. Engineers generate code from the `SensorCodeGenerationWrapper.slx` model and provide the compiled code to the device manufacturer.
4. The manufacturer then produces hundreds of identical sensors, each running the exact same code.
5. As the car is being assembled, the manufacturer flashes unique ID numbers into the NVM of each sensor.

6 As each sensor powers up, the sensor runs the exact same code as all the others, but that code reads the unique ID number from NVM, seeding it to send uniquely addressed messages.

The following sections in this example focus on steps 2 and 3 of the overall manufacturing process and explain how to generate deployable code for the sensor component in this distributed messaging system. For more information about modeling the sensor core algorithm or system-level simulation, see “Wireless Tire Pressure Monitoring System with Fault Logging”.

Access Models and Other Project Files

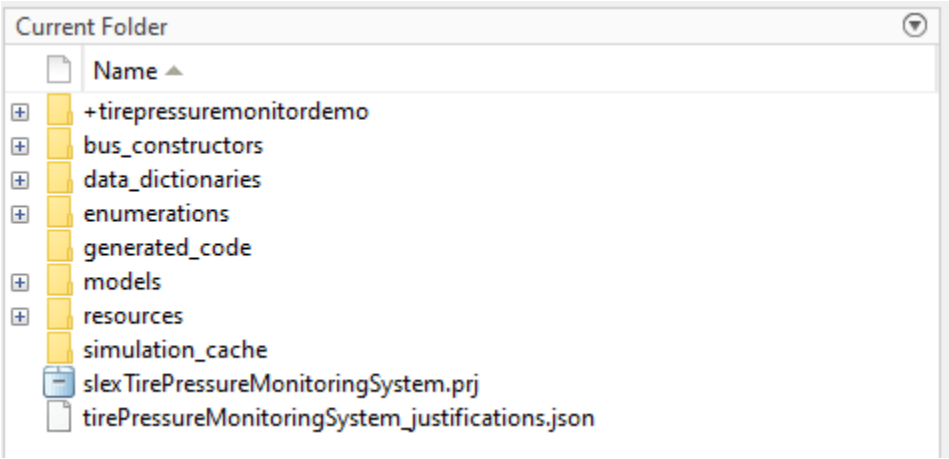
The models and support files for this example are contained in a Simulink project.

To access the project files, enter `slexTirePressureMonitoringSystem` at the MATLAB® command line.

Opening the project:

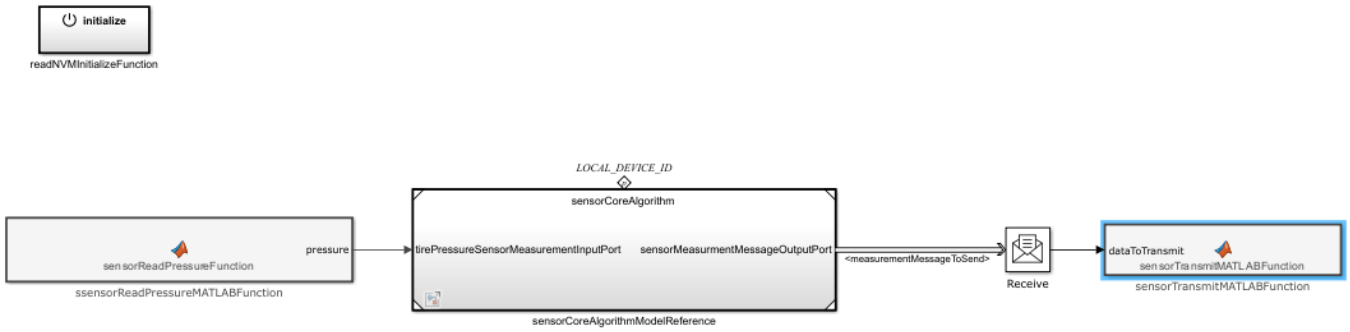
- Creates a working folder under your MATLAB current folder
- Unzips project files to that working folder
- Runs the project initialization script

After the project opens, your working folder contains the top-level items shown in this image.



Open Wrapper Model for Sensor

In the current folder, navigate to `models > sensor > wrappers`, and open `sensorCodeGenerationWrapper.slx`.



The wrapper model `sensorCodeGenerationWrapper` is a top-level model that isolates the sensor core model for code generation. The base algorithm for the sensor model converts pressure readings into messages that can be used with a messaging protocol and is implemented in `sensorCoreAlgorithmModelReference`, a reference to the `sensorCoreAlgorithm` model in the system-level model.

Locate Simulink Blocks that Contain Hooks for Inserting External Code

How you read NVM, acquire pressure data, and transmit messages depends on what operating system, tire pressure monitoring system (TPMS) sensor, and messaging protocol you are using. For this example, those system-dependent tasks require custom externally-written code.

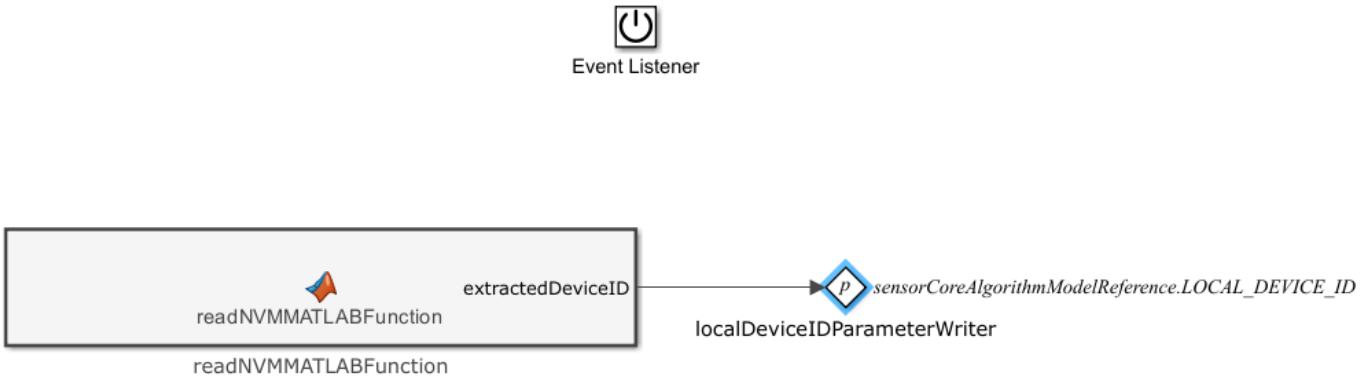
MATLAB Function blocks in the wrapper model contain hooks for inserting the external code into the generated C code. This example uses the abstract class `coder.ExternalDependency` for code insertion.

In the sensor wrapper, the following blocks call public methods defined in the class `SensorTargetSpecificCodeExternalDependency`, a subclass of `coder.ExternalDependency`:

- `sensorReadPressureMATLABFunction` — MATLAB Function block that supports reading pressure data
- `sensorTransmitMATLABFunction` — MATLAB Function block that supports publishing transmit messages
- `readNVMInitializeFunction` — Initialization block that runs at system startup and supports accessing NVM

For managing the sensor ID in NVM at startup, `readNVMInitialization` contains these blocks:

- Event Listener block
- MATLAB Function block `readNVMMATLABFunction`
- Parameter Writer block `localDeviceIDParameterWriter`



To explore hooks in `readNVMInitialization`:

1. In the sensor wrapper, open the Initialization block `readNVMInitialization`. On start up, the Parameter Writer block `localDeviceIDParameterWriter` initializes the local sensor ID variable `sensorCoreAlgorithmModelReference.LOCAL_DEVICE_ID` to the value of `SENSOR_PARAMETERS_NVM_ADDRESS_AS_UINT32` read from NVM.
2. Open the MATLAB Function block `readNVMMATLABFunction` to view the code.

```
function extractedDeviceID = readNVMMATLABFunction(SENSOR_PARAMETERS_NVM_ADDRESS_AS_UINT32)
coder.extrinsic( 'warning' );
defaultDeviceID = deviceIDBusTypeDefaultConstructor( );

if( coder.target( 'Rtw' ) )
    extractedDeviceID = ...
        SensorTargetSpecificCodeExternalDependency.sensorReadNVMMatlabWrapper( SENSOR_PARAMETERS_
else
    warning( 'This block is intended only for code generation.' );
    extractedDeviceID = defaultDeviceID;
end
return;

end
```

The function `readNVMMATLABFunction` references the `coder.ExternalDependency` subclass `SensorTargetSpecificCodeExternalDependency`.

To view the class definition for `SensorTargetSpecificCodeExternalDependency` and code for creating the function `readNVMMATLABFunction`:

1. In your current folder, navigate to `models > sensor > target_specific_code > @SensorTargetSpecificCodeExternalDependency`.
2. Open `SensorTargetSpecificCodeExternalDependency.m`.
3. Scroll to the method definitions in `SensorTargetSpecificCodeExternalDependency.m`.

```
28
29      methods( ...
30              Abstract = false, ...
31              Access = public, ...
32              Hidden = false, ...
33              Static = true, ...
34              Sealed = true ...
35              )
36
37          function [populatedVariable] = sensorReadNVMMatlabWrapper(nvmSourceAddressAsUint32,exampleVariableDataTypeAndDims)
38
39              coder.cinclude('stddef.h');
40
41              coder.cinclude(SensorTargetSpecificCodeExternalDependency.HEADER_FILE_SHORT_NAME);
42
43              assert((true == isa(nvmSourceAddressAsUint32,'uint32')), ...
44                  'The source address must be a "uint32".');
45
46              assert((true == isscalar(nvmSourceAddressAsUint32)), ...
47                  'The source address must be a scalar.');
48
49              populatedVariable = exampleVariableDataTypeAndDims;
50
51              elementSize = coder.opaque('size_t','0');
52
53              elementSize = coder.ceval('sizeof',exampleVariableDataTypeAndDims);
54
55              elementQuantity = cast(numel(exampleVariableDataTypeAndDims),'like',coder.opaque('size_t','0'));
56
57              sourceAddress = ...
58                  cast(nvmSourceAddressAsUint32,'like',coder.opaque('void *','NULL'));
59
60              coder.ceval('sensor_read_nvm',sourceAddress,coder.wref(populatedVariable),elementSize,elementQuantity);
61
62              return;
63          end
```

Note that the `coder.ceval` statement invokes the external source code in `sensor_read_nvm` to the code generator.

You can repeat these steps to explore the MATLAB Function blocks for inserting custom code that acquires the pressure readings and transmits messages:

- `sensorReadPressureMATLABFunction`, which uses the external source code sensor_read_pressure
- `sensorTransmitMATLABFunction`, which uses the external source code sensor_transmit_data

For more information about using `coder.ExternalDependency` for code integration, see “Develop Interface for External C/C++ Code”.

Integrate Custom Target-Specific Driver and Middleware Code

You customize this example by incorporating driver and middleware code specific to your system. You include your custom code in the generated code by updating the preexisting stub functions in the source and header files:

- `sensor_hand_code_source.c`
- `sensor_hand_code_header.h`

To include target-specific code:

1. In your current folder, navigate to `models > sensor > target_specific_code > source`.
2. Open `sensor_hand_code_source.c`.
3. Insert your custom code into the existing `sensor_read_nvm`, `sensor_read_pressure`, and `sensor_transmit_data` stub functions.

```
/*
* The functions below are stubbed to allow compilation. However, the premise of the example is th
* are target-specific and would not be compiled for desktop simulation.
*/

#include "sensor_hand_code_header.h"

void sensor_read_nvm( void * nvm_source_address, void * ram_destination_addresss, size_t single_e
    return;
}

extern double sensor_read_pressure( void ){
   double result = 0.0;
   return( result );
}

void sensor_transmit_data( void * message_source_address, size_t single_element_size, size_t elem
    return;
}
```

The functions insert into the generated code by using the `coder.ceval` command in the MATLAB Function block definitions.

Build and Generate C Code from Sensor Wrapper

Open the **Embedded Coder** app and build the sensor wrapper. Simulink builds the models and support files and generates code.

Generated code for the sensor wrapper and referenced core algorithm is located under the `generated_code` folder in your current folder:

- `sensorCodeGenerationWrapper_ert_rtw` — Generated code for the sensor wrapper, including the generated makefile `sensorCodeGenerationWrapper.mk`
- `slprj` — Generated code for shared utilities, including the code for the sensor core algorithm `sensorCoreAlgorithm`

Inspect Generated Makefile and Build Information

Embedded Coder generates the customized makefile `sensorCodeGenerationWrapper.mk` by modifying a template makefile with specifics of the Simulink models and further modifications introduced by using the `coder.ExternalDependency` subclass.

Specifically, `SensorTargetSpecificCodeExternalDependency.m` invokes `coder.ExternalDependency.updateBuildInfo` to modify the build information object `buildInfo` and provide additional information required to link to external code. This information includes the external files defined in the class.

```
properties( ...
       ...
       )
    HEADER_FILE_SHORT_NAME = 'sensor_hand_code_header.h';
    SOURCE_FILE_SHORT_NAME = 'sensor_hand_code_source.c';
end

methods( ...
       ...
       )
```

```
...
function [ ] = updateBuildInfo(buildInfo,buildContext)

     externalCodeRootDirectoryAbsolutePath = ...
         SensorTargetSpecificCodeExternalDependency.getHandCodeRootDirectoryAbsolutePath( );

     headerFileDirectoryAbsolutePath = ...
         fullfile(externalCodeRootDirectoryAbsolutePath,'include');

     buildInfo.addIncludePaths(headerFileDirectoryAbsolutePath);

     sourceFileDirectoryAbsolutePath = ...
         fullfile(externalCodeRootDirectoryAbsolutePath,'source');

     buildInfo.addSourceFiles(SensorTargetSpecificCodeExternalDependency.SOURCE_FILE_SHORT_NA

     buildInfo.addSourcePaths(sourceFileDirectoryAbsolutePath);

     return;

 end
```

When you build the wrapper model, the code generator adds external code to the makefile and build information.

Open `sensorCodeGenerationWrapper.mk` located in `generated_code` > `sensorCodeGenerationWrapper_ert_rtw` to review the external source files and objects, which include `sensor_hand_code_source.c` and `sensor_hand_code_source.obj`, respectively. An example of this in the makefile is:

```
...
## SOURCE FILES
SRCS = sensor_hand_code_source.c $(START_DIR)\sensorCodeGenerationWrapper_ert_rtw\sensorCodeGene
MAIN_SRC = $(START_DIR)\sensorCodeGenerationWrapper_ert_rtw\ert_main.c
ALL_SRCS = $(SRCS) $(MAIN_SRC)

## OBJECTS
OBJS = sensor_hand_code_source.obj sensorCodeGenerationWrapper.obj sensorCodeGenerationWrapper_da
MAIN_OBJ = ert_main.obj
ALL_OBJS = $(OBJS) $(MAIN_OBJ)
...
```

The absolute path for the external source file is defined in `updateBuildInfo` by using `addSourcePaths`. You can review the makefile to confirm the absolute path for the target object for the external source code `sensor_hand_code_source.c`.

```
## INTERMEDIATE TARGETS
# SOURCE-TO-OBJECT

.c.obj :
   $(CC) $(CFLAGS) -Fo"$@" "$<"
...
{C:\<yourPath>\examples\slexTirePressureMonitoringSystem\models\sensor\target_specific_code\sourc
   $(CC) $(CFLAGS) -Fo"$@" "$<"
```

The absolute path for the external include file is defined in `updateBuildInfo` by using `addIncludePaths`. The path for the external header file `sensor_hand_code_source.h` is specified in the response file `sensorCodeGenerationWrapper_comp.rsp`, which is a text file that contains data to include as command-line arguments to the Windows compiler.

Open the file `sensorCodeGenerationWrapper_comp.rsp`, which is also located in `generated_code` > `sensorCodeGenerationWrapper_ert_rtw`, to review the include folders and confirm the folder path for the external target-specific header file. An example of the include arguments is:

```
...
-IC:/<yourPath>/examples/slexTirePressureMonitoringSystem66/models/sensor/target_specific_code/i
-IC:/<yourPath>/examples/slexTirePressureMonitoringSystem66/generated_code/sensorCodeGenerationW
...
```

The generated makefile and supplemental compiler command-line arguments instruct the make system utility to compile and link code generated from the model and the externally provided source and header files.

Inspect Generated C Code for Sensor

In the **Embedded Coder** app, you can inspect the entry-point functions containing your custom code and the sensor core algorithm in the generated code.

Under the **C Code** tab, open the **Code** pane to view the generated code.

Inspect the initialization function `sensorCodeGenerationWrapper_initialize` defined in `sensorCodeGenerationWrapper.c`. The initialization function:

- Initializes memory pools and initial states used in connection with the referenced sensor core algorithm model
- Calls custom code in `sensor_read_nvm`, which initializes the parameter argument `LOCAL_DEVICE_ID` with the sensor ID `SENSOR_PARAMETERS_NVM_ADDRESS_AS_UINT32` read from NVM

```
/* Model initialize function */
void sensorCodeGenerationWrapper_initialize(void)
{

   /* states (dwork) */
   rtDW.sensorCoreAlgorithmModelReference_InstanceData.rtdw.sensorMeasurmentMessageOutputPort_Se
     = Receive_SendData;
   rtDW.sensorCoreAlgorithmModelReference_InstanceData.rtdw.sensorMeasurmentMessageOutputPort_Se
     = (NULL);

   /* Assign pointer for instance parameters, Block: '<Root>/sensorCoreAlgorithmModelReference' 
   rtDW.sensorCoreAlgorithmModelReference_InstanceData.rtm.sensorCoreAlgorithm_InstP_ref
     = &rtInstP.sensorCodeGenerationWrapperrtsensorCoreAlgorithmModelReference;

   {
     DeviceIDBusType s;
     int32_T i;

     /* Start for slMsgMgr: '<Root>/slMsgMgr' */
     sensorCodeGenerationWrapper_initMemPool
       (&rtDW.slMsgMgr_MemPool_MeasurementMsgBusType,
        &rtDW.slMsgMgr_memArray_MeasurementMsgBusType[0],
        &rtDW.slMsgMgr_freeList_MeasurementMsgBusType[0], 2);

     /* Start for Receive: '<Root>/Receive' */
     rtDW.Receive = rtP.Receive_InitialValue;

     /* Outputs for Atomic SubSystem: '<Root>/readNVMInitializeFunction' */
     /* MATLAB Function: '<S1>/readNVMMATLABFunction' */
```

```
      for (i = 0; i < 10; i++) {
        s.device_id[i] = 0U;
      }

      size_t elementSize;
      elementSize = sizeof(s);

      /* ParameterWriter: '<S1>/localDeviceIDParameterWriter' incorporates:
       *  MATLAB Function: '<S1>/readNVMMATLABFunction'
       */
      sensor_read_nvm((void *)rtP.SENSOR_PARAMETERS_NVM_ADDRESS_AS_UINT32,
                      &rtInstP.sensorCodeGenerationWrapperrtsensorCoreAlgorithmModelReference.LOCA
                      elementSize, (size_t)1.0);

      /* End of Outputs for SubSystem: '<Root>/readNVMInitializeFunction' */
    }
  }
```

Inspect the execution step function `sensorCodeGenerationWrapper_step` defined in `sensorCodeGenerationWrapper.c`. The step function:

- Calls custom code in `sensor_read_pressure`, which returns the acquired pressure data and stores it in `real_T rtb_pressure`
- Runs the sensor model core algorithm `sensorCoreAlgorithm` during each execution cycle
- Calls custom code in `sensor_transmit_data()`, which transmits the sensor payload data by using calls specific to your message protocol service

Code

sensorCodeGenerationWrapper.c ▾ sensorCodeGenerationWrapper_step

```
190
191  /* Model step function */
192  void sensorCodeGenerationWrapper_step(void)
193  {
194    size_t elementSize;
195    real_T rtb_pressure;
196
197    /* MATLAB Function: '<Root>/ssensorReadPressureMATLABFunction' */
198    rtb_pressure = sensor_read_pressure();
199
200    /* ModelReference generated from: '<Root>/sensorCoreAlgorithmModelReference' */
201    sensorCoreAlgorithm(&(rtDW.sensorCoreAlgorithmModelReference_InstanceData.rtm),
202                        &rtb_pressure,
203                        &(rtDW.sensorCoreAlgorithmModelReference_InstanceData.rtdw));
204
205    /* Receive: '<Root>/Receive' */
206    (void)Receive_RecvData(&rtDW.Receive);
207
208    /* MATLAB Function: '<Root>/sensorTransmitMATLABFunction' incorporates:
209     *  Receive: '<Root>/Receive'
210     */
211    elementSize = sizeof(rtDW.Receive);
212    sensor_transmit_data(&rtDW.Receive, elementSize, (size_t)1.0);
213  }
```

Inspect the sensor core algorithm `sensorCoreAlgorithm` defined in `sensorCoreAlgorithm.c`. The sensor core algorithm constructs a message from the unique sensor ID, `LOCAL_DEVICE_ID` , and

the data acquisition payload, `real_T rtb_pressure`, as `MeasurementMsgBusType` structure elements for transmission.

Note that `sensorCoreAlgorithm.c` is generated from the shared reference model. You can double-click the `sensorCoreAlgorithmModelReference` block in the sensor wrapper model to open `sensorCoreAlgorithm.c` in the **Code** pane.

See Also

`coder.ceval` | `coder.ExternalDependency` | `coder.ExternalDependency.updateBuildInfo` | Parameter Writer | Send | Receive | Queue

Related Examples

- "Simulink Messages Overview"
- "Establish Message Send and Receive Interfaces Between Software Components"
- "Model Message-Based Communication Integrated with POSIX Message Queues" on page 3-47
- "Generate Code for Wireless Sensor in Tire Pressure Monitoring Messaging System" on page 3-54
- "Wireless Tire Pressure Monitoring System with Fault Logging"
- "Prepare Sensor and Controller Models in a Distributed Monitoring System for Code Generation"
- "Develop Interface for External C/C++ Code"

Client-Server Communication Interfaces

Applications and components in a platform environment can communicate by using client-server communication. To model client-server communication, use the Simulink Function and Function Caller blocks. A Simulink Function block represents a shared resource. You define the logic as a resource in a Simulink Function block, which separates the function interface (name and arguments) from the logic implementation. Function callers (Function Caller blocks, MATLAB Function blocks, and Stateflow charts) can use the function logic at different levels of the model hierarchy.

The Simulink Function block shares states between function callers. The code generator produces one function. If the Simulink Function block contains blocks that have states, such as a delay or memory, the states persist between function callers.

“Client-Server C Code Commuication” on page 3-65 shows how to use the Simulink Function and Function Caller blocks to generate client-service communication C code. “Client-Server C++ Code Communication” on page 3-70 shows how to use the blocks to generate client-server communication C++ code. For more information about using the Simulink Function and Function Caller blocks, see “Simulink Function Blocks and Code Generation”.

Client-Server C Code Commuication

This example shows how to use the Simulink Function and Function Caller blocks to generate client-server communication code. The code generator produces a local server function and a global server function that complies with code requirements so that existing external code can call the function. The code generator also produces calls to the server functions. The call to the global server function shows that the global function is reused (shared).

Code requirements for the global function are:

- Function names start with prefix `func_`.
- Names of input arguments are of the form `x`*n*, where *n* is a unique integer value.
- Names of output arguments are of the form `y`*n*, where *n* is a unique integer value.
- Input and output arguments are integers (`int`) and are passed by reference. The native integer size of the target hardware is 32 bits.

You create a function that calls the generated server function code. Then, you construct and configure a model to match the code requirements.

Inspect External Code That Calls Server Functions

Open and inspect the example files `call_times2.h` and `call_times2.c`. After you inspect the files, close them.

call_times2.h

```
edit("call_times2.h");
```

```
typedef int my_int;
```

call_times2.c

```
edit("call_times2.c");
```

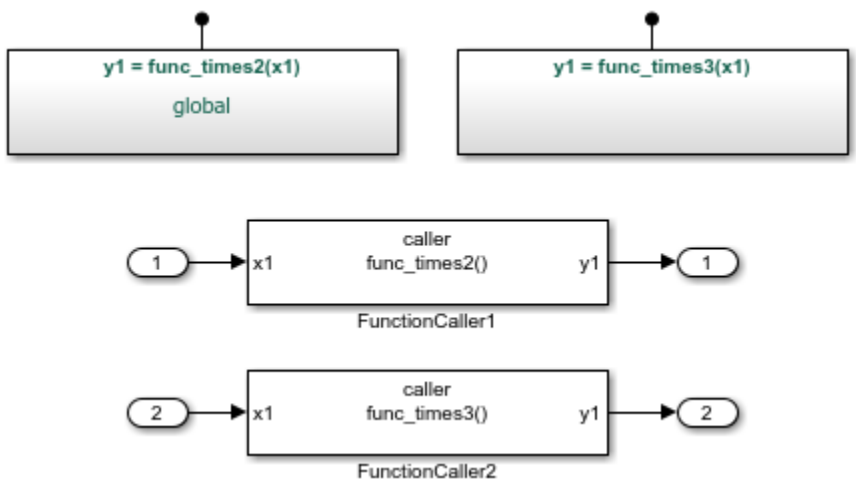
```
#include "call_times2.h"

void call_times2(void)
{
  int times2result;

  func_times2(x1, &y1);

  printf('Times 2 Value:', y1);
}
```

This C code calls global server function `func_times2`. The function multiplies an integer input value `x1` by 2 and returns the result as `y1`.

Create Model

Open the example model `ClientServerSimulinkFunctions`. The model includes two Simulink functions modeled as Simulink Function blocks, `func_times2` and `func_times3`, and a call to each function. As indicated in the model, the visibility of the Simulink Function block for `func_times2` is set to `global`. That visibility setting makes the function code accessible to other code, including external code that you want to integrate with the generated code. The visibility for `func_times3` is set to `scoped`.

```
open_system("ClientServerSimulinkFunctions.slx");
```



If you set model configuration parameter **File packaging format** to `Modular`, the code generator produces function code for the model.

For more information, see “Generate Code for Simulink Function and Function Caller” on page 8-9.

Configure Generated Code to Reuse Custom Data Type

The example assumes that the generated code runs on target hardware using a native integer size of 32 bits. The external code represents integers by using data type `my_int`, which is an alias of `int`. Configure the code generator to use `my_int` in place of the data type that the code generator uses by default, which is `int32_T`.

1. Create a `Simulink.AliasType` object to represent the custom data type `my_int`.

```
my_int = Simulink.AliasType;
```

2. Set the alias type properties. Enter a description, set the scope to `Imported`, specify the header file that includes the type definition, and associate the alias type with the Simulink base type `int32`.

```
my_int.Description='Custom 32-bit int representation';
my_int.DataScope='Imported';
my_int.HeaderFile='call_times2.h';
my_int.BaseType='int32';
```

```
my_int
```

```
my_int =
  AliasType with properties:

    Description: 'Custom 32-bit int representation'
      DataScope: 'Imported'
     HeaderFile: 'call_times2.h'
       BaseType: 'int32'
```

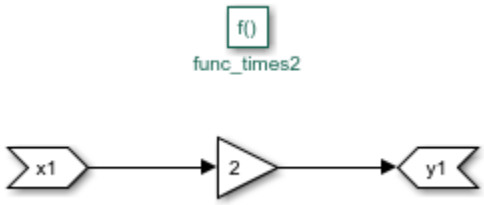
3. Configure the code generator to replace instances of type `int32_T` with `my_int`. In the Configuration Parameters dialog box, open the **Code Generation > Data Type Replacement** pane.

- Select **Replace data type names in the generated code**.
- In the **Data type names** table, enter `my_int` for the replacement name for `int32`.

Configure Server Functions

For external code to call a global server function, configure the corresponding Simulink Function block to have global visibility and an interface that matches what is expected by the external callers.

1.Open the block that represents the `times2` function.



2. Configure the function name and visibility by setting Trigger Port block parameters. For example, the code requirements specify that the function name start with the prefix `func_`.

- Set **Function name** to `func_times2`.
- Set **Function visibility** to `global`.

3. Configure the function input and output arguments by setting Argument Inport and Argument Outport block parameters. For example, the code requirements specify that argument names be of the form `x`*n* and `y`*n*. The requirements also specify that arguments be type `my_int`.

- On the **Main** tab, set **Argument name** to `x1` (input) and `y1` (output).
- On the **Signal Attributes** tab, set **Data type** to `int32`. With the data type replacement that you specified previously, `int32` appears in the generated code as `myint`.

4. Configure the Simulink Function block code interface. At the top level of the model, right-click the block representing global function `func_times2`. From the context menu, select **C/C++ Code > Configure C/C++ Function Interface**.

- Set **C/C++ function name** to `func_times2`.
- Set **C/C++ return argument** to `void`.
- Set **C/C++ Identifier Name** for argument `x1` to `x1`.
- Set **C/C++ Identifier Name** for argument `y1` to `y1`.

If the dialog box lists output argument `y1` before input argument `x1`, reorder the arguments by dragging the row for `x1` above the row for `y1`.

Configure Local Server Function

Configure a Simulink Function block that represents a local server function to use scoped visibility. Based on code requirements, you might also have to configure the function name, input and output argument names and types, and the function interface.

1.Open the block that represents the times3 function.

2. Configure the function name and visibility by setting Trigger Port block parameters. For example, the code requirements specify that the function name start with the prefix func_.

- Set **Function name** to `func_times3`.
- Set **Function visibility** to `scoped`.

3. Configure the function input and output arguments by setting Argument Inport and Argument Outport block parameters. For example, the code requirements specify that argument names be of the form `x`*n* and `y`*n*. The requirements also specify that arguments be type `my_int`.

- On the **Main** tab, set **Argument name** to `x1` (input) and `y1` (output).
- On the **Signal Attributes** tab, set **Data type** to `int32`. With the data type replacement that you specified previously, `int32` appears in the generated code as `my_int`.

4. Configure the Simulink Function block code interface. At the top level of the model, right-click the scoped function `func_times3`. From the menu, select **C/C++ Code > Configure C/C++ Function Interface**. In this case, the code generator controls naming the function and arguments. The function name combines the name of the root model and the function name that you specify for the trigger port of the Simulink Function block. In this example, the name is `ex_slfunc_comp_func_times3`.

You can set **C/C++ return argument** to the argument name that you specify for the Argument Outport block or `void`. For this example, set it to `void`.

For arguments, the code generator prepends `rtu_` (input) or `rty_` (output) to the argument name that you specify for the Argument Inport block.

If the dialog box lists output argument `y1` before input argument `x1`, reorder the arguments by dragging the row for `x1` above the row for `y1`.

Configure Function Callers

For each function caller, configure Function Caller block parameters:

- Set **Function prototype** to `y1 = func_times2(x1)`.
- Set **Input argument specifications** to `int32(1)`.
- Set **Output argument specifications** to `int32(1)`.

Generate and Inspect Code

Generate code for the model.

Global Server Function Code

Source code for global server function `func_times2` is in the build folder in subsystem file, `func_times2.c`.

```
#include "func_times2.h"

/* Include model header file for global data */
#include "ClientServerSimulinkFunctions.h"
#include "ClientServerSimulinkFunctions_private.h"

void func_times2(my_int x1, my_int *y1)
{
  *y1 = x1 << 1;
}
```

Local Server Function Code

The code generator places the definition for the local (scoped) function `func_times3` in file build folder in file `ClientServerSimulinkFunctions.c`.

```
void ClientServerSimulinkFunctions_times3(my_int rtu_x1, my_int *rty_y1)
{
  *rty_y1 = 3 * rtu_x1;
}
```

Calls to Generated Functions

The model execution (step) function, in model file `ClientServerSimulinkFunctions.c`, calls the two Simulink functions: global function `func_times2` and local function `ClientServerSimulinkFunctions_times3`. The name `ClientServerSimulinkFunctions_times3` reflects the scope of the local function by combining the name of the model and the name of the function.

```
void ClientServerSimulinkFunctions_step(void)
{
  my_int rtb_FunctionCaller2;

  func_times2(ClientServerSimulinkFunctions_U.In1, &rtb_FunctionCaller2);

  ClientServerSimulinkFunctions_Y.Out1 = rtb_FunctionCaller2;

  ClientServerSimulinkFunctions_func_times3(ex_slfunc_comp_U.In2, &rtb_FunctionCaller2);

  ClientServerSimulinkFunctions_Y.Out2 = rtb_functionCaller2;
  .
  .
  .
```

Entry-Point Declaration for Local Server Function

The model header file `ClientServerSimulinkFunctions`.h includes an `extern` declaration for function `ClientServerSimulinkFunctions`_times3. That statement declares the function entry point.

```
extern void ClientServerSimulinkFunctions_times3(my_int rtu_x1, my_int *rty_y1);
```

Include Statements for Global Server Function

The model header file `ClientServerSimulinkFunctions`.h lists include statements for the global function `func_times2`.

```
#include "func_times2_private.h"
#include "func_times2.h"
```

Local Macros and Data for Global Server Function

The subsystem header file `func_times2_private.h` defines macros and includes header files that declare data and functions for the global server function, `func_times2`.

```
#ifndef RTW_HEADER_func_times2_private_h_
#define RTW_HEADER_func_times2_private_h_
#ifndef ClientServerSimulinkFunctions_COMMON_INCLUDES_
#define ClientServerSimulinkFunctions_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif
#endif
```

Entry-Point Declaration for Global Server Function

The shared header file `func_times2.h`, in the shared utilities folder `slprj/`*`stf`*`/_sharedutils`, lists shared type includes for `rtwtypes.h`. The file also includes an `extern` declaration for the global server function, `func_times2`. That statement declares the function entry point.

```
#ifndef RTW_HEADER_func_times2_
#define RTW_HEADER_func_times2_

#include "rtwtypes.h"

extern void func_times2(my_int rtu_x1, my_int *rty_y1);

#endif
```

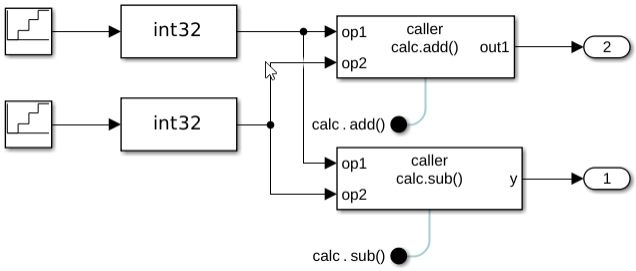
Client-Server C++ Code Communication

You can generate C++ code to support client-server communication by generating abstract classes for modeled client and server components. This type of code generation is supported for models that use scoped function ports as described in “Call Simulink Functions in Other Models Using Function Ports”. After code generation, you must then handwrite code to implement the generated classes. The basic workflow is the following:

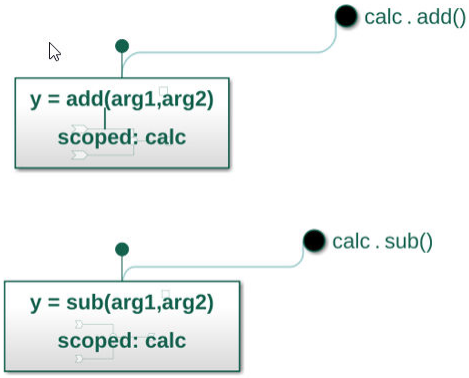
1. Create your own main file. In the program, create concrete methods from the generated abstract pure virtual methods shown in the example as `add` and `sub`.
2. Implement the class functions `add` and `sub` to call into your selected middleware to publish and subscribe data.

3. Create an instance of your implemented client and server scoped function port classes.
4. Create an instance of the model class by using the instances of each scoped function port class as arguments in the model constructor.

For example, the following modelled client requests the services of `add` and `sub`:



The corresponding modelled server provides these services:



The resulting generated code, as shown in the header files, is the following:

Generated Client-Server Code

Client Code

```
// Class declaration for model calc_client
class calc_client final
{
  // public data and function members
 public:
  // External inputs (root inport signals with default storage)
  struct ExtU_calc_client_T {
    real_T calc_add_p;                // '<Root>/calc_add'
    real_T calc_sub_p;                // '<Root>/calc_sub'
    real_T Input;                     // '<Root>/Input'
    real_T Input1;                    // '<Root>/Input1'
    real_T Input2;                    // '<Root>/Input2'
    real_T Input3;                    // '<Root>/Input3'
  };

  // External outputs (root outports fed by signals with default storage)
  struct ExtY_calc_client_T {
    real_T Outport;                   // '<Root>/Outport'
    real_T Outport1;                  // '<Root>/Outport1'
  };

  // Real-time Model Data Structure
  struct RT_MODEL_calc_client_T {
    const char_T * volatile errorStatus;
  };

 ...

  // Real-Time Model get method
  calc_client::RT_MODEL_calc_client_T * getRTM();

  // Constructor
  calc_client(calcT &calc_arg);

  // Root inports set method
  void setExternalInputs(const ExtU_calc_client_T *pExtU_calc_client_T)
  {
    calc_client_U = *pExtU_calc_client_T;
  }

  // Root outports get method
  const ExtY_calc_client_T &getExternalOutputs() const
  {
    return calc_client_Y;
  }

  // model initialize function
  static void initialize();

  // model step function
  void doAdd();

  // model step function
  void doSub();

  // model terminate function
  static void terminate();
...
  // private data and function members
 private:
  // External inputs
  ExtU_calc_client_T calc_client_U;

  // External outputs
  ExtY_calc_client_T calc_client_Y;
  calcT &calc;

  // Real-Time Model
```

Server Code

```
// Class declaration for model calc_server
// Forward declaration
class calc_server;
class calc_servercalcT : public calcT
{
  // public data and function members
 public:
  calc_servercalcT(calc_server &aProvider);
  virtual void add(real_T u2, real_T u1, real_T *y);
  virtual void sub(real_T u2, real_T u1, real_T *y);

  // private data and function members
 private:
  calc_server &calc_server_mProvider;
};

class calc_server final
{
 public:
  // Service port get method
  calcT & get_calc();
...
 private:
  calc_servercalcT calc;
...
};
```

Client Code	Server Code
`  RT_MODEL_calc_client_T calc_client_M;` `};`	

To implement these abstract classes, you must handwrite the code. The following is a POSIX implementation example:

Implemented Client-Server Code

Client Code	Server Code

`class mCalcModelClasscalcT : public calcT{`
`public:`
`    mCalcModelClasscalcT() {`
`        clnt = clnt_create (host, COMPUTE, COMPUTE_VERS, "udp");`
`        if (clnt == NULL) {`
`            clnt_pcreateerror (host);`
`            exit (1);`
`        }`
`    }`
`    void add( int32_T arg0, int32_T arg1, int32_T* arg2)`
`    {`
`        int32_T *result_1;`
`        operands add_6_arg;`
`        // Do data marshaling into struct`
`        add_6_arg.num1 = arg0;`
`        add_6_arg.num2 = arg1;`
`        result_1 = add_6(&add_6_arg, clnt);`
`        if (result_1 == (int *) NULL) {`
`            clnt_perror (clnt, "call failed");`
`        }`
`        // Marshal the result`
`        *arg2 = *result_1;`
`    }`
`    void sub( int32_T arg0, int32_T arg1, int32_T* arg2)`
`    {`
`        int32_T *result_2;`
`        operands sub_6_arg;`
`        sub_6_arg.num1 = arg0;`
`        sub_6_arg.num2 = arg1;`
`        result_2 = sub_6(&sub_6_arg, clnt);`
`        if (result_2 == (int *) NULL) {`
`            clnt_perror (clnt, "call failed");`
`        }`
`        *arg2 = *result_2;`
`    }`
`private:`
`    CLIENT *clnt;`
`};`
 	
`static mCalcModelClasscalcT calc_arg;`
`static mCalcModelClass mCalc_Obj( calc_arg);// Instance of model class`
	`static mServer mCalcSvr_Obj;`
	
	`int *`
	`add_6_svc(operands *argp, struct svc_req *rqstp)`
	`{`
	`    static int  result;`
	`    mCalcSvr_Obj.get_calc().add(argp->num1, argp->num`
	`    return &result;`
	`}`
	`int *`
	`sub_6_svc(operands *argp, struct svc_req *rqstp)`
	`{`
	`    static int  result;`
	`    mCalcSvr_Obj.get_calc().sub(argp->num1, argp->num`
	`    return &result;`
	`}`

Client-server communication uses the same modeling pattern as messages by generating an abstract interface that can be elaborated on for each scoped function service port.

See Also

Simulink Function | Function Caller

Related Examples

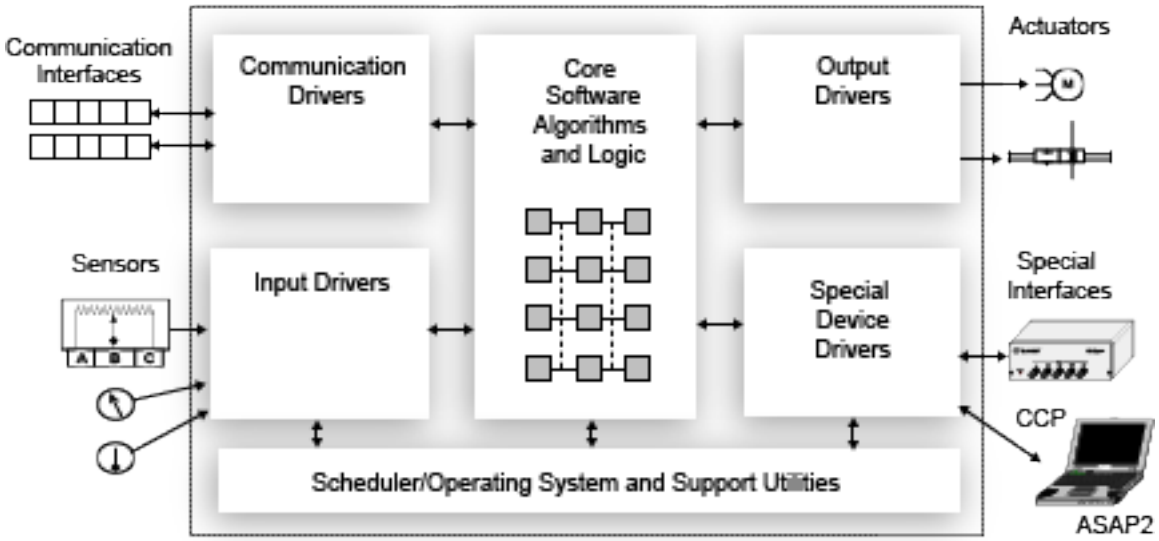
- “Simulink Functions Overview”
- “Simulink Function Blocks and Code Generation” on page 8-6
- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2

Device Driver I/O Interfaces

Device drivers interact with hardware directly. They depend on characteristics of the hardware. For example, a device driver for an analog-to-digital converter initializes, reads data from, and writes data to hardware registers. Hardware differences and dependencies concern data type size, endianess, shift operations, compiler directives, and optimized function and operator support. Other application code interfaces with device drivers by using a programming interface and data mapped to specific memory addresses. Typically, simulation on a development computer is not possible. Reading from and writing to a register during simulation on a development computer produces unexpected and unwanted results.

Device drivers for communication protocols and target hardware are essential to real-time development projects. For example, you can have a working device driver that you want to integrate with algorithmic code that reads data from and writes data to an I/O device that the driver supports. The code generator can produce a single set of application source files from an algorithm model and integrated driver code.

This figure shows an application that requires integration of existing driver code for hardware devices. The core software algorithms and logic can be a combination of code modules for external reusable algorithms integrated into the Simulink environment and code generated as part of an overall model design.



Integrate device driver code in a component model by using one of these methods:

- Import driver function code into a C Caller block.
- Generate an S-function, compile and link the generated S-function, and create a mask for calling the generated S-function by using the Legacy Code Tool.

See Also

Related Examples

- “Integrate C Code Using C Caller Blocks”
- “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6

Multi-Instantiation

You can model multi-instance rate-based component models in Simulink. For example, you can:

- Configure a model as a multi-instance component by setting model configuration parameter Code interface packaging to `Reusable function`.
- Generate code with reentrant functions and multi-instance calls to platform services. You can access external I/O, tunable parameters, and per-instance memory, and use reusable subsystems in multi-instance mode.
- Verify multi-instance code by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

See Also

Related Examples

- “Component-Based Modeling Guidelines”
- “Choose a Componentization Technique for Code Reuse” on page 8-4

Model Single-Core, Rate-Monotonic Multitasking Application

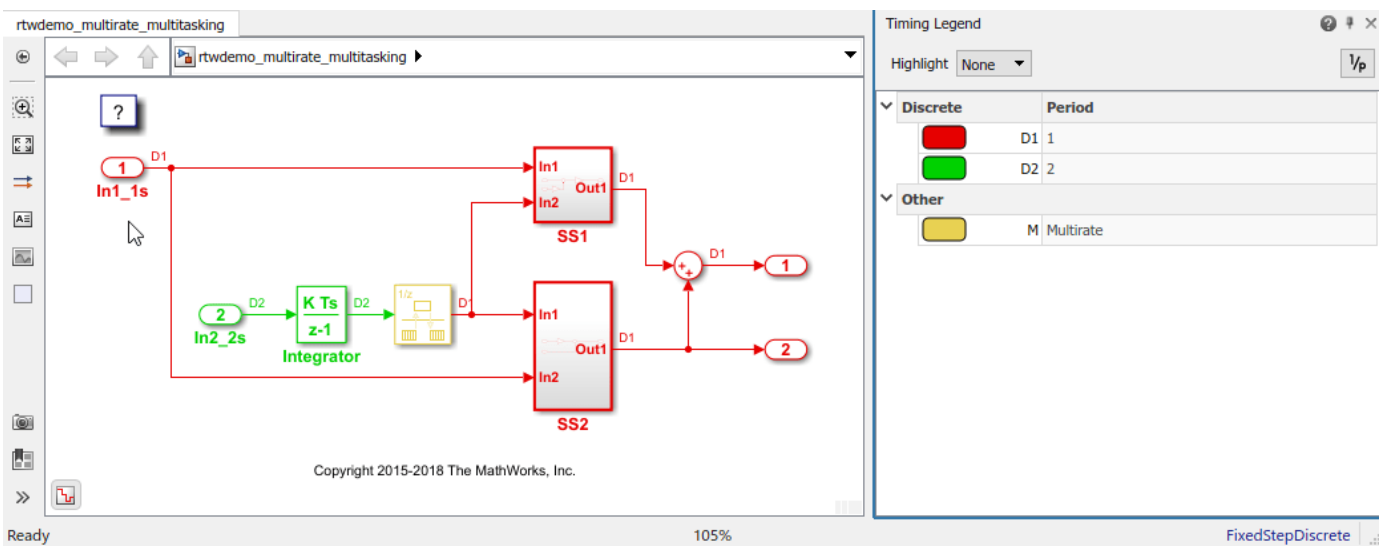
This example shows a model designed and configured for generating embedded system code intended to execute as an application in a single-core, multitasking target environment. The application algorithm is captured as a multirate top model that is configured for C data interface code generation. The model simulates and the generated code executes based on the model configuration and a rate monotonic scheduling algorithm.

Periodic Multirate Model Set Up for Multitasking Execution

Open the example model.

```
open_system('MultirateMultitasking');
```

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.
- To provide a clear partitioning of rates, sample times for subsystems `SS1` and `SS2` are set to 1.
- The Rate Transition block models an explicit rate transition. Alternatively, instruct Simulink to insert Rate Transition blocks for you by selecting model configuration parameter **Automatically handle rate transition for data transfer**.

Relevant Model Configuration Parameter Settings

- **Type** set to `Fixed-step`.
- **Solver** set to `discrete (no continuous states)`.
- **Treat each discrete rate as a separate task** selected.

Scheduling

Simulink® simulates the model based on the model configuration. Code that this model generates implements the same execution semantics. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, multitasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate. Yellow represents the mixture of the two rates.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks that run at the different rates.

Benefits of implicit rate grouping:

- Simulink does not impose architectural constraints on the model. Create a model without imposing software architecture constraints within the model.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, the model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates based on multitasking execution semantics.

Simulink enforces data transfer constraints to achieve rate-monotonic scheduling:

- Data transfers occur between a single read task and a single write task.
- When data transfers between two tasks, only one task can preempt the other task.
- For periodic tasks, a task with a faster rate has a higher priority than a task with a slower rate. In addition, a task with the faster rate, preempts a task with a slower rate.
- Tasks run on a single processor.
- Time slicing, use of a defined time period during which a task can run in a preemptive multitasking system, is not allowed.
- Processes do not crash and restart, especially during data transfers between tasks.
- Read and write operations on byte-sized variables are atomic.

Your execution framework communicates with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Open the **Embedded Coder** app. Then, generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point functions `MultirateMultitasking_step0` and `MultirateMultitasking_step1`. Use this file as a starting point for coding your execution framework.
- `MultirateMultitasking.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `MultirateMultitasking.h` declares model data structures and a public interface to the model entry points and data structures.

- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include MultirateMultitasking.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.In1_1s` of data type `real_T` with dimension of 1
- `rtU.In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void MultirateMultitasking_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void MultirateMultitasking_step0(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function, `void MultirateMultitasking_step1(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every two seconds. To achieve real-time execution, attach this function to a timer.

Output ports:

- `rtY.Out1` of data type `real_T` with dimension of 1
- `rtY.Out2` of data type `real_T` with dimension of 1

More About

- “Modeling for Multitasking Execution”
- “Deploy Applications to Target Hardware” on page 71-2
- “Customize Code Organization and Format” on page 55-46

Model Multicore Concurrent Tasking Application

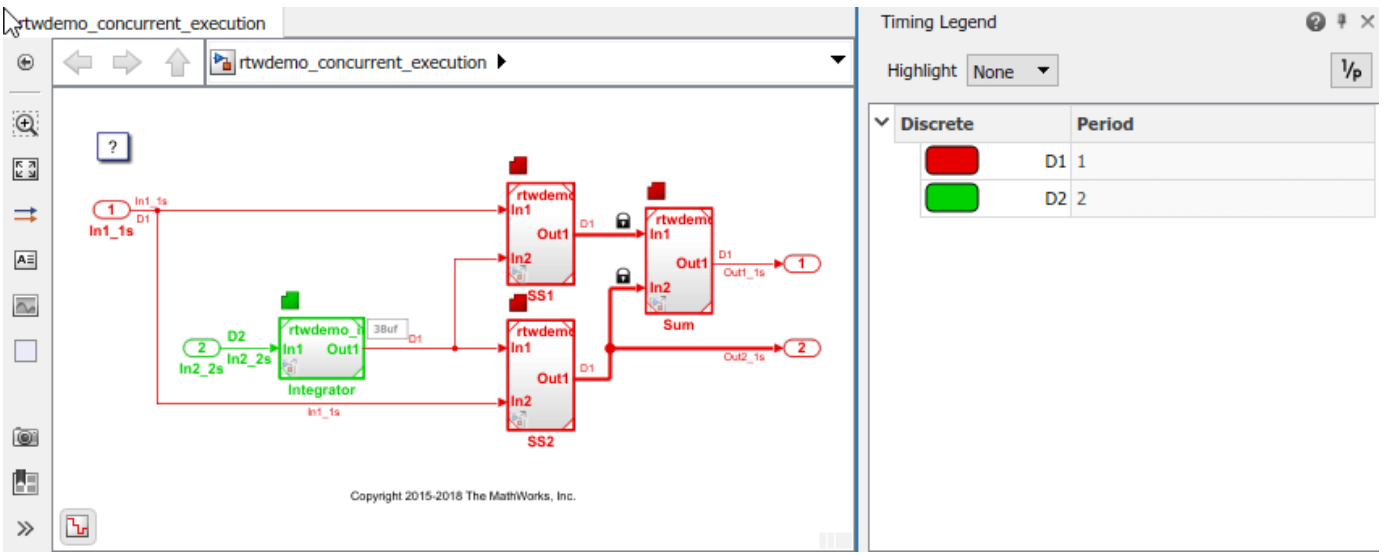
This example shows a model designed and configured for generating embedded system code intended to execute as an application in a symmetric multicore, multitasking target environment. The application algorithm is captured as a multirate top model that is configured for C data interface code generation.

Periodic Multirate Model Set Up for Multitasking Concurrent Execution

Open the example model.

```
open_system('MulticoreConcurrentTasking');
```

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



Simulink supports simulating concurrent task execution by assigning partitions of a model to tasks that you designate to run concurrently on multicore hardware. Use an implicit or explicit approach to designating partitions.

Simulink implicit partitioning:

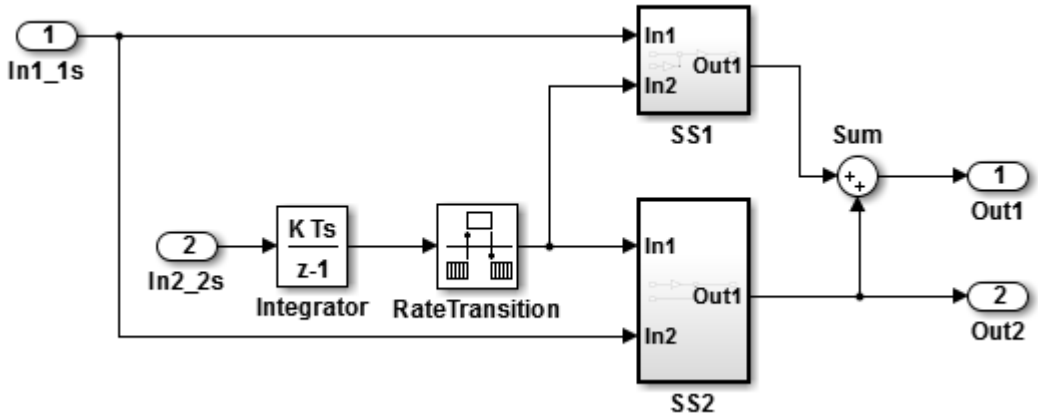
- Partitions the model based on sample times specified in the model.
- Assigns a task to each sample rate and designates that the tasks run concurrently.
- Controls the granularity of partitions. For example, you cannot split a sample rate into multiple tasks.
- Does not impose modeling constraints.
- Provides ready-to-use hardware solutions, such as solutions that the Simulink® Real-Time™ product produces.
- Is not relevant to standalone production code generation due to the lack of control over partition granularity.

Explicit partitioning:

- Use Model and Subsystem blocks to partition the model.
- Create an arbitrary number of tasks.
- Simulink assigns each partition to a task.
- Simulink imposes modeling constraints.
- Control the granularity of partitions.
- Split a sample rate into multiple tasks.
- Assign partitions to different processor cores.
- Is for standalone production code generation due to the level of control you have over granularity of partitions.

This example shows explicit partitioning.

Consider the following periodic multirate model that is set up for multitasking execution.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.
- To provide a clear partitioning of rates, sample times for models `SS1` and `SS2` are set to 1.
- The Rate Transition block explicitly models a rate transition.

To support concurrent execution of tasks in a multicore run-time environment, the preceding model was modified:

- The Integrator block is in a Model block configured with a fixed-step discrete solver and a step size of two seconds.
- Subsystems `SS1` and `SS2` were converted to Model blocks configured with a fixed-step discrete solver and a step size of one second.
- The Sum block is in a Model block configured with a fixed-step discrete solver and a step size of one second. Another option for the Sum block is to place it in `SS1` or `SS2` and compute its value coincident with the Model block. For concurrent execution of tasks, only connection blocks, Model blocks, and Subsystem blocks can be at the root level of a model.
- The Rate Transition block was removed.

Relevant Model Configuration Parameter Settings

- **Type** set to `Fixed-step`.

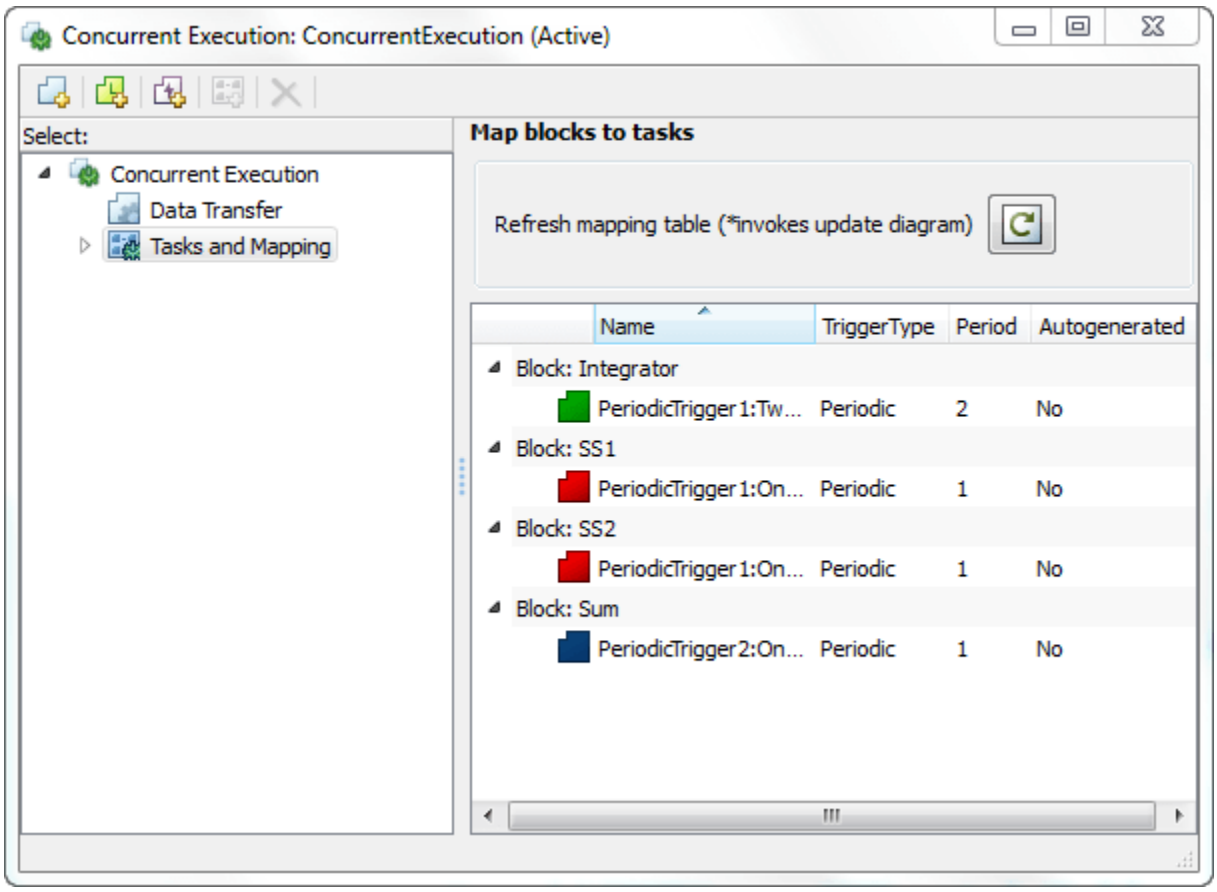
- **Solver** set to `discrete (no continuous states)`.
- **Treat each discrete rate as a separate task** selected.
- **Automatically handle rate transition for data transfer** selected. Necessary because Rate Transition block was removed.
- **Allow tasks to execute concurrently on target** selected.

Concurrent Execution Parameter Settings

Open the Concurrent Execution dialog box by clicking **Configure Tasks** on the Configuration Parameters **Solver** pane. Selecting parameter **Allow tasks to execute concurrently on target** enables the **Configure Tasks** button.

When selected, parameter **Enable explicit model partitioning for concurrent behavior** enables concurrent execution options for the top-level model.

Click **Tasks and Mapping** to review the tasks and mapping.



Simulink creates a default mapping for each partition (Model block) by assigning each partition to a separate task. Simulink designates that each partition executes concurrently and simulates latency effects that data communication between processor cores imposes. This dialog box displays a mapping consisting of partitions spread across two independent periodic triggers: `SS1`, `SS2`, and `Sum` mapped to periodic trigger 1 and `Integrator` mapped to periodic trigger 2.

Scheduling

Simulink® simulates the model based on the model configuration. Code generated from the model implements the same execution semantics. Simulink propagates and uses the Inport block sample times to order block execution based on a multicore, multitasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between the rates.

Benefits of implicit Simulink rate grouping:

- Simulink does not impose architectural constraints on the model. Create a model without imposing software architecture constraints within it.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates, based on multitasking execution semantics.

Simulink enforces data transfer constraints:

- Data transfers occur between a single read task and a single write task.
- Tasks run on a single processor.
- Processes do not stop or restart, especially during data transfers between tasks.
- Read and write operations on byte-sized variables are atomic.

Your execution framework can communicate with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by indirectly calling entry-point functions `PeriodicTrigger1_OneSecond_step`, `PeriodicTrigger1_TwoSecond_step`, and `PeriodicTrigger2_OneSecond_step` with the function `MulticoreConcurrentTasking_step`. Use this file as a starting point for coding your execution framework.
- `MulticoreConcurrentTasking.c` contains entry points for the code that implements the model algorithm. This file includes the rate and task scheduling code.
- `MulticoreConcurrentTasking.h` declares model data structures and a public interface to the model entry points and data structures.
- `model_reference_types.h` contains type definitions for timing bridges. These type definitions are generated for a model reference target or a model containing Model blocks.

- `rtw_windows.h` declares mutex and semaphore function prototypes that the generated code uses for concurrent execution on Microsoft® Windows® platforms.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include MulticoreConcurrentTasking.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `In1_1s` of data type `real_T` with dimension of 1
- `In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void MulticoreConcurrentTasking_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void PeriodicTrigger1_OneSecond_step(void)`. Call this function periodically for one of two tasks that require scheduling at the fastest rate in the model. For this model, call the function every second.
- Output and update entry-point function, `void PeriodicTrigger1_TwoSecond_step(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every two seconds.
- Output and update entry-point function, `void PeriodicTrigger2_OneSecond_step(void)`. Call this function periodically for the second task that requires scheduling at the fastest rate in the model. For this model, call the function every second.

To achieve real-time execution, define a task or thread for each entry-point step function. Trigger execution of each function based on a timer that has the same rate as the given function. The operating system schedules the tasks across cores dynamically or based on your mapping of tasks to cores.

Output ports:

- `Out1_1s` of data type `real_T` with dimension 1
- `Out2_1s` of data type `real_T` with dimension 1

More About

- “Multicore Processor Targets”
- “Deploy Applications to Target Hardware” on page 71-2
- “Customize Code Organization and Format” on page 55-46

Generate Code for Atomic Subsystems

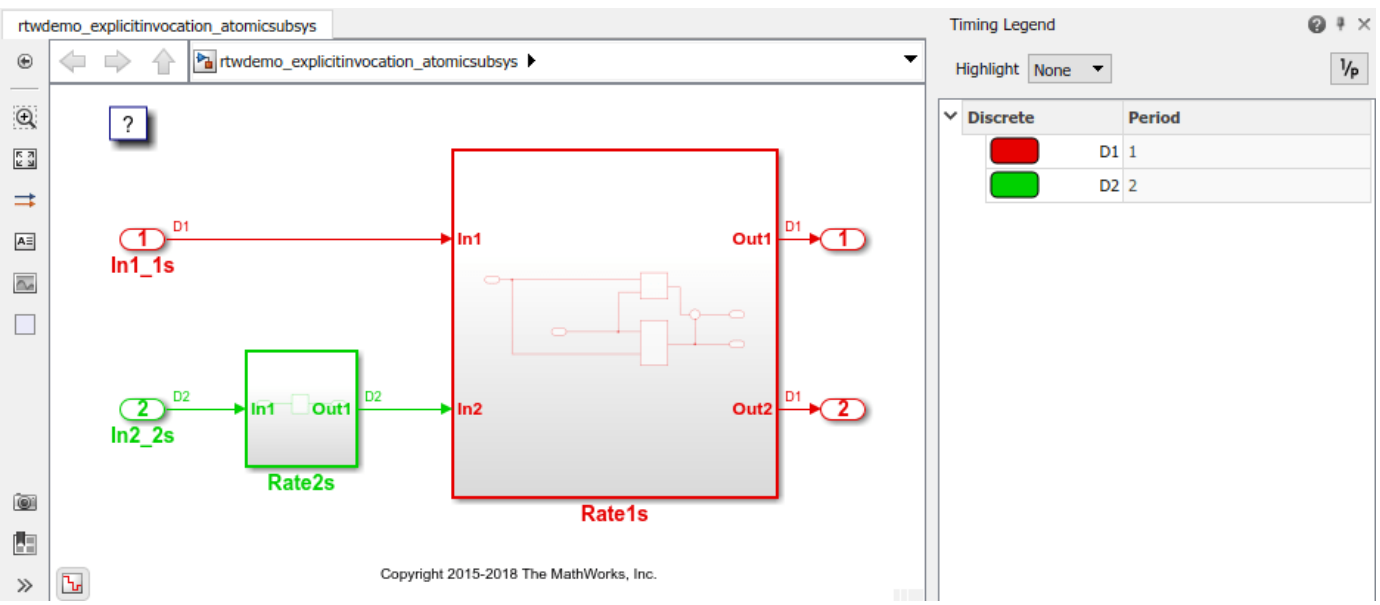
This example shows a model designed and configured for deploying embedded system component code that is generated from a model that is partitioned into multiple atomic subsystems. The subsystems execute at different rates. The multirate top model is configured for C data interface code generation. You integrate the generated source code for each subsystem with other code, including an external scheduler, that runs in a target environment.

Atomic Subsystem Model

Open the example model.

```
open_system("AtomicSubsystem");
```

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



This model partitions an algorithm into two atomic subsystems: `Rate1s` and `Rate2s`. Subsystem `Rate1s` is configured with a sample time of 1 second. Subsystem `Rate2s` is configured with a sample time of 2 seconds.

Relevant Model Configuration Parameter Settings

- **Type** set to `Fixed-step`.
- **Solver** set to `discrete (no continuous states)`.
- **Treat each discrete rate as a separate task** cleared.

Scheduling

Simulink® simulates the model based on the model configuration. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, single-tasking execution platform.

For this example, the sample time legend shows implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

Based on rate-monotonic scheduling, your application code (execution framework) must transfer data between subsystems `Rate2s` and `Rate1s` at a frequency of 2 seconds with the priority of 1 second. That is, the generated function transfers data in the 1 second task every other time prior to executing code for subsystem `Rate1s`.

Your execution framework must schedule the generated function code and handle the data transfers between them. This is an advantage for multirate models because the generated code does not assume scheduling or data transfer semantics. However, the execution framework must handle data transfers explicitly.

Generate Code and Report

Generate a single callable function for each subsystem without connections between them. Multiple ways are available to generate code for a subsystem, including from the subsystem context menu.

1. Open the **Embedded Coder** app.
2. Right-click a subsystem block.
3. Click **C/C++ Code > Build This Subsystem**.

The example model generates a report.

Review Generated Code

Review the generated code in the **Code** view or the code generation report.

- `ert_main.c` is an example main program (execution framework) for the subsystem. This code controls model code execution by calling entry-point function `Rate1s_step` or `Rate2s_step`. Use this file as a starting point for coding your execution framework.
- `Rate1s.c` and `Rate2s.c` contain entry points for the code that implements subsystem `Rate1s` and `Rate2s`, respectively. This file includes the rate and task scheduling code.
- `Rate1s.h` and `Rate2s.h` declare model data structures and a public interface to subsystem entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include AtomicSubsystem.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports, `Rate1s`:

- `rtU.In1` of type `real_T` with dimension of 1
- `rtU.In2` of type `real_T` with dimension of 1

Entry-point functions, `Rate1s`:

- Initialize entry-point function, `void Rate1s_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void Rate1s_step(void)`. Call this function periodically, every second.
- Termination function, `void Rate1s_terminate(void)`. Call this function once from your shutdown code.

Output ports, `Rate1s`:

- `rtY.Out1` of type `real_T` with dimension of 1
- `rtY.Out2` of type `real_T` with dimension of 1

Input ports, `Rate2s`:

- `rtU.In1` of type `real_T` with dimension of 1

Entry-point functions, `Rate2s`:

- Initialize entry-point function, `void Rate2s_initialize(void)`. Call this function once at startup.
- Output and update entry-point (step), `void Rate2s_step(void)`. Call this function periodically, every 2 seconds.
- Termination function, `void Rate2s_terminate(void)`. Call this function once from your shutdown code.

Output ports, `Rate2s`:

- `rtY.Out1` of type `real_T` with dimension of 1

More About

- “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2
- “Deploy Applications to Target Hardware” on page 71-2
- “Customize Code Organization and Format” on page 55-46

Deploy Export-Function Component Configured for C Service Interface Code Generation

This example shows how to develop an export-function model as a component that includes service interface support for interacting with target platform software. In this example you:

1. Assess project requirements.
2. Define the service code interface configuration in the coder dictionary.
3. Design the component model.
4. Configure the component model interfaces for code generation.
5. Assess component model readiness for code generation.
6. Generate and inspect the generated component code.
7. Test the generated service interface code by using a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation.
8. Get meta-information about the generated code interface for integration with target platform software.
9. Integrate generated component code with target environment code.

Assess Project Requirements

This example assumes these component and target platform requirements.

Component Description and Interface Requirements

The component model must:

- Favor memory optimization over data freshness when communicating data.
- Include an initialize function that reads a value from nonvolatile memory and uses that value to initialize a state variable.
- Include an integrator function that applies a forward integration method, applies a gain value of 1.25 to the state variable, and sends its output to an accumulator function. The integrator function gets called aperiodically.
- Include an accumulator function that receives the value of the state variable from the integrator function, increments the value by one, and applies a gain value, which is tunable during execution. The accumulator function gets called periodically, every 1 second and maintains the most recent value of the state variable.
- Include a terminate function that writes the output of the algorithm function to nonvolatile memory after the function stops executing.
- Make state variables accessible for monitoring during execution.

Target Platform Interface Requirements

- To initiate component initialization, the target platform function scheduler expects to call entry-point function `CD_initialize` with no arguments.
- To initiate execution of the aperiodic integrator function, the target platform function scheduler expects to call entry-point function `CD_integrator` with no arguments.
- To initiate execution of the periodic algorithm function, the target platform function scheduler expects to call entry-point function `CD_accumulator` with no arguments.

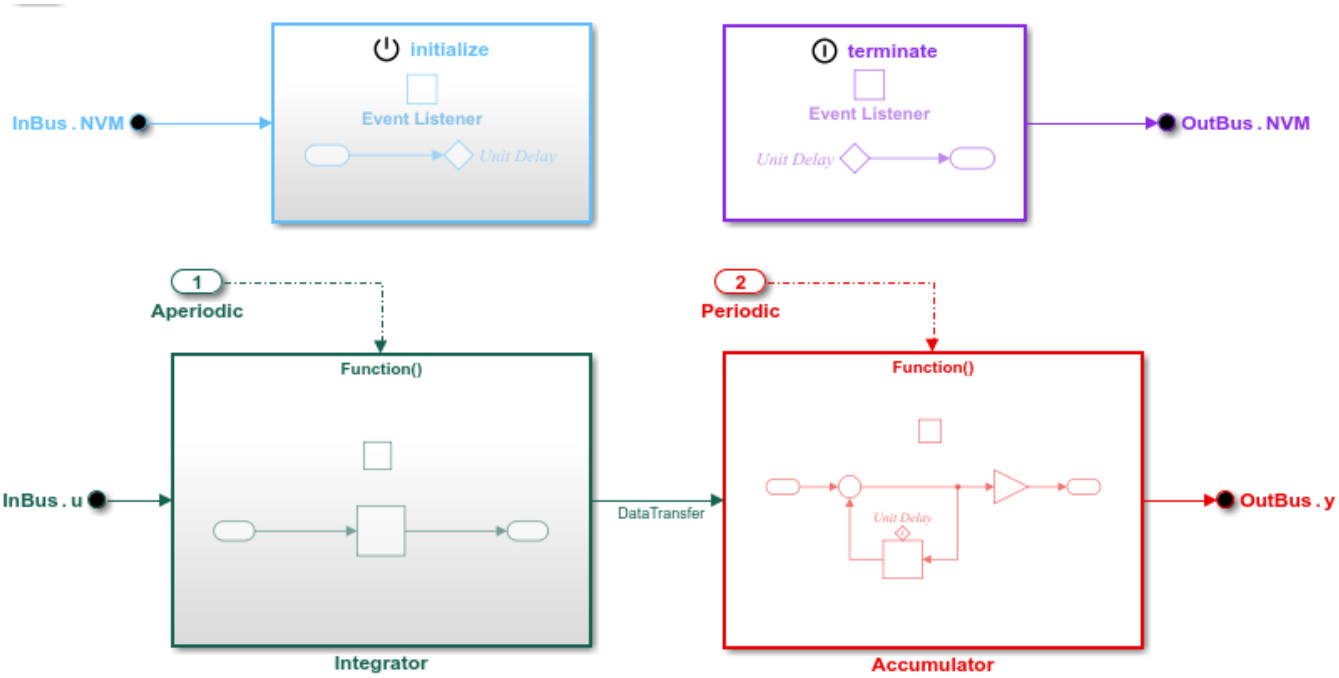
- To complete component termination, the target platform function scheduler expects to call entry-point function `CD_terminate` with no arguments.
- The component reads from and writes to target platform nonvolatile memory.
- The platform receiver service function is named `get_CD_integrator_InBus_u` and uses outside-execution data communication. For more information, see "Data Communication Methods" on page 1-14.
- The platform sender service function is named `getref_CD_accumulator_OutBus_y` and uses outside-execution data communication.
- The platform data transfer service sender function is named `set_CD_integrator_DataTransfer`, the receiver function is named `get_CD_accumulator_DataTransfer`, and the service uses outside-execution data communication.
- The platform timer service uses outside-execution. The get function clock tick service function is named `get_CD_tick_`*`callable-function`*, where *`callable-function`* is `CD_integrator`. The function returns the tick value as a value of type `unint32`.
- The variable for the tunable gain value applied by the accumulator function is named `CD_tunable.k`.
- Variables representing the discrete time integrator and delay state values are named `CD_measured.dti` and `CD_measured.delay`.

Define Service Code Interface Configuration

You address the target platform requirements by defining a service code interface configuration in a shared Embedded Coder Dictionary. For this example, the requirements are defined in the dictionary file `ComponentDeploymentCoderDictionary.sldd`, which is linked to the example model.

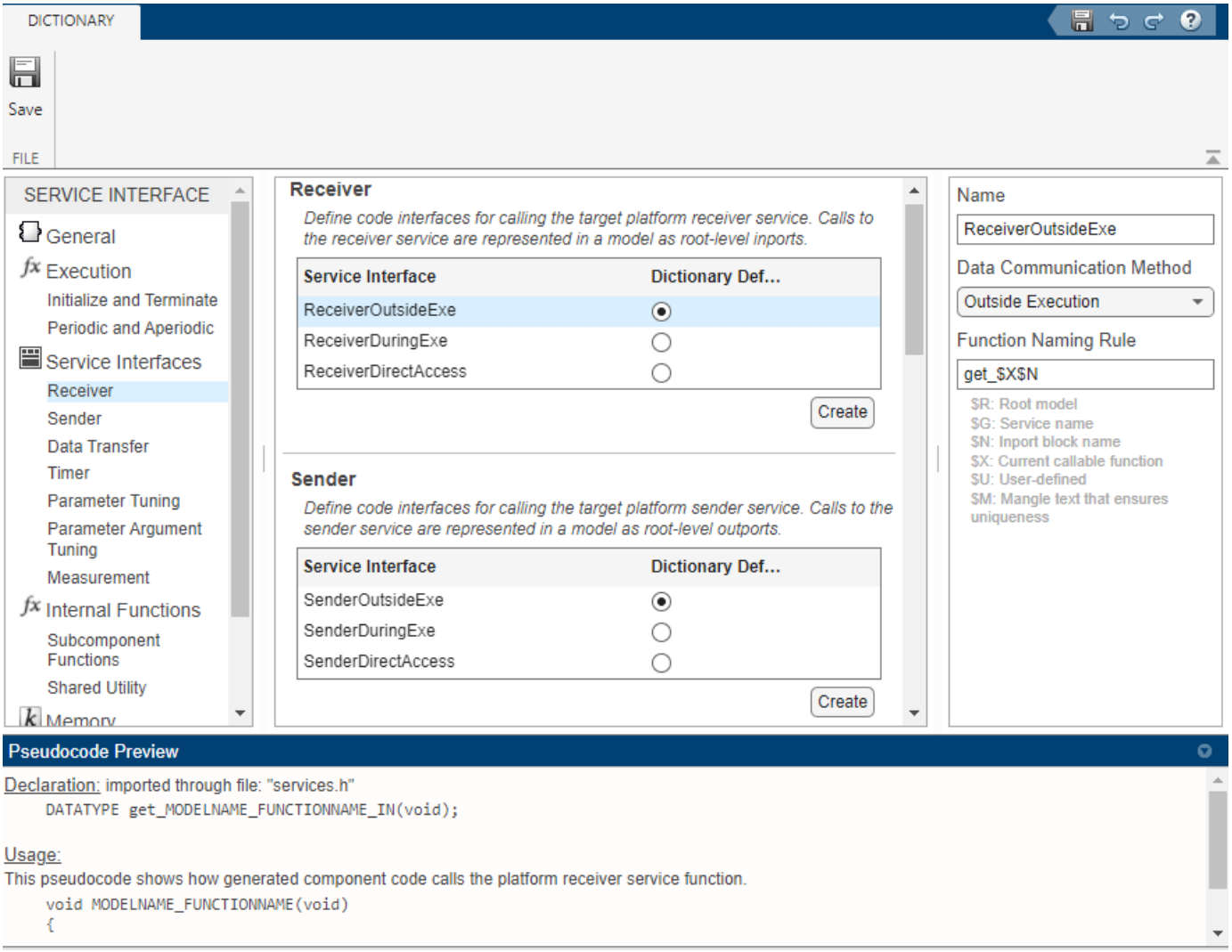
Open example component model `ComponentDeploymentFcn`.

```
open_system('ComponentDeploymentFcn');
```



Open the Embedded Coder Dictionary (`ComponentDeploymentCoderdictionary.sldd`), which is linked to the model.

1. Open the Embedded Coder app.
2. In the toolstrip, under **Code interface**, select `Embedded Coder Dictionary (Shared)`. The Embedded Coder Dictionary linked to the example model opens. You can create a coder dictionary in the Model Explorer (click **File** > **New** > **Embedded Coder Dictionary**) or in the model Configuration Parameters dialog box (click **Set up** to the right of the **Shared coder dictionary** parameter).
3. Under **Service Interface**, select **Receiver**. The Embedded Coder Dictionary shows content in three panes. Use the left pane to navigate through the dictionary configurations. The center pane lists the interfaces that are defined in a code interface configuration for the item you select in the navigation pane. From the center pane you can create and delete interfaces and specify a default interface. The right pane shows the property settings for the interface selected in the center pane. The Pseudocode Preview updates as you change property settings.



In the dictionary, browse through the interface settings. Check that the settings align with the requirements noted in Target Platform Requirements on page 3-89.

Based on requirements listed in Assess Project Requirements on page 3-89, the dictionary defines these relevant interfaces:

- Initialize and terminate callable entry-point function interface `InitTermInterface`, which specifies naming rule `CD_$N`, where `$N` is `initialize` or `terminate`, depending on the type of function generated.
- Periodic and aperiodic callable entry-point function interface `PeriodicAperiodicInterface`, which specifies naming rule `CD_$N`, where `$N` is is the name of the function-call inport block. Later, you override the default naming in the code mappings.
- Receiver service interface `ReceiverOutsideExe` specifies outside-execution data communication. The interface specifies naming rule `get_$X_$N`, where `$X` is the name of the callable function and `$N` is the name of the receiver port for the receive operation.
- Sender service interfaces Sender`OutsideExe` specifies outside-execution data communication. The interface specifies naming rules `set_$X_$N` and `getref_$X_$N` for the by value and by reference functions, where `$X` is the name of the callable function and `$N` is the name of the sender port for the send operation.
- Data transfer service interfaces `DataTransferOutsideExe` and `DataTransferDuringExe`. `DataTransferOutsideExe`, the default, specifies outside-execution data communication. `DataTransferDuringExe` specifies during-execution data communication. Both interfaces specify naming rules `set_$X$N` and `get_$X$N`, where `$X` is the name of the callable function and `$N` is the name of the signal that represents the data transfer.
- Timer service interfaces `TimerService` and `get_CD_tick`. `TimerService`, the default, specifices outside-execution data communication and naming rule `$G_tick_$X`, where `$G` token is the service name `TimerService` and `$X` is the name of the function that calls the service. Interface `get_CD_tick` specifies outside-execution data communication and a naming rule that aligns with the project requirements, `$G$X`, where `$G` is the service interface name and `$X` is the name of the callable function.
- Parameter tuning service interface `ParameterTuningService` applies storage class `TunableStruct` to data elements. To align with the platform requirements for naming tunable parameters, property **Type Name** was set to `CD_tunable_T$M` and property **Instance Name** was set to `CD_tunable`.
- Measurement service interface, `MeasurementService`, applies storage class `MeasurementStruct`. To align with the platform requirements for naming state data, property **Type Name** was set to `CD_measured_T$M` and property **Instance Name** was set to `CD_measured`.

The during-execution receiver, sender, and data transfer service interfaces are defined so that you can explore differences in the generated code for the different data communication.

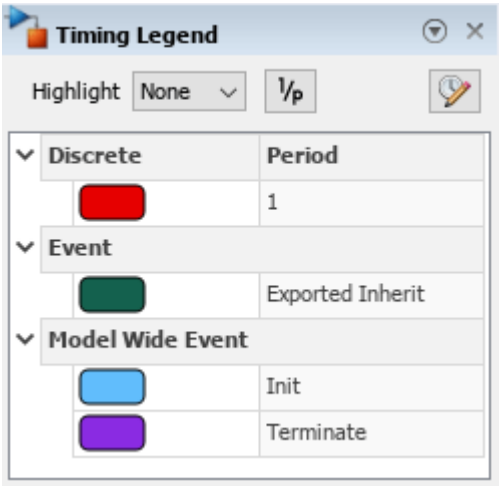
Later, in Configure Component Model Interface for Code Generation on page 3-94, you map elements of the example model to interfaces defined by the service interface configuration in the dictionary.

For more information about defining code interface configurations, see “Code Interface Definitions” and Embedded Coder Dictionary.

Design Component Model

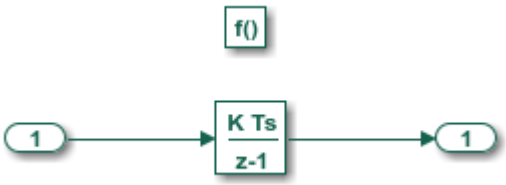
The design of example component model `ComponentDeploymentFcn` reflects the requirements listed in Assess Project Requirements on page 3-89.

The timing legend for the model identifies four functions: two model-wide event-driven functions `Init` and `Term`, a discrete periodic function, and an exported event-driven (aperiodic) function. To open the timing legend, on the **Debug** tab, click **Information Overlays** > **Timing Legend**.



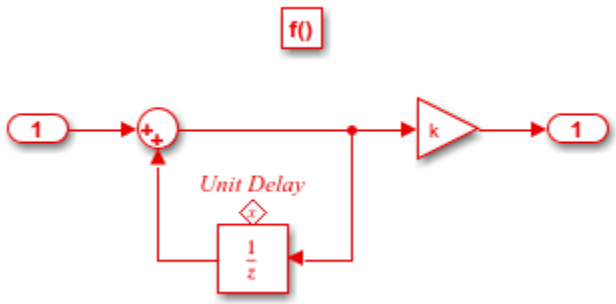
- The Initialize Function block represents a function that initializes a state variable `x` with a value read from nonvolatile memory before the algorithm executes. See “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment” and “cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks”.

- The In Bus Element block `InBus.NVM`, which feeds the initialize function, represents a call to the platform service that reads values from nonvolatile memory. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0405: Data receive for component deployment”.
- Function-call subsystem `Integrator` represents an integrator function. The function applies a forward integration method and a gain value of 1.25 to the value of state variable `x` each time the function executes. See “cgsl_0401: Modeling styles for component deployment”.



- The inport for Integrator, `InBus.u`, represents a call to the platform receiver service for reading the input value from the target environment. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0405: Data receive for component deployment”.

- Function-call subsystem Accumulator represents a function that increments the value of state variable `x` by 1 and applies a tunable gain value `k`. The function executes every second. See "cgsl_0401: Modeling styles for component deployment".



- The signal line that connects the output port of function-call subsystem `Integrator` to the input port of function-call subsystem `Accumulator` represents a data transfer. `Integrator` calls the data transfer service to send data to `Accumulator`. `Accumulator` calls the data transfer service to receive the data from `Integrator`. See "cgsl_0409: Data transfer for component deployment".
- The outport of `Accumulator` represents a call to the platform sender service for writing the output value of the Gain block. See "cgsl_0402: Signal interfaces for component deployment" and "cgsl_0406: Data send for component deployment".
- The Terminate Function block represents the function that writes the output of `Accumulator` to nonvolatile memory after the component stops executing. See "cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment" and "cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks".

- The outport of the terminate function represents a call to the platform service that writes values to nonvolatile memory. See "cgsl_0402: Signal interfaces for component deployment" and "cgsl_0406: Data send for component deployment".

For more information about modeling guidelines that can help you develop and deploy component models that use a service code interface, see "Modeling Guidelines for Generated Code" on page 23-5.

Configure Component Model Interface for Code Generation

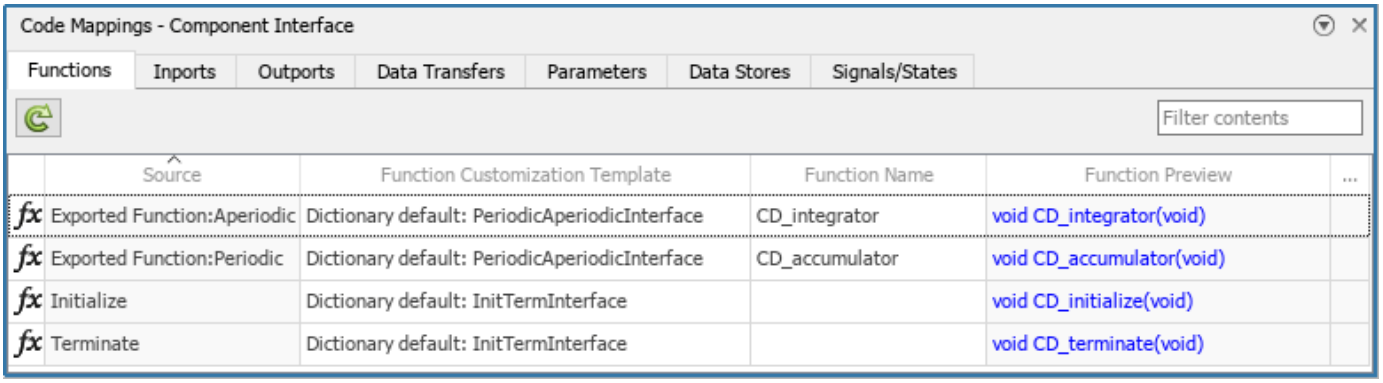
For the code generator to produce interface code that aligns with the target environment requirements, there needs to be a mapping between the interface elements in the model and code interfaces. A model can be associated with one of three types of code interfaces: data interface, service interface, or C++ interface. The code generator determines the type of code interface to apply automatically based on this information:

- Modeling styles and patterns used in the model

- Embedded Coder app **Output** selection
- Type and contents of the Embedded Coder Dictionary associated with the model

For this example, the code generator applies a service interface because the model is linked to shared Embedded Coder Dictionary `ComponentDeploymentCoderDictionary.sldd`, which defines a service interface configuration. The code generator uses the the linked dictionary to establish default mappings that you can view and change by using the Code Mappings Editor or code mappings programming interface.

To open the Code Mappings editor and explore the interface settings, in the toolstrip, under **Code Interface**, select **Component Interface**.



For elements for which the default interface aligns with the requirements, you do not need to make changes. Depending on your model, your code generation goals, and available interfaces, you might need to make some adjustments.

For this example, to meet the code interface requirements listed in Assess Project Requirements on page 3-89, these adjustments were made:

- On the **Functions** tab, the default function customization template for the aperiodic integrator and periodic accumulator functions produces function names `CD_Aperiodic` and `CD_Periodic`. To align with target platform requirements for these functions, the code mappings override the default functions names, by specifying the function names `CD_integrator` and `CD_accumulator`.
- An alternate timer service interface was associated with the integrator function. The coder dictionary defines two timer service interfaces: the default interface and an alternate interface get_CD_tick, which aligns with the target platform timer service requirements for function names. To see the timer service setting, on the **Functions** tab, select the row for `Exported Function:Aperiodic` and click the pencil icon. In the dialog box that appears, see that the **Timer Service** property is set to `get_CD_tick`. Alternatively, you can set the property in the Property Inspector.
- On the **Parameters** tab, a code identifier was specified for the Gain block gain parameter `k`. To see the identifier setting, under **Model Parameters**, select the row for the parameter and click the pencil icon. In the dialog box that appears, see the value specified for the **Identifier** property, k. The identifier informs the code generator how to represent the variable in the generated code. Alternatively, you can set the property in the Property Inspector.
- On the **Signals/States** tab, code identifiers were specified for the states associated with the Discrete-Time Integrator and Unit Delay blocks. To see the identifier settings (`dti` and `delay`), select the row for the state of interest and click the pencil icon. In the dialog box that appears, see

the value specified for the **Identifier** property. The identifiers inform the code generator how represent corresponding variables in the generated code. Alternatively, you can set the property in the Property Inspector.

In the Code Mappings editor, you can change the interfaces for the sender, receiver, data transfer, and timer services. To change the sender, receiver, and data transfer service, click the corresponding tab and, in the service column, select an alternative service. To change the timer service, click the **Functions** tab, select the row for the integrator function, click the pencil icon, and in the dialog tht appears, select an alternative interface. The default interfaces use the outside-execution data communication method. The alternative interfaces use during-execution and direct-access data communication. Consider changing the interfaces to view differences in the generated code. If you choose to change the interfaces, for each service called along the data path of a signal, set the interfaces such that they apply the same data communication method.

For guidance on setting up a model to use a service interface configuration, see “cgsl_0414: Configure service interface for component model”. For more information about model code mappings for service interfaces, see “C Service Interfaces” and Code Mappings Editor – C.

Assess Component Model Readiness for Code Generation

Run the Simulink Model Advisor to confirm that the component model passes component modeling guideline checks.

1. In the Embedded Coder app, select **C/C++ Code Advisor**.
2. In the System Selector dialog box, select `ComponentDeploymentFcn`. Click **Ok**.
3. In the Code Generation Advisor dialog box, in the left pane, select **Model Advisor**.
4. In the Model Advisor dialog box, in the left pane, expand `By Product` and `Embedded Coder`.
5. Select the checks “Check modeling style for component deployment”, “Check signal interfaces”, and “Check configuration for component deployment”.
6. Run the selected checks.

If the checks do not pass, use the information provided to correct the conditions reported.

For more information, see “Modeling Guidelines for Generated Code” on page 23-5 and “Model Readiness for Code Generation”.

Generate and Inspect Generated Component Code

In the toolstrip, click **Generate Code**.

From the component model, the code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. Model data and entry-point functions are accessible to a caller by including the header file.

The code generator also produces an interface header file in subfolder `services`, which by default the code generator names `services.h`. This file includes declarations for platform service interface functions called by the component code.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with the target platform requirements.

For each callable entry-point function, the Code Interface Report provides:

- Function prototype
- Description
- Header file name
- Information about platform services called, such as prototypes and data communication method used

For each platform service called, the report provides:

- Function prototypes
- Data communication method used
- Callable functions that call the service

For more information, see “Analyze Generated Service Code Interface Report” on page 54-21.

Header File

Header file `ComponentDeploymentFcn.h` includes interface header file `services.h`, defines an instance of the real-time model data structure, and declares:

- Structure for representing DWork vectors
- Structure for representing measurable state data
- Structure for representing tunable parameters
- Structure for representing timing data
- Callable entry-point functions

```
#include "ComponentDeploymentFcn_types.h"
#include "services.h"
#define ComponentDeploymentFcn_M      (rtM)

typedef struct {
  real_T DataTransfer_WriteBuf[10];
  real_T DiscreteTimeIntegrator_PREV_U[10];
  uint32_T Integrator_PREV_T;
  uint8_T DiscreteTimeIntegrator_SYSTEM_E;
  boolean_T Integrator_RESET_ELAPS_T;
} D_Work;

typedef struct {
  real_T delay[10];
  real_T dti[10];
} CD_measured_T;

typedef struct {
  real_T gain;
} CD_tunable_T;

struct tag_RTM {
  struct {
    uint32_T clockTick2;
    uint8_T rtmH2LBufBeingRead2;
    uint8_T rtmH2LLastBufWr2;
    uint32_T rtmH2LDbBufClockTick2[2];
```

```
  } Timing;
};

extern D_Work rtDWork;
extern void CD_initialize(void);
extern void CD_terminate(void);
extern void CD_accumulator(void);
extern void CD_integrator(void);
extern CD_measured_T CD_measured;
extern CD_tunable_T CD_tunable;
extern RT_MODEL *const rtM;
```

Source Code File

The generated source code file `ComponentDeploymentFcn.c` shows the code for the four functions represented in the component model.

- Accumulator function calls the data transfer service function `get_CD_accumulator_DataTransfer()` to read an input value transferred from function `CD_integrator`. For each element of the bus signal, the function applies a delay and gain value and writes the output by calling sender service function `getref_CD_accumulator_OutBus_y`.
- Integrator function sets the elapsed time value to 0, if initialized, or to the current elapsed time. The integrator function calculates the current elapsed time by subtracting the cached elapsed time value (`rtDWork.Integrator_PREV_T)` from the current tick value that the function gets by calling the timer service function `get_tick_outside_CD_integrator`. The code generator assumes that the resolution of the clock is the fundamental step size of the model. The function uses the calculated elapsed time to calculate the integration and calls the receiver service function `get_CD_integrator_InBus_u` to receive the next input value.
- Initialize function calls the data transfer service function **`set_ModelInitialize_DataTransfer`** and receiver service function `get_CD_initialize_InBus_NVM` to initialize state data that is read from nonvolatile memory.
- Terminate function calls the sender service function `getref_CD_terminate_OutBus_NVM` to write the output of the accumulator function to memory.

```
#include "rtwtypes.h"
#include <string.h>
#include "ComponentDeploymentFcn.h"
#include "services.h"

CD_measured_T CD_measured;
CD_tunable_T CD_tunable = {

  3.0
};

D_Work rtDWork;
static RT_MODEL rtM_;
RT_MODEL *const rtM = &rtM_;
void CD_accumulator(void)
{
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += tmpIrvIRead[i];
```

```
    (getref_CD_accumulator_OutBus_y())[i] = CD_tunable.gain * CD_measured.delay[i];
  }
}

void CD_integrator(void)
{
  real_T tmp;
  real_T *tmp_0;
  int32_T i;
  uint32_T Integrator_ELAPS_T;
  tmp_0 = set_CD_integrator_DataTransfer();
  if (rtDWork.Integrator_RESET_ELAPS_T) {
    Integrator_ELAPS_T = 0U;
  } else {
    Integrator_ELAPS_T = (uint32_T)(get_tick_outside_CD_integrator() -
      rtDWork.Integrator_PREV_T);
  }

  rtDWork.Integrator_PREV_T = get_tick_outside_CD_integrator();
  rtDWork.Integrator_RESET_ELAPS_T = false;
  tmp = 1.25 * (real_T)Integrator_ELAPS_T;
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) {
      CD_measured.dti[i] += tmp * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.DiscreteTimeIntegrator_PREV_U[i] = (get_CD_integrator_InBus_u())[i];
}

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  memcpy(&tmp_0[0], &CD_measured.dti[0], (uint32_T)(10U * sizeof(real_T)));
}

void CD_initialize(void)
{
  {
    real_T *tmp;
    int32_T i;
    tmp = set_ModelInitialize_DataTransfer();
    for (i = 0; i < 10; i++) {
      tmp[i] = 5.0;
    }

    memcpy(&CD_measured.delay[0], &(get_CD_initialize_InBus_NVM())[0], (uint32_T)
            (10U * sizeof(real_T)));
    rtDWork.Integrator_RESET_ELAPS_T = true;
    rtDwork.DiscreteTimeIntegrator_SYSTEM_E = 1U;
  }
}

void CD_terminate(void)
{
  memcpy(&(getref_CD_terminate_OutBus_NVM())[0], &CD_measured.delay[0], (uint32_T)
          (10U * sizeof(real_T)));
}
```

Services Header File

The services header file `services.h` declares prototypes for calling services that run as platform software in a target environment.

```
/* timer services */
extern uint32_T get_tick_outside_CD_integrator(void);

/* data transfer services */
const real_T * get_CD_accumulator_DataTransfer(void);
real_T * set_CD_integrator_DataTransfer(void);
real_T * set_ModelInitialize_DataTransfer(void);

/* receiver services */
extern const real_T * get_CD_integrator_InBus_u(void);
extern const real_T * get_CD_initialize_InBus_NVM(void);

/* sender services */
extern real_T * getref_CD_accumulator_OutBus_y(void);
extern real_T * getref_CD_terminate_OutBus_NVM(void);
```

Test Generated Service Interface Code by Using SIL and PIL Simulations

Consider using a software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations to verify the generated code. A SIL simulation compiles and runs the generated code on your development computer. A PIL simulation compiles and runs the generated code on a target computer. During a SIL and PIL simulations, you can collect code coverage and execution-time metrics for the generated code.

To test numeric equivalence between the model and generated code:

1. Run a normal mode simulation and capture the results.
2. Run a SIL or PIL simulation and capture the results.
3. Compare the results from steps 1 and 2.

To run SIL and PIL simulations on the example model, you can use the example test harness model `ComponentDeploymentRateTestHarness.slx` and data file `ComponentDeploymentBus.mat`.

For more information, see “Software-in-the-Loop Simulation” and “Processor-in-the-Loop Simulation”.

Get Meta-Information About Generated Code Interface for Integration

By default, the code generator creates a code descriptor file (`codedescriptor.dmr`) in the build folder. That file contains meta-information about the generated code, including:

- Data interfaces (inports, outports, parameters, datastores, and internal data)
- Function interfaces (initialize, output, update, and terminate)
- Execution information for the data and function interfaces, such as timing requirements
- Model hierarchy information and information for referenced models

You can use the code descriptor programming interface to get access to the contents of the code descriptor file and use the results to confirm that generated interfaces meet integration requirements. You can also use the programming interface to provide input to tools that generate interfaces for target platform services.

For more information, see “Get Code Description of Generated Code” on page 54-44 and `coder.codedescriptor.CodeDescriptor` class.

Integrate Generated Component Code with Target Environment Code

To integrate the generated component code with a main function and other target environment code, you must:

- Match the data and function interfaces of the generated code with other interfaces of existing system code.
- Connect input data.
- Connect output data.
- Access other data, such as block state values, local parameters, and time.

The model used in this example is designed and configured such that the generated code aligns with the code interfaces of the target environment.

- The target environment software calls the generated entry-point functions, which provide input signal data and scheduling information.
- The generated algorithm calculates output data that the calling environment uses.
- The set of input and output data and the data access mechanisms constitute the interfaces of the entry-point functions.

In a model, root-level inports and outports represent the primary inputs and outputs of the component algorithm. By default, the code generator aggregates this input and output data into standard structures.

You can generate code for a component model and compile a component model library by clearing model configuration parameter **Generate code only** and initiating a model build. If code for the component model is already generated, you can build a component model library by using the `codebuild` command with the path *`codeGenerationFolder/modelBuildFolder/`*`services/lib`. You can link the generated component model library with a `main` function and other target environment code to create an executable program.

For the component model library, you can also create a:

- CMake configuration (`CMakeLists.txt`) file by using the `codebuild` function
- ZIP file by setting model configuration parameter **Package code and artifacts** and using the `packNGo` function

For more information, see “Manage Build Process Folders” on page 52-30, “Approaches for Building Code Generated from Simulink Models” on page 59-11, Configure CMake Build Process, “Compile Code in Another Development Environment” on page 52-99, and “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24.

For examples, see “Configure Generated Code According to Interface Control Document Specifications” on page 40-54 and “Integrate External Application Code with Code Generated from PID Controller” on page 40-45.

For more information, see “Deployment, Integration, and Supported Hardware”.

Related Examples

- “Generate Sender and Receiver C Interface Code for Component Deployment” on page 34-2
- “Generate C Data Transfer Service Interface Code for Component Deployment” on page 34-6
- “Generate C Timer Service Interface Code for Component Deployment” on page 34-13
- “Generate C Parameter Tuning Service Interface Code for Component Deployment” on page 34-20
- “Generate C Measurement Service Interface Code for Component Deployment” on page 34-24

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Deploy Rate-Based Component Configured for C Service Interface Code Generation

This example shows how to develop a single-rate model as a component that includes service interface support for interacting with target platform software. In this example you:

1. Assess project requirements.
2. Define the service code interface configuration in the coder dictionary.
3. Design the component model.
4. Configure the component model interfaces for code generation.
5. Assess component model readiness for code generation.
6. Generate and inspect the generated component code.
7. Test the generated service interface code by using a software-in-the-loop (SIL) or processor-in-the-loop (PIL).
8. Get meta-information about the generated code interface for integration with target platform software.
9. Integrate generated component code with target environment code.

Assess Project Requirements

This example assumes these component and target platform requirements.

Component Description and Requirements

The component model must:

- Favor memory optimization over data freshness when communicating data.
- Include an initialize function that reads a value from nonvolatile memory and uses that value to initialize a state variable.
- Include a component algorithm that receives the value of the state variable from the initialize function, increments the value by one, and applies a gain value, which is tunable during execution. The algorithm gets called every 1 second and maintains the most recent value of the state variable.
- Include a terminate function that writes the output of the integrator function to nonvolatile memory after the function stops executing.
- Make the state variable accessible for monitoring during execution.

Target Platform Requirements

The target platform for this example project has these requirements:

- To initiate component initialization, the target platform function scheduler expects to call entry-point function `CD_initialize` with no arguments.
- To execute the component algorithm, the target platform function scheduler expects to call entry-point function `CD_accumulator` with no arguments.
- To complete component termination, the target platform function scheduler expects to call entry-point function `CD_terminate` with no arguments.

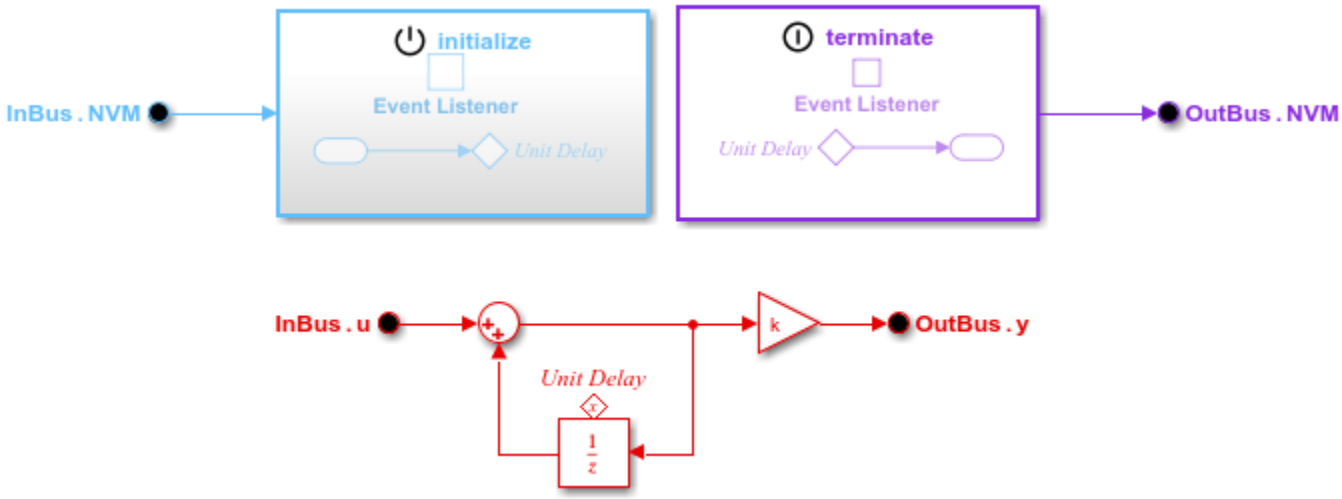
- The component reads from and writes to target platform nonvolatile memory.
- The platform receiver service function is named `get_CD_accumulator_InBus_u` and uses outside-execution data communication. For more information, see “Data Communication Methods” on page 1-14.
- The platform sender service function is named `getref_CD_accumulator_OutBus_y` and uses outside-execution data communication.
- The variable for the tunable gain value applied by the accumulator function is named `CD_tunable.k`.
- The variable that represents the delay state value is named `CD_measured.delay`.

Define Service Code Interface Configuration

You address the target platform requirements by defining a service code interface configuration in a shared Embedded Coder Dictionary. For this example, the requirements are defined in the dictionary file `ComponentDeploymentCoderDictionary.sldd`, which is linked to the example model.

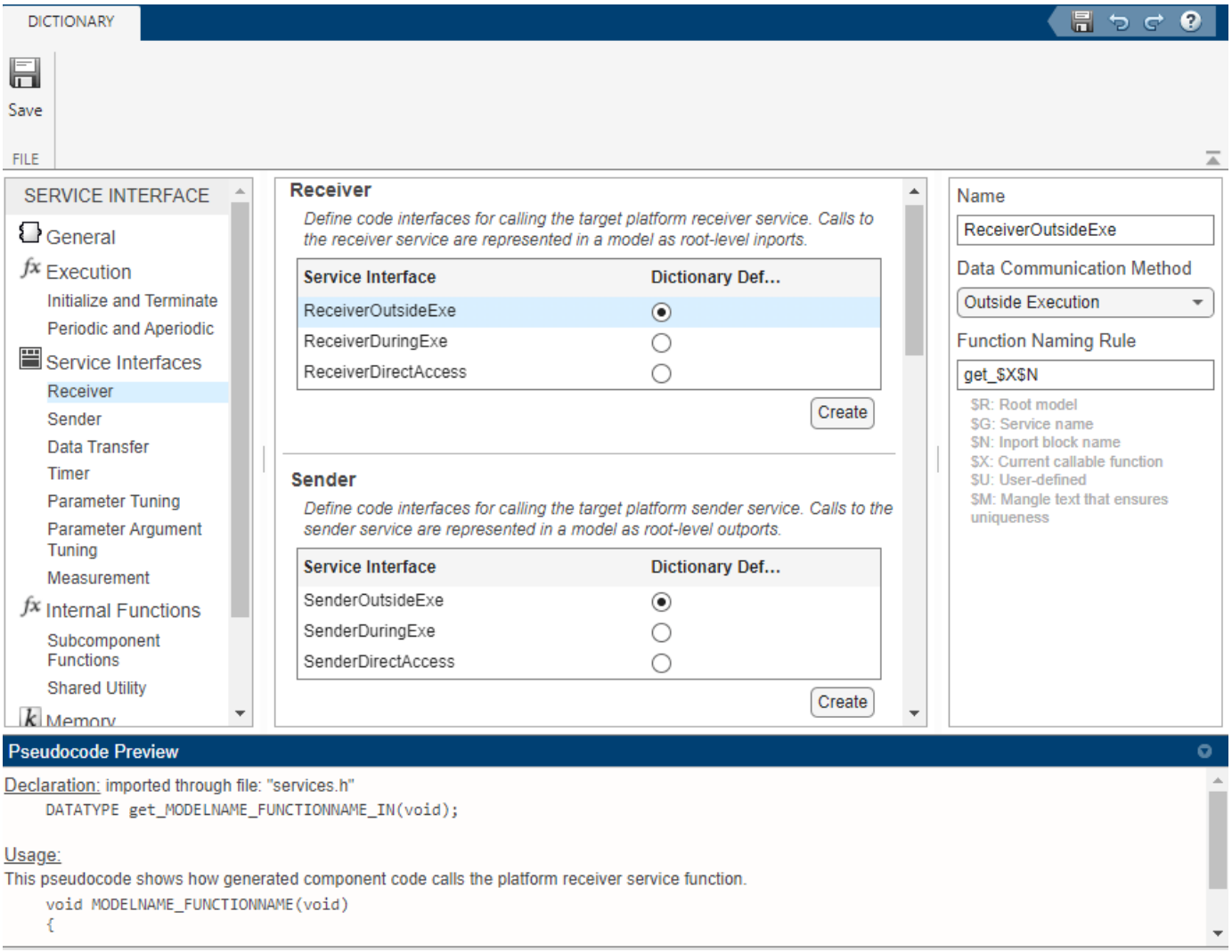
Open example component model `ComponentDeploymentRate`.

```
open_system('ComponentDeploymentRate');
```



Open the Embedded Coder Dictionary (`ComponentDeploymentCoderdictionary.sldd`), which is linked to the model.

1. Open the Embedded Coder app.
2. In the toolstrip, under **Code interface**, select `Embedded Coder Dictionary (Shared)`. The Embedded Coder Dictionary linked to the example model opens. You can create a coder dictionary in the Model Explorer (click **File** > **New** > **Embedded Coder Dictionary**) or in the model Configuration Parameters dialog box (click **Set up** to the right of the **Shared coder dictionary** parameter).
3. Under **Service Interface**, select **Receiver**. The Embedded Coder Dictionary shows content in three panes. Use the left pane to navigate through the dictionary configurations. The center pane lists the interfaces that are defined in a code interface configuration for the item you select in the navigation pane. From the center pane you can create and delete interfaces and specify a default interface. The right pane shows the property settings for the interface selected in the center pane. The Pseudocode Preview updates as you change property settings.



In the dictionary, browse through the interface settings. Check that the settings align with the requirements noted in Target Platform Requirements on page 3-103.

Based on requirements listed in Assess Project Requirements on page 3-103, the dictionary defines these relevant interfaces:

- Initialize and terminate callable entry-point function interface `InitTermInterface`, which specifies naming rule `CD_$N`, where `$N` is `initialize` or `terminate`, depending on the type of function generated.
- Periodic and aperiodic callable entry-point function interface `PeriodicAperiodicInterface`, which specifies naming rule `CD_$N`, where `$N` is the name of the function-call inport block. Later, you override the default naming in the code mappings.
- Receiver service interface `ReceiverOutsideExe` specifies outside-execution data communication and naming rule `get_$X$N`, where `$X` is the name of the callable function and `$N` is the name of the receiver port for the receive operation.
- Sender service interface `SenderOutsideExe` specifies outside-execution data communication and naming rules `set_$X$N` and `getref_$X$N` for the by value and by reference functions,

where `$X` is the name of the callable function and `$N` is the name of the sender port for the send operation.

- Parameter tuning service interface `ParameterTuningService` applies storage class `TunableStruct` to data elements. To align with the platform requirements for naming tunable parameters, property **Type Name** is set to `CD_tunable_T$M` and property **Instance Name** is set to `CD_tunable`.
- Measurement service interface, `MeasurementService`, applies storage class `MeasurementStruct`. To align with the platform requirements for naming state data, property **Type Name** is set to `CD_measured_T$M` and property **Instance Name** is set to `CD_measured`.

The during-execution and direct-access receiver and sender service interfaces are defined so that you can explore differences in the generated code for the different data communication methods.

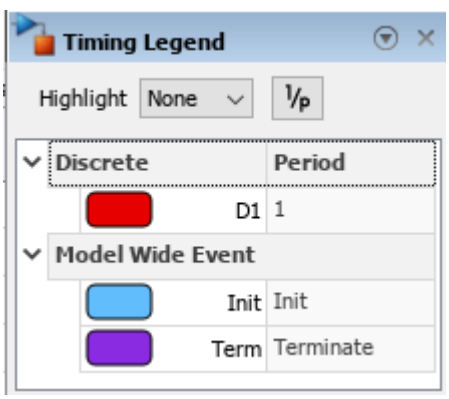
Later, in Configure Component Model Interface for Code Generation on page 3-107, you map elements of the example model to interfaces defined by the service interface configuration in the dictionary.

For more information about defining code interface configurations, see “Code Interface Definitions” and Embedded Coder Dictionary.

Design Component Model

The design of example component model `ComponentDeploymentRate` reflects the requirements listed in Assess Project Requirements on page 3-103.

The timing legend for the model identifies three callable functions: functions `Init` and `Term` and a discrete periodic function. To open the timing legend, on the **Debug** tab, click **Information Overlays** > **Timing Legend**.



- The component algorithm receives the value of the state variable from the initialize function, increments the value by one, and applies a gain value, which is tunable during execution. The algorithm gets called every 1 second and maintains the most recent value of the state variable. See “cgsl_0401: Modeling styles for component deployment”.
- The Initialize Function block represents a function that initializes a state variable `x` with a value read from nonvolatile memory before the algorithm executes. See “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment” and “cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks”.

- The In Bus Element block `InBus.NVM`, which feeds the initialize function, represents a call to the platform service that reads values from nonvolatile memory. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0405: Data receive for component deployment”.
- The inport for the model algorithm, `InBus.u`, represents a call to the platform receiver service for reading the input value from the target environment. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0405: Data receive for component deployment”.
- The outport of the algorithm represents a call to the platform sender service for writing the output value of the Gain block. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0406: Data send for component deployment”.
- The Terminate Function block represents the function that writes the output of the integration to nonvolatile memory after the component stops executing. See “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment” and “cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks”.

- The outport of the terminate function represents a call to the platform service that writes values to nonvolatile memory. See “cgsl_0402: Signal interfaces for component deployment” and “cgsl_0406: Data send for component deployment”.

For more information about modeling guidelines that can help you develop and deploy component models that use a service code interface, see “Modeling Guidelines for Generated Code” on page 23-5.

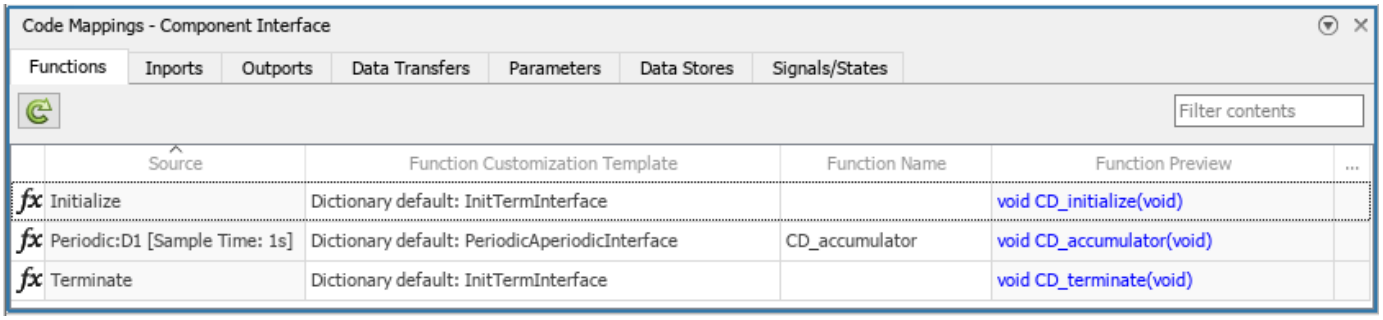
Configure Component Model Interface for Code Generation

For the code generator to produce interface code that aligns with the target environment requirements, there needs to be a mapping between the interface elements in the model and code interfaces. A model can be associated with one of two types of code interfaces: data interface or service interface. The code generator determines the type of code interface to apply automatically based on this information:

- Modeling styles and patterns used in the model
- Embedded Coder app **Output** selection
- Type and contents of the Embedded Coder Dictionary associated with the model

For this example, the code generator applies a service interface because the model is linked to a shared Embedded Coder Dictionary that defines a service interface configuration. The code generator uses the the linked dictionary to establish default mappings that you can view and, in some cases, change by using the Code Mappings Editor or code mappings programming interface.

To open the Code Mappings editor and explore the interface settings, in the toolstrip, under **Code Interface**, select **Component Interface**.



For elements for which the default interface aligns with the requirements, you do not need to make changes. Depending on your model, your code generation goals, and available interfaces, you might need to make some adjustments.

For this example, to meet the code interface requirements listed in Assess Project Requirements on page 3-103, these adjustments were made:

- On the **Functions** tab, the default function customization template for the periodic algorithm function produces function name `CD_Periodic`. To align with target platform requirements for this function, the code mapping overrides the default function name, by specifying the function name `CD_accumulator`.
- On the **Signals/States** tab, a code identifier was specified for the state associated with `delay`. To see the identifier setting for the state, select the row for the state and click the pencil icon. In the dialog box that appears, see the value specified for the **Identifier** property, `delay`. The identifier informs the code generator how to represent the corresponding variable in the generated code.

In the Code Mappings editor, you can change the interfaces for the sender and receiver services by clicking the corresponding tab and, in the service column, selecting an alternative service. The default interfaces use the outside-execution data communication method. Alternative interfaces use during-execution and direct-access data communication. Consider changing the interfaces to view differences in the generated code. If you choose to change the interfaces, for each service called along the data path of a signal, set the interfaces such that they apply the same data communication method.

For guidance on setting up a model to use a service interface configuration, see “cgsl_0414: Configure service interface for component model”. For more information about model code mappings for service interfaces, see “C Service Interfaces” and Code Mappings Editor – C.

Assess Component Model Readiness for Code Generation

Run the Simulink Model Advisor to confirm that the component model passes component modeling guideline checks.

1. In the Embedded Coder app, select **C/C++ Code Advisor**.
2. In the System Selector dialog box, select `ComponentDeploymentFcn`. Click **Ok**.
3. In the Code Generation Advisor dialog box, in the left pane, select **Model Advisor**.
4. In the Model Advisor dialog box, in the left pane, expand `By Product` and `Embedded Coder`.
5. Select the checks “Check modeling style for component deployment”, “Check signal interfaces”, and “Check configuration for component deployment”.

6 Run the selected checks.

If the checks do not pass, use the information provided to correct the conditions reported.

For more information, see “Modeling Guidelines for Generated Code” on page 23-5 and “Model Readiness for Code Generation”.

Generate and Inspect Generated Component Code

In the toolstrip, click **Generate Code**. From the component model, the code generator creates the folder `ComponentDeploymentRate_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentRate.h` and source code file `ComponentDeploymentRate.c`. Model data and entry-point functions are accessible to a caller by including the header file.

The code generator also produces an interface header file, which by default the code generator names `services.h`. This file includes declarations for platform service interface functions called by the component code.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with the target platform requirements.

For each callable entry-point function, the Code Interface Report provides:

- Function prototype
- Description
- Header file name
- Information about platform services called, such as prototypes and data communication method used

For each platform service called, the report provides:

- Function prototypes
- Data communication method used
- Callable functions that call the service

For more information, see “Analyze Generated Service Code Interface Report” on page 54-21.

Header File

Header file `ComponentDeploymentRate.h` includes interface header file `services.h` and declares:

- Structure for representing measurable state data
- Structure for representing tunable parameters
- Callable entry-point functions

```
#include"ComponentDeploymentRate_types.h"
#include "services.h"

typedef struct {
  real_T delay[10];
```

```
} CD_measured_T;

typedef struct {
  real_T k;
} CD_tunable_T;

extern void CD_initialize(void);
extern void CD_accumulator(void);
extern void CD_terminate(void);
extern CD_measured_T CD_measured;
extern CD_tunable_T CD_tunable;
```

Source Code File

Generate source code file `ComponentDeploymentRate.c` shows the code for the functions represented in the component model:

- Accumulator function, for each element of the bus signal, calls the receiver service function `get_CD_accumulator_Inbus_u()` to read the value of the state variable, increments the value by one, applies the delay and gain value, and calls the sender service function `getref_CD_accumulator_OutBus_y` to send the results.
- Initialize function calls the receive service function `get_CD_initialize_InBus_NVM` to initialize state data that is read from nonvolatile memory.
- Terminate function calls the sender service function `getref_CD_terminate_OutBus_NVM` to write the output of the accumulator algorithm to memory.

```
#include "services.h"
#include "rtwtypes.h"
#include <string.h>
#include "ComponentDeploymentRate.h"

CD_measured_T CD_measured;
CD_tunable_T CD_tunable = {

  0.0
};

void CD_accumulator(void)
{
  int32_T i;
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += (get_CD_accumulator_InBus_u())[i];
    (getref_CD_accumulator_OutBus_y())[i] = CD_tunable.k * CD_measured.delay[i];
  }
}

void CD_initialize(void)
{
  memcpy(&CD_measured.delay[0], &(get_CD_initialize_InBus_NVM())[0], (uint32_T)
         (10U * sizeof(real_T)));
}

void CD_terminate(void)
{
  memcpy(&(getref_CD_terminate_OutBus_NVM())[0], &CD_measured.delay[0], (uint32_T)
         (10U * sizeof(real_T)));
}
```

Services Header File

The services header file `services.h` declares prototypes for calling receiver and sender services that run as platform software in a target environment.

```
/* receiver services */
extern const real_T * get_CD_accumulator_InBus_u(void);
extern const real_T * get_CD_initialize_InBus_NVM(void);

/* sender services */
extern real_T * getref_CD_accumulator_OutBus_y(void);
extern real_T * getref_CD_terminate_OutBus_NVM(void);
```

Test Generated Service Interface Code by Using SIL and PIL Simulations

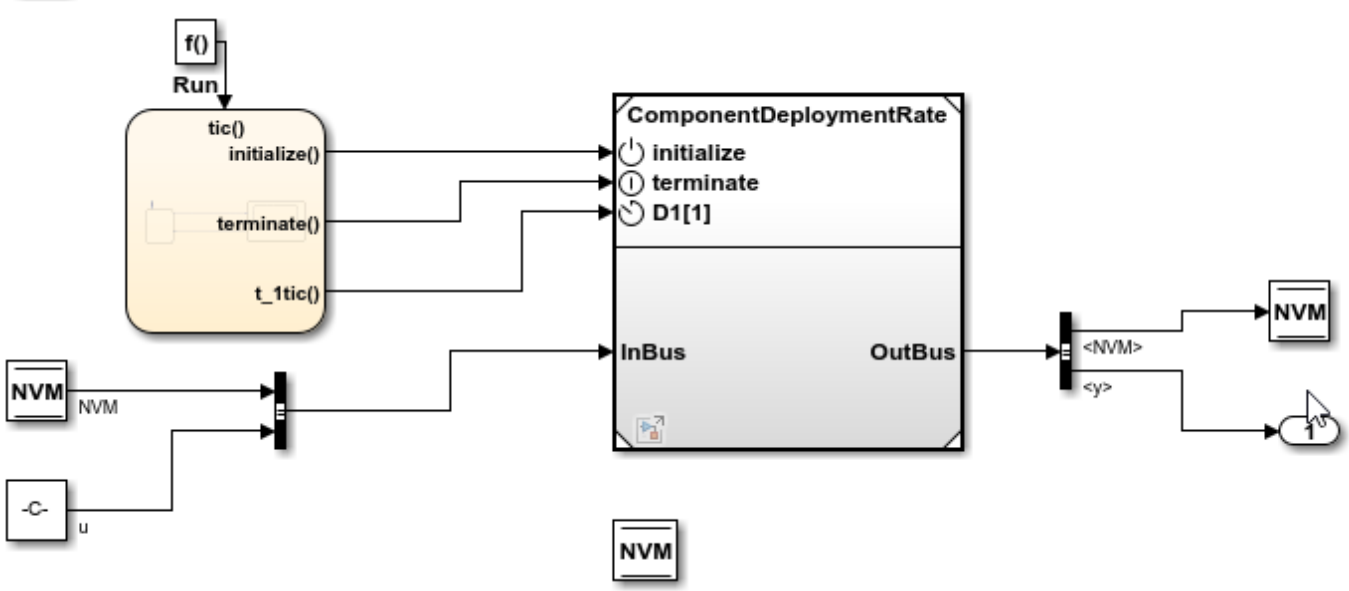
Consider using a software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations to verify the generated code. A SIL simulation compiles and runs the generated code on your development computer. A PIL simulation compiles and runs the generated code on a target computer. During a SIL and PIL simulations, you can collect code coverage and execution-time metrics for the generated code. For example, you can use the SIL simulation results to check for numerical equivalence between a normal mode simulation and the SIL simulation.

To test numeric equivalence between the model and generated code:

1. Run a normal mode simulation and capture the results.
2. Run a SIL or PIL simulation and capture the results.
3. Compare the results from steps 1 and 2.

To run SIL and PIL simulations on the example model, you can use the example test harness model `ComponentDeploymentRateTestHarness.slx` and data file `ComponentDeploymentBus.mat`.

```
open_system('ComponentDeploymentRateHarness');
```



For more information, see “Software-in-the-Loop Simulation” and “Processor-in-the-Loop Simulation”.

Get Meta-Information About Generated Code Interface for Integration

By default, the code generator creates a code descriptor file (`codedescriptor.dmr`) in the build folder. That file contains meta-information about the generated code, including:

- Data interfaces (inports, outports, parameters, datastores, and internal data)
- Function interfaces (initialize, output, update, and terminate)
- Execution information for the data and function interfaces, such as timing requirements
- Model hierarchy information and information for referenced models

You can use the code descriptor programming interface to get access to the contents of the code descriptor file and use the results to confirm that generated interfaces meet integration requirements. You can also use the programming interface to provide input to tools that generate interfaces for target platform services.

For more information, see “Get Code Description of Generated Code” on page 54-44 and `coder.codedescriptor.CodeDescriptor` class.

Integrate Generated Component Code with Target Environment Code

To integrate the generated component code with a main function and other target environment code, you must:

- Match the data and function interfaces of the generated code with other interfaces of existing system code.
- Connect input data.
- Connect output data.
- Access other data, such as block state values, local parameters, and time.

The model used in this example is designed and configured such that the generated code aligns with the code interfaces of the target environment.

- The target environment software calls the generated entry-point functions, which provide input signal data and scheduling information.
- The generated algorithm computes output data that the calling environment uses.
- The set of input and output data and the data access mechanisms constitute the interfaces of the entry-point functions.

In a model, root-level inports and outports represent the primary inputs and outputs of the component algorithm. By default, the code generator aggregates this input and output data into standard structures.

You can generate code for a component model and compile a component model library by clearing model configuration parameter **Generate code only** and initiating a model build. If code for the component model is already generated, you can build a component model library by using the `codebuild` command with the path *`codeGenerationFolder/modelBuildFolder/`*`services/lib`. You can link the generated component model library with a main function and other target environment code to create an executable program.

For the component model library, you can also create a:

- CMake configuration (`CMakeLists.txt`) file by using the `codebuild` function

- ZIP file by setting model configuration parameter **Package code and artifacts** and using the `packNGo` function

For more information, see “Manage Build Process Folders” on page 52-30, “Approaches for Building Code Generated from Simulink Models” on page 59-11, Configure CMake Build Process, “Compile Code in Another Development Environment” on page 52-99, and “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24.

For examples, see “Configure Generated Code According to Interface Control Document Specifications” on page 40-54 and “Integrate External Application Code with Code Generated from PID Controller” on page 40-45.

For more information, see “Deployment, Integration, and Supported Hardware”.

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Model Single-Core, Single-Tasking Application

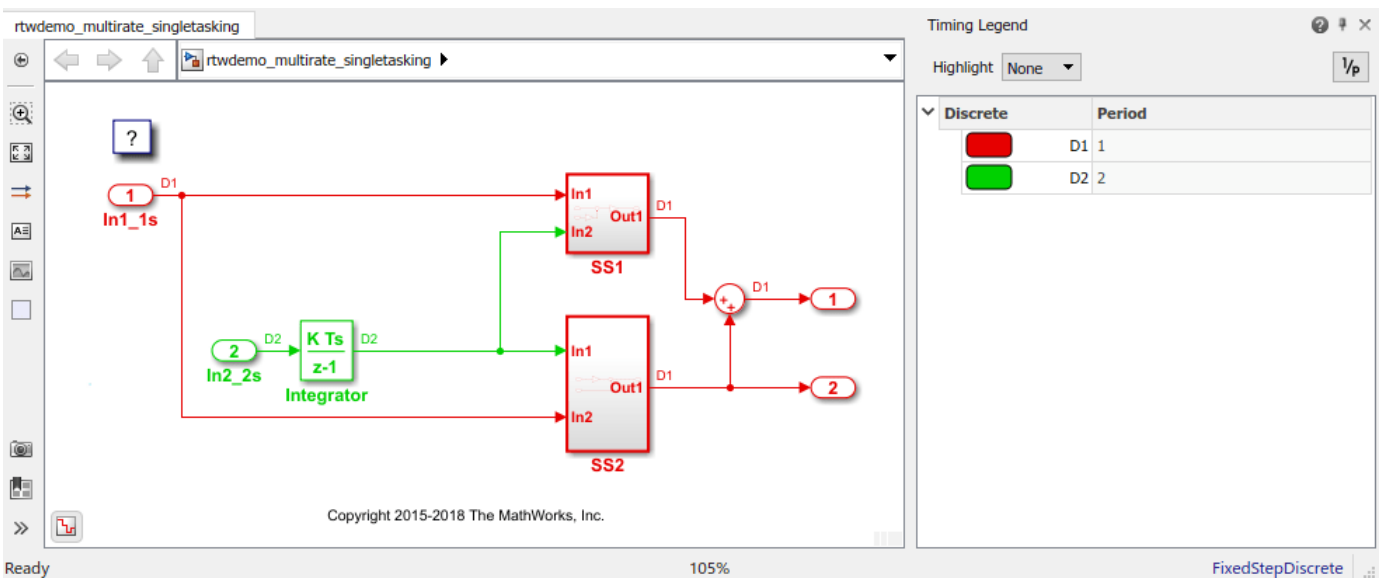
This example shows a model designed and configured for generating embedded system code intended to execute as an application in a single-core, single-tasking target environment. The application algorithm is captured as a multirate top model that is configured for C data interface code generation.

Periodic Multirate Model Set Up for Single-Tasking Execution

Open the example model.

```
open_system('MultirateSingleTasking');
```

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.
- To provide clean partitioning of rates, sample times for subsystems `SS1` and `SS2` are set to 1.

Relevant Model Configuration Parameter Settings

- **Type** set to `Fixed-step`.
- **Solver** set to `discrete (no continuous states)`.
- **Treat each discrete rate as a separate task** cleared.

Scheduling

Simulink® simulates the model based on the model configuration. Code generated from the model implements the same execution semantics. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, single-tasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks running at the different rates.

Benefits of implicit rate grouping:

- Simulink does not impose architectural constraints on the model.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, the model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates based on single-tasking execution semantics.

Your execution framework can communicate with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Open the **Embedded Coder** app. Then, generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point function `MultirateSingleTasking_step`. Use this file as a starting point for coding your execution framework.
- `MultirateSingleTasking.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `MultirateSingleTasking.h` declares model data structures and a public interface to the model entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include MultirateSingleTasking.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.In1_1s` of data type `real_T` with dimension of 1
- `rtU.In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void MultirateSingleTasking_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void rMultirateSingleTasking_step(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.

Output ports:

- `rtY.Out1` of data type `real_T` with dimension of 1
- `rtY.Out2` of data type `real_T` with dimension of 1

More About

- “Modeling for Single-Tasking Execution”
- “Deploy Applications to Target Hardware” on page 71-2
- “Customize Code Organization and Format” on page 55-46

Subsystems in Simulink Coder

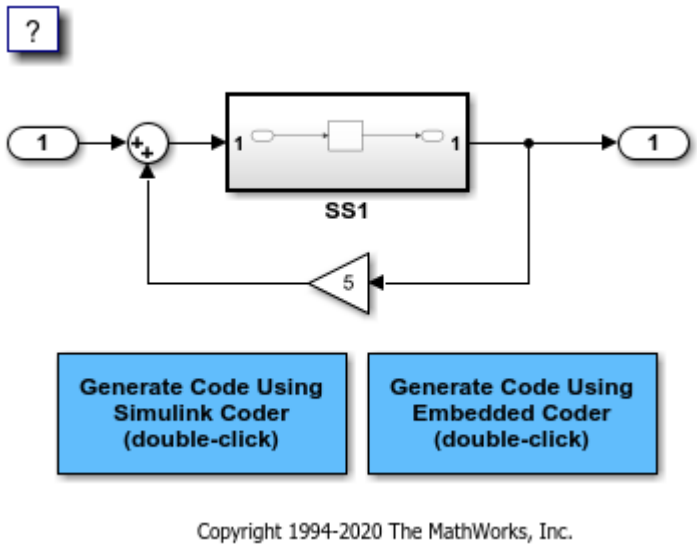
- “Generate Code and Executables for Individual Subsystems” on page 4-2
- “Generate Inlined Subsystem Code” on page 4-5
- “Generate Subsystem Code as Separate Function and Files” on page 4-7

Generate Code and Executables for Individual Subsystems

You can generate code and build a standalone executable for a subsystem within a model. When you generate code for an individual subsystem, the code generation and build process uses the code generation and build parameters of the root model.

1. Open a model with a subsystem, such as `rtwdemo_atomic`.

   ```
   open_system('rtwdemo_atomic')
   ```



2. Open Simulink Coder or Embedded Coder.

   If you are using Embedded Coder, from the **C Code** tab, click **View Code** to open the **Code View** editor.

3. Right-click the Subsystem block. From the context menu, click **Block Parameters (Subsystem)**. From the **Block Parameters: SS1** window, open the **Code Generation** tab.

4. Use the **Function name options** and **File name options** parameters to control the names of the generated functions and files. Set the **Function name options** parameter to `Auto` so that the code generator uses the name of the subsystem in the generated file names.

5. Right-click the Subsystem block. From the context menu, select **C/C++ Code > Build This Subsystem**.

6. To configure properties such as the name, class, and storage class of variables (or data objects) that are referenced as block parameters in the subsystem, convert the subsystem to a Model block. Configure the properties through the Code Mappings editor.

   For more information on tunable and inlined parameters and storage classes, see “Create Tunable Calibration Parameter in the Generated Code” on page 36-112.

   The build process displays status messages in the MATLAB Command Window. When the build is complete, the generated executable is in your working folder. The name of the generated executable is *`subsystem`*`.exe` (on PC platforms) or *`subsystem`* (on The Open Group UNIX®

platforms). *subsystem* is the name of the source Subsystem block. In this example, the Subsystem block is named SS1.

The generated code is in a build subfolder named *subsystem_target*_rtw. *subsystem* is the name of the source subsystem block and *target* is the name of the target configuration.

Subsystem Build Limitations

These limitations apply to building subsystems:

- You cannot use a subsystem build for a model configured with an ERT-based system target file and service code interface.
- A subsystem build does not support a subsystem that has a function-call trigger input or a function-call output.
- When you build a subsystem that includes an Outport block for which the **Data type** parameter specifies a bus object, you must address errors that result from setting signal labels. To configure the software to display these errors, in the Configuration Parameters dialog box for the parent model, on the **Diagnostics** > **Connectivity** pane, set the **Signal label mismatch** parameter to `error`.
- When a subsystem is in a triggered or function-call subsystem, the right-click build process might fail if the subsystem code is not sample-time independent. To determine whether a subsystem is sample-time independent:

  1. Copy all blocks in the subsystem to an empty model.
  2. In the Configuration Parameters dialog box, on the **Solver** pane, set:

     a. **Type** to `Fixed-step`.

     b. **Periodic sample time constraint** to `Ensure sample time independent`.

     c. Click **Apply**.
  3. Update the model. If the model is sample-time dependent, Simulink generates an error in the process of updating the diagram.
- When you use the right-click build process for a subsystem, the code generator attempts to use the subsystem name for generated code files. In some cases, there can be a conflict with the name that you specify when you set, for example, **File name options** to `Use function name` or **Function name options** to `Use subsystem name`. You see an error:

  ```
  The subsystem 'model/subsys'
  is trying to generate code to an reserved file (subsys) for
  the model 'subsys'...
  ```

  To resolve the error, modify one of the conflicting file names so that the names are unique.
- In a subsystem build warning, the subsystem block path hyperlink that is created references a temporary model block path instead of the actual model block path. In the Diagnostic Viewer, clicking the subsystem hyperlink does not take you to the block. In the Command Window, you see a message:

  ```
  ...
  No system or file called 'subsystemName' found.
  ...
  ```

See Also

Related Examples

- “Generate Predictable Function Interface to Match Graphical Block Interface” on page 5-2
- “Generate Subsystem Code as Separate Function and Files” on page 4-7

More About

- “Modeling Guidelines for Subsystems” on page 25-12

Generate Inlined Subsystem Code

To inline subsystem code with your model code, configure a nonvirtual subsystem. In the Subsystem Block Parameters dialog box, the **Function packaging** parameter specifies the format of the subsystem generated code. This parameter has these settings:

- Auto
- Inline
- Nonreusable function
- Reusable function

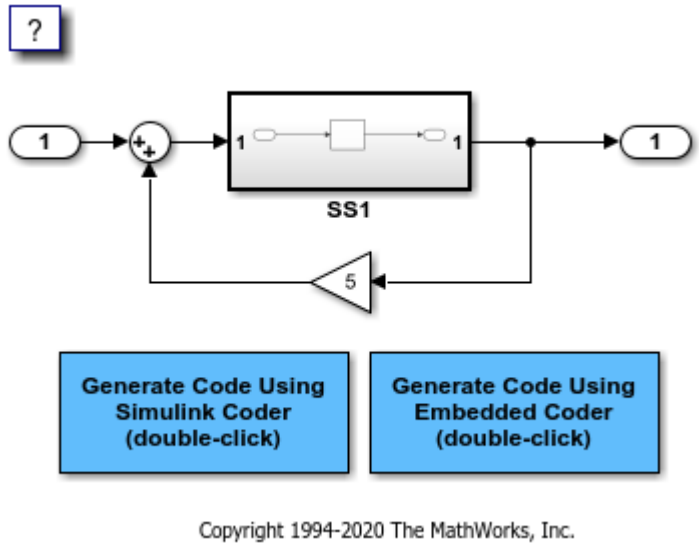
The code generator can generate inlined code for the `Auto` and `Inline` settings. The inline setting directs the code generator to inline the subsystem code unconditionally.

The default `Auto` setting directs the code generator to generate the most efficient code for the subsystem based on the type and number of instances of the subsystem that exist in the model. When there is only one instance of a subsystem, the `Auto` setting inlines the subsystem code. In Embedded Coder, when there are multiple instances of a subsystem that are not too complex, the `Auto` setting inlines the code for each subsystem. Otherwise, the `Auto` setting generates a single copy of the function (as a reusable function). For a function-call subsystem that has multiple callers, the `Auto` setting generates subsystem code that is consistent with the `Nonreusable function` setting.

Configure Subsystem to Inline Code

1. Open a model with a subsystem, such as `rtwdemo_atomic`.

   ```
   open_system('rtwdemo_atomic')
   ```

2. Open Simulink Coder or Embedded Coder.

   If you are using Embedded Coder, from the **C Code** tab, click **View Code** to open the **Code View** editor.

3. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
4. In the Subsystem Parameters dialog box, verify that **Treat as atomic unit** is selected. This option makes the subsystem nonvirtual. When the subsystem is nonvirtual, the **Function packaging** option is available on the **Code Generation** tab.
5. Click the **Code Generation** tab and select `Auto` or `Inline` from the **Function packaging** parameter.
6. Click **Apply** and close the dialog box.

   The border of the subsystem thickens, indicating that it is nonvirtual.
7. Generate code from the model. The code generator inlines subsystem code within `rtwdemo_atomic.c`. You can identify this code by system and block identification tags, such as:

   ```
   /* Atomic SubSystem Block: <Root>/AtomicSubsys1 */
   ```

Exceptions to Inlining

There are certain cases in which the code generator does not inline a nonvirtual subsystem, even though you select the `Inline` setting.

- If a noninlined S-function calls a function-call subsystem, the code generator ignores the `Inline` setting. Because noninlined S-functions use function pointers to make function calls, the code generator must generate a function with all arguments present.
- In a feedback loop involving function-call subsystems, the code generator generates a function instead of inlined code for one of the subsystems. Based on the internal, sorted order of the subsystems, the code generator selects which subsystem to generate a function.
- If an S-function, an Async Interrupt, or a Task Sync block that has the option `SS_OPTION_FORCE_NONINLINED_FCNCALL` set to `TRUE` calls a subsystem, the code generator generates a function instead of inlined code for the subsystem. The VxWorks® block library (`vxlib1`) contains the user-defined Async Interrupt and Task Sync blocks.[1]

The **Function packaging** parameter is disabled in Simulink Function blocks.

See Also

- “Generate Subsystem Code as Separate Function and Files” on page 4-7
- “Generate Reentrant Code from Subsystems”
- “Remove Algebraic Loops”

1 VxWorks is a registered trademark of Wind River Systems, Inc.

Generate Subsystem Code as Separate Function and Files

You can configure an atomic subsystem to generate code to a separate function and file. This results in code that is more modular and enables you to unit test code for individual subsystems.

The **Function packaging** Subsystem block parameter has two settings that cause a subsystem to generate code as a separate function. The `Reusable function` option generates a function that passes I/O, states, and parameters as function arguments. The `Nonreusable function` option generates a function that passes I/O, states, and parameters as a combination of function arguments and global data structures.

Subsystem Function Dependence

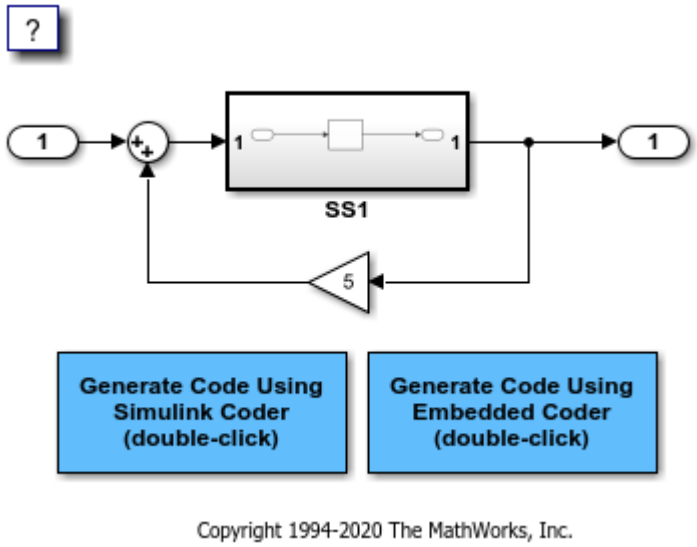
When generating code for a subsystem, the code can reference global data structures of the model, even if the subsystem function code is in a separate file. Each subsystem code file contains `include` directives and comments describing the dependencies. The code generator checks for cyclic file dependencies and produces warnings about them at build time. For descriptions of how the code generator packages code, see “Manage Build Process File Dependencies”.

To generate subsystem function code that is independent of the code generated for the parent model, place the subsystem in a library and configure it as a reusable subsystem, as described in “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36.

If you have Embedded Coder, you can generate code for library consisting of reusable subsystems that have different function interfaces. For more information, see “Library-Based Code Generation for Reusable Library Subsystems” on page 9-2.

Generate Subsystem as a Reusable Function

1 Open a model with a subsystem, such as `rtwdemo_atomic`.



If you are using Embedded Coder, from the **C Code** tab, click **View Code** to open the **Code View** editor.

2. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
3. In the Subsystem Parameters dialog box, verify that **Treat as atomic unit** is selected. With that parameter selected, on the **Code Generation** tab, the **Function packaging** parameter is available.
4. Click the **Code Generation** tab and select `Reusable function` from the **Function packaging** parameter. This enables two parameters:
   - The **Function name options** parameter controls the naming of the generated function.
   - The **File name options** parameter controls the naming of the generated file.
5. Set the **File name options** parameter to `Use subsystem name`.
6. Click **Apply** and close the dialog box.
7. If you are using Embedded Coder, open the Configuration Parameters dialog box. Verify that the model configuration parameter **File packaging format** is set to `Modular`.
8. Generate the code.

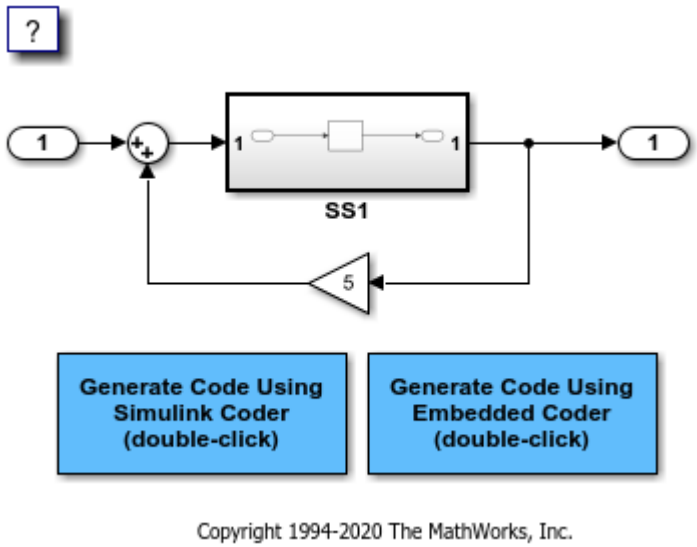
```
#include "SS1.h"

/* Include model header file for global data */
#include "rtwdemo_atomic.h"
#include "rtwdemo_atomic_private.h"

/* Outputs for atomic system: '<Root>/SS1' */
real_T myfun(DW_myfun_T *localDW)
{
  /* DiscreteIntegrator: '<S1>/Integrator' */
  return localDW->Integrator_DSTATE;
}

/* Update for atomic system: '<Root>/SS1' */
void myfun_Update(real_T rtu_In1, DW_myfun_T *localDW)
{
  /* Update for DiscreteIntegrator: '<S1>/Integrator' */
  localDW->Integrator_DSTATE += rtu_In1;
}
```

   The reusable function `myfun` passes in inputs and states as arguments to the subsystem function.

For more information, see “Generate Reentrant Code from Subsystems” on page 8-28 and “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36.

Generate Subsystem as a Nonreusable Function

1. Open a model with a subsystem, such as `rtwdemo_atomic`.



If you are using Embedded Coder, from the **C Code** tab, click **View Code** to open the **Code View** editor.

2. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
3. In the Subsystem Parameters dialog box, verify that **Treat as atomic unit** is selected. With that parameter selected, on the **Code Generation** tab, the **Function packaging** parameter is available.
4. Click the **Code Generation** tab and select `Nonreusable function` from the **Function packaging** parameter. This enables two parameters:
   - The **Function name options** parameter controls the naming of the generated function.
   - The **File name options** parameter controls the naming of the generated file.
5. Set the **File name options** parameter to `Use subsystem name`.

   If you are using Embedded Coder, verify that the **Function interface** parameter is set to `void_void`. For more information, see “Generate Predictable Function Interface to Match Graphical Block Interface” on page 5-2.
6. Click **Apply** and close the dialog box.
7. If you are using Embedded Coder, open the Configuration Parameters dialog box. Verify that the model configuration parameter **File packaging format** is set to `Modular`.
8. Generate the code.

```
#include "SS1.h"

/* Include model header file for global data */
#include "rtwdemo_atomic.h"
#include "rtwdemo_atomic_private.h"

/* Outputs for atomic system: '<Root>/SS1' */
void myfun(void)
{
  /* Outport: '<Root>/Out1' incorporates:
```

```
   *  DiscreteIntegrator: '<S1>/Integrator'
   */
  rtwdemo_atomic_Y.Out1 = rtwdemo_atomic_DW.Integrator_DSTATE;
}

/* Update for atomic system: '<Root>/SS1' */
void myfun_Update(void)
{
  /* Update for DiscreteIntegrator: '<S1>/Integrator' */
  rtwdemo_atomic_DW.Integrator_DSTATE += rtwdemo_atomic_B.Sum;
}
```

The nonreusable function `myfun` passes in inputs and states through the global data structures `rtwdemo_atomic_Y` and `rtwdemo_atomic_DW`.

See Also

More About

- “Generate Reentrant Code from Subsystems” on page 8-28
- “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36
- “Generate Code and Executables for Individual Subsystems”
- “Generate Predictable Function Interface to Match Graphical Block Interface” on page 5-2
- “Remove Algebraic Loops”

Subsystems in Embedded Coder

Generate Predictable Function Interface to Match Graphical Block Interface

You can generate code for a nonreusable subsystem that has a predictable function interface. A predictable interface can be useful for debugging and testing your code and integrating it with external applications. The generated code with a predictable function interface is more readable.

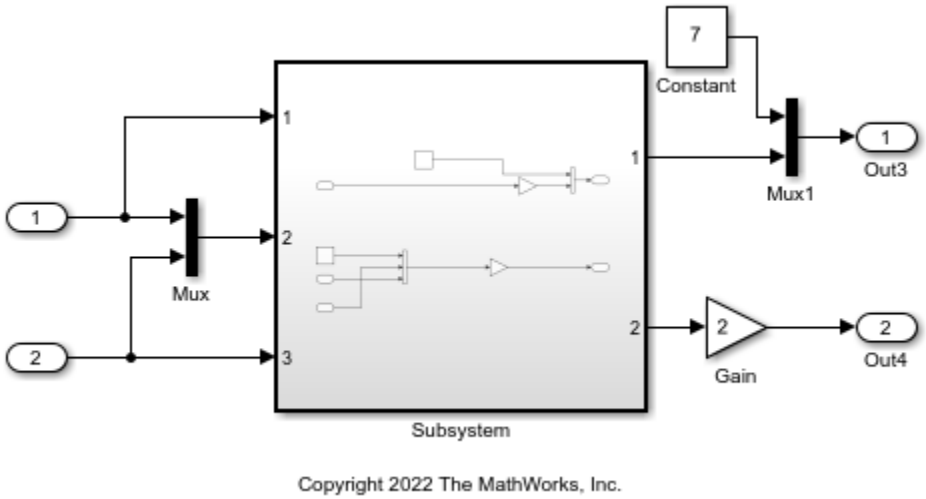
When you specify block parameter **Function interface** as `Allow arguments (Match graphical interface)`, the generated function interface matches the subsystem graphical block interface. For example, if a subsystem has two Inports and one Outport, the function interface generates two arguments for the Inports and one function argument for the Outport. The order of the arguments is determined by the port index. The function interface is predictable and does not change.

When you generate code for a nonreusable subsystem by using the block parameter **Function interface** set to `Allow arguments (Optimized)`, the code generator generates a subsystem function interface that passes data through arguments. The function arguments might not match the subsystem graphical block interface.

This example shows how the generated code is different when you specify **Function interface** as `Allow arguments (Optimized)` and `Allow arguments (Match graphical interface)`.

Generate Function Interface That Has Optimized Arguments

1. Open the example model `ex_mux_allow_arguments`. The subsystem in the model contains three inputs and two outputs.



2. Right-click the subsystem and select **Block Parameters (Subsystem)** from the context menu. Alternatively, click in the subsystem to open the Block Parameters dialog box in the **Property Inspector** pane.

These preset settings are required to generate expected code:

- In the **Main** tab, the block parameter **Treat as atomic unit** is selected.

- In the **Code Generation** tab, **Function packaging** is `Nonreusable function` and **Function interface** is `Allow arguments (Optimized)`.

3. Open the **Embedded Coder** app. Click **Build**.

The generated code in `ex_mux_allow_arguments.c` file contains a subsystem function interface with arguments:

```
/* Output and update for atomic system: '<Root>/Subsystem' */
static void FPCSS_Mux(real_T rtu_In1, real_T rtu_In2, real_T rty_Out2[4])
{
  /* Constant: '<S1>/Constant1' */
  rtDW.Constant1 = 5.0;

  /* Gain: '<S1>/Gain4' */
  rtDW.Gain4 = 10.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' incorporates:
   *  Constant: '<S1>/Constant'
   */
  rty_Out2[0] = 150.0;
  rty_Out2[1] = 15.0 * rtu_In2;
  rty_Out2[2] = 15.0 * rtu_In1;
  rty_Out2[3] = 15.0 * rtu_In2;
}
```

The `Allow arguments (Optimized)` specification leads to optimized generated code by reducing global RAM. The function interface has some inputs and outputs as function arguments. The generated function interface does not match the subsystem graphical block interface.

Generate Function Interface That Has Arguments Matching Graphical Interface

1. In the example model `ex_mux_allow_arguments`, open the Subsystem Block Parameters dialog box.

2. On the **Code Generation** tab, specify **Function interface** as `Allow arguments (Match graphical interface)` and click **Apply**.

3. Generate code for the model.

The generated code in `ex_mux_allow_arguments.c` file contains a subsystem function interface with arguments that match the subsystem graphical block interface:

```
/* Output and update for atomic system: '<Root>/Subsystem' */
static void FPCSS_Mux(real_T rtu_In1, const real_T rtu_In2[2], real_T rtu_In3,
                      real_T rty_Out1[2], real_T rty_Out2[4])
{
  /* SignalConversion generated from: '<S1>/Out1' incorporates:
   *  Constant: '<S1>/Constant1'
   *  Gain: '<S1>/Gain4'
   */
  rty_Out1[0] = 5.0;
  rty_Out1[1] = 10.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' incorporates:
```

```
   *  Constant: '<S1>/Constant'
   */
  rty_Out2[0] = 150.0;
  rty_Out2[1] = 15.0 * rtu_In3;
  rty_Out2[2] = 15.0 * rtu_In2[0];
  rty_Out2[3] = 15.0 * rtu_In2[1];
}
```

The `Allow arguments (Match graphical interface)` specification generates a predictable interface and maps the function arguments to the graphical interface of the Subsystem block. Arrays as part of the function arguments account for the combination of input or output signals that the Mux block uses.

For the Enabled, Triggered, or Resettable Subsystem with the `Allow arguments (Match graphical interface)` specification, the generated function contains arguments corresponding to each Enable, Trigger block, and Reset block present within that subsystem.

Limitations

- The **Function interface** set as `Allow arguments (Match graphical interface)` specification is not applicable to context menu option **C/C++** > **Build this Subsystem**.
- You cannot control the argument names.

See Also

Related Examples

- Subsystem
- “Reduce Global Variables in Nonreusable Subsystem Functions” on page 80-8

Referenced Models in Simulink Coder

- “Generate Code for Model Reference Hierarchy” on page 6-2
- “Set Configuration Parameters for Code Generation of Model Hierarchies” on page 6-8
- “Convert Subsystem to Referenced Model and Generate Code” on page 6-13
- “Generate Reusable Code from Model Reference Hierarchy” on page 6-16
- “Storage Classes for Parameters and Signals Used in Model Blocks” on page 6-19
- “Code Generation Limitations for Model Reference” on page 6-22
- “Simulink Cache Files for Incremental Code Generation” on page 6-24
- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 6-27
- “Generated Functions in a Model Reference Hierarchy” on page 6-51

Generate Code for Model Reference Hierarchy

To achieve incremental loading of models, incremental code generation, modular development of referenced models, and code reuse, you can generate code for the model reference hierarchy.

You can include one model inside another model by using a Model block. Each instance of a Model block is a model reference. For simulation and code generation, blocks within a referenced model execute together as a unit. The model that contains a referenced model is a parent model. A collection of parent and referenced models constitutes a model hierarchy.

Model Reference Technology

When you generate code for a referenced model hierarchy, the code generator produces a standalone executable for the top model and a library module called a model reference target for each referenced model. When the code executes, the top model executable invokes the model reference targets to compute the referenced model output. Model reference targets are sometimes called *Simulink Coder targets*.

Do not confuse a model reference target (Simulink Coder target) with these other types of targets:

- Target hardware — A platform for which the Simulink Coder software generates code.
- System target file — A file that instructs the Simulink Coder software how to generate code for particular purpose.
- Rapid Simulation target (RSim) — A system target file that is part of the Simulink Coder product.
- Simulation target — A MEX file that implements a referenced model, which is executed by using Simulink Accelerator™ software.

The code generator places the code for the top model in a hierarchy in the code generation folder and places the code for referenced models in an `slprj` folder. Subfolders in `slprj` are separate places for different types of files. For folder information, see “Manage Build Process Folders”.

By default, the Simulink Coder uses *incremental code generation*. When generating code, the software compares structural checksums of referenced model files to the generated code files to determine whether to regenerate model reference targets. To control when rebuilds occur, use the configuration parameter **Model Referencing** > **Rebuild**. For details, see Rebuild.

The Simulink Coder software also uses *incremental loading*. The code for a referenced model is not loaded into memory until the code for its parent model executes and requires the outputs of the referenced model. Simulink Coder then loads the referenced model target and executes the code. Once loaded, the target remains in memory until it is no longer used.

If you have Embedded Coder, custom targets must declare themselves as compliant with the model reference if they support Model blocks. For more information, see “Support Model Referencing” on page 94-61.

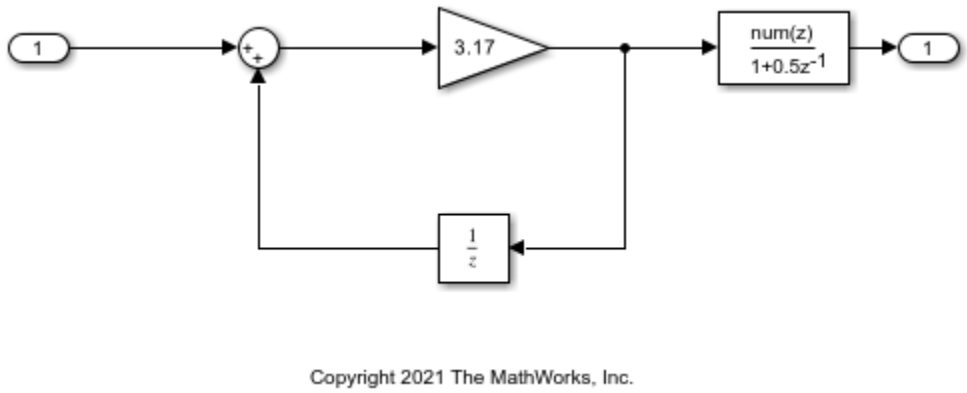
Generate Code for Model Reference Hierarchy

This example shows how to reference an existing model and generate code for the model reference hierarchy.

Create Model Reference Hierarchy

1. Open an existing model, for example the `ex_model_arg_ref` model. This model represents a reusable algorithm. This is a GRT model. If you have Embedded Coder, change the model configuration parameter **System target file** to `ert.tlc`.

```
open_system('ex_model_arg_ref');
```

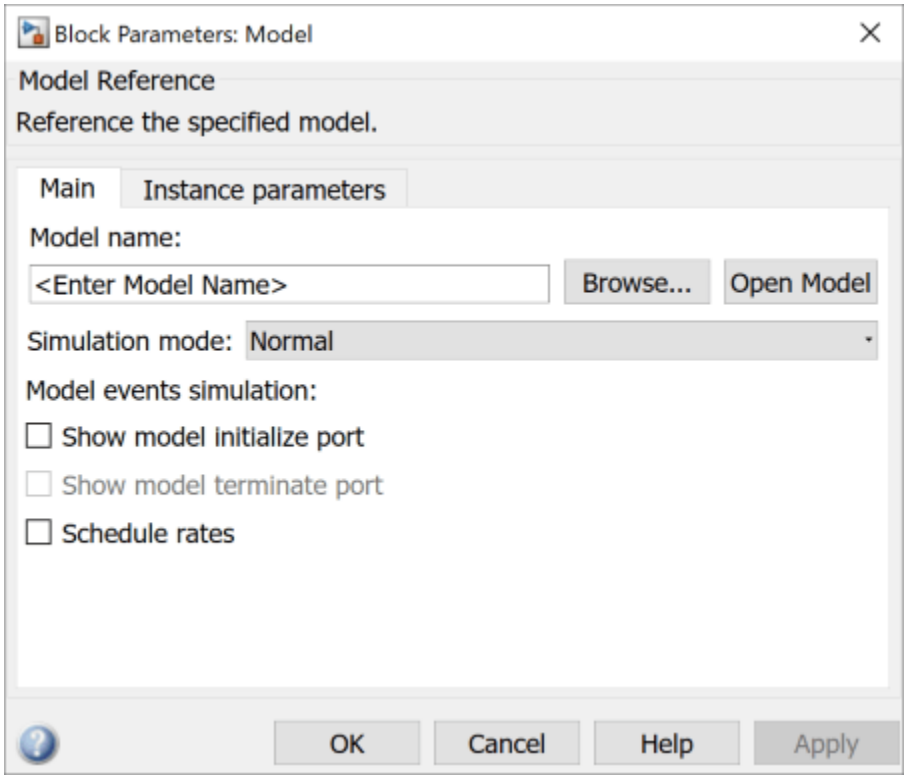


2. A referenced model must specify that it is available to be referenced and whether it can be referenced just once or multiple times. Specify the model configuration parameter Total number of instances allowed per top model as `One` or `Multiple`. By default, **Total number of instances allowed per top model** is set to `Multiple`.

3. Create a new model that you want as a top model in the model reference hierarchy. The top model refers to other models as referenced models or multiple instances of a referenced model.

4. Add Inports and Outports to the top model as required. To add a referenced model, use the Model block.

5. The new block is initially unresolved because it does not specify a referenced model. To open the block parameters dialog box, double-click the Model block or use the context menu of the Model block.

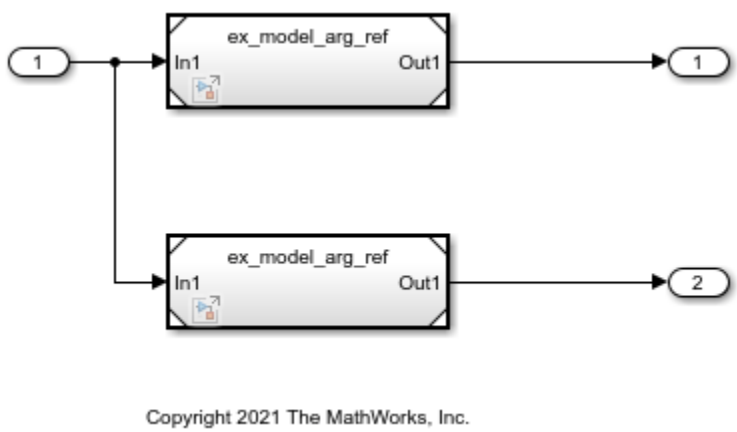


6. Enter the name of the referenced model in the **Model name** field. This name must contain fewer than 60 characters, exclusive of the `.slx` suffix.

7. Click **OK**. If the referenced model contains root-level inputs or outputs, the Model block displays corresponding input and output ports.

8. Create another instance of the Model block so that the top model looks like model:

```
open_system('ex_model_arg');
```



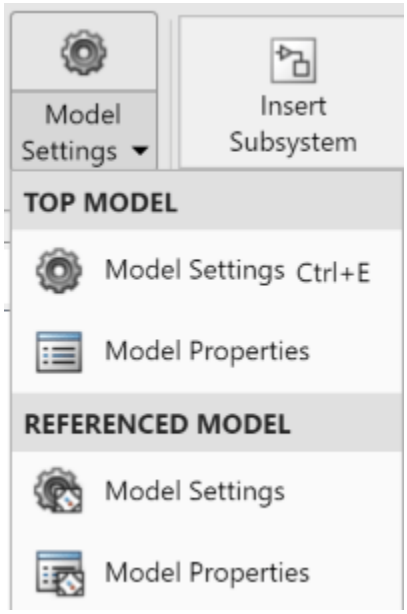
9. Name the model `ex_model_arg` and save the model.

You can also convert a subsystem to a referenced model. For more details, see “Convert Subsystems to Referenced Models”.

Generate Code for Model Reference Hierarchy

1. Verify that the model configuration parameter Solver selection **Type** is set to `Fixed-step` for the top model and the referenced model. To open the respective Configuration Parameters dialog boxes, on the **Modeling** tab, select **Model Settings** for the respective models.

For a model reference hierarchy that has multiple referenced models, you can access the **Model Settings** of the referenced model that is open and the top model through this menu. To access the **Model Settings** of another referenced model in the model hierarchy, open the referenced model to access this menu.



2. For the top model, open the **C Code** tab by selecting either **Simulink Coder** app or **Embedded Coder** app. To generate code for the model reference hierarchy, click the **Build** button.

The Simulink Coder build process generates and compiles code. The current folder now contains new files and a new folder.

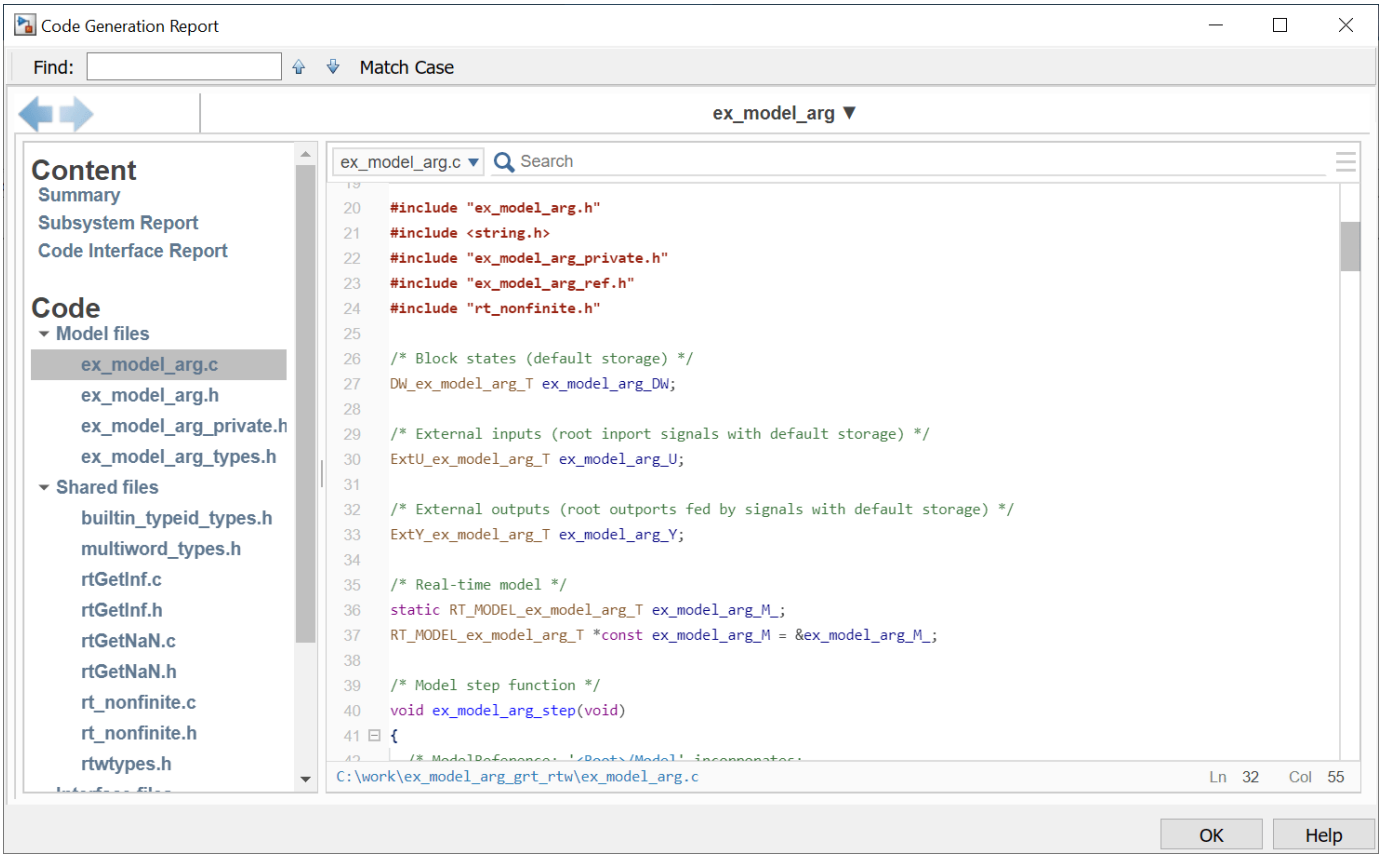
- `ex_model_arg_grt_rtw` — The build folder containing the generated code for the top model.
- `slprj > grt > ex_model_arg_ref` — The folder containing the generated code for the referenced model.
- `ex_model_arg.exe` — The executable created by the build process.
- `ex_model_arg.slxc` — The Simulink cache file for the top model.
- `ex_model_arg_ref.slxc` — The Simulink cache file for the referenced model.

For more information about the configuration parameter requirements, see “Set Configuration Parameters for Code Generation of Model Hierarchies”.

View Generated Code in Code Generation Report

To view the generated code, in the **C Code** tab, select **Open Report**. The code generation report contains hyperlinks to all the files generated for the model reference hierarchy.

To open the code generation report for a referenced model, at the top of the report, click `ex_model_arg`. In the drop-down, click `ex_model_arg_ref`. The source files for the referenced model are displayed along with the generated code for the referenced model.



To navigate back to the top model, click the **Go Backward** button on the top-left corner of the code generation report window.

Working with Code Generation Folders

Model reference code is generated in your code generation folder. Simulation target code is generated in your simulation cache folder. Due to the two different folders, there are constraints on:

- When and where you build model reference targets.
- How you access the model reference targets.

You can store the models referenced by Model blocks anywhere on the MATLAB path. A top model can include models stored on different file systems or in different folders. The simulation targets and generated code derived from these models cannot be stored on different file systems or in different folders. To allow code reuse, you must set up models referenced by a top model to simulate and generate model reference target code in a single code generation folder.

If you reference the same model from several top models, each stored in a different folder, you must choose one of these approaches:

- Work with the same code generation folder and be sure that the models are on your path.
- Allow separate code generation folders, simulation targets, and Simulink Coder targets to be generated in each folder in which you work.

The second approach requires maintenance of several instances of the model reference code. It is possible for generated code to become redundant, for example, when you make changes to the referenced model. To minimize code regeneration of referenced models, choose a specific code generation folder for all sessions.

Build Model Reference Targets

The code generator produces a model reference target directly from the Simulink model. The code generator generates or regenerates model reference targets, for example, when they require an update.

When you execute the `slbuild` command by using arguments in the MATLAB Command Window, Simulink and Simulink Coder can generate a simulation target for an accelerator mode referenced model and a model reference target for a referenced model.

The code generator produces only one model reference target for multiple instances of a referenced model. For more information, see “Generate Reentrant Code from Subsystems”.

Reduce Change Checking Time

You can reduce the time that Simulink and Simulink Coder spend checking whether model reference targets need to be rebuilt by setting configuration parameter values:

- In the top model, consider setting the model configuration parameter **Model Referencing** > **Rebuild** to `If changes in known dependencies detected`.
- In the referenced models throughout the hierarchy, set the configuration parameter **Diagnostics** > **Data Validity** > **Signal resolution** to `Explicit only` or `None`.

These parameter values exist in a referenced model configuration set, not in the individual Model block. Setting either value for an instance of a referenced model sets it for the instances of that model.

See Also

Related Examples

- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 6-27
- “Establish Data Ownership in a Model Hierarchy” on page 37-10
- “Use Configuration Reference to Select Code Generation Target” on page 48-11
- “Manage Simulation Targets for Referenced Models”
- “Generate Reentrant Code from Top Models” on page 8-13

Set Configuration Parameters for Code Generation of Model Hierarchies

A model reference hierarchy must satisfy various Simulink Coder requirements, as described in this topic. A model referencing hierarchy to be processed by the Simulink Coder software must also satisfy:

- The Simulink requirements listed in:
  - "Set Configuration Parameters for Model Hierarchies"
  - "Model Reference Interface and Boundary"
- The Simulink limitations listed in "Signal Requirements and Limitations"
- The Simulink Coder limitations listed in "Code Generation Limitations for Model Reference" on page 6-22

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way a top model does. By default, every model in a hierarchy has its own configuration set, which it uses in the same way that it would if the model executed independently.

Because each model can have its own configuration set, configuration parameter values can be different in different models. Furthermore, some parameter values are intrinsically incompatible with model referencing. The response of the code generator to an inconsistent or unusable configuration parameter depends on the parameter:

- Where a parameter inconsistency has no significance, the code generator ignores or resolves the inconsistency without posting a warning.
- Where a nontrivial and possibly acceptable solution exists to an inconsistent parameter, the code generator resolves the conflict silently; resolves it with a warning; or generates an error.
- If an acceptable resolution is not possible, the code generator produces an error. You must then change parameter values to eliminate the issue.

When a model reference hierarchy contains many referenced models that have incompatible parameter values, or a changed parameter value must propagate to many referenced models, manually eliminating configuration parameter incompatibilities can be time-consuming. You can control or eliminate such overhead by using configuration references to assign an externally stored configuration set to multiple models. For more information, see "Share a Configuration with Multiple Models".

To select a code generation target for a model reference hierarchy without modifying the individual models, use a configuration reference in the Simulink data dictionary. For more information, see "Use Configuration Reference to Select Code Generation Target" on page 48-11.

The following tables list configuration parameters that can cause problems if set in certain ways, or if set differently in a referenced model than in a parent model. Where possible, the code generator resolves violations of these requirements, but most cases require you to make changes to parameters in your models.

Configuration Requirements for Model Referencing with System Target Files

Dialog Box Pane	Option	Requirement
Solver	**Start time**	Some system target files require the start time of models to be zero.
Hardware Implementation	All options	Values must be the same for top and referenced models.
Code Generation	**System target file**	Must be the same for top and referenced models.
	Language	Must be the same for top and referenced models.
	Generate code only	Must be the same for top and referenced models.
Identifiers	**Maximum identifier length**	Cannot be longer for a referenced model than for its parent model.
Interface	**Code replacement library**	Must be the same for top and referenced models.
	ASAP2 interface	Can be `on` or `off` in a top model, but must be `off` in a referenced model. If it is not `off`, the code generator temporarily sets it to `off` during code generation.

Configuration Requirements for Model Referencing with ERT System Target Files (Requires Embedded Coder)

Dialog Box Pane	Option	Requirement
Identifiers	**Global variables** **Global types** **Subsystem methods** **Local temporary variables** **Constant macros**	$R token must be included.
	Signal naming	Must be the same for top and referenced models.
	M-function	If specified, must be the same for top and referenced models.
	Parameter naming	Must be the same for top and referenced models.
	#define naming	Must be the same for top and referenced models.
Interface	**Ignore custom storage classes**	Must be the same for top and referenced models.
	Support floating-point numbers	Must be the same for top and referenced models.
	Support non-finite numbers	If `off` for top model, must be `off` for referenced models.
	Support complex numbers	If `off` for top model, must be `off` for referenced models.
	Remove error status field in real-time model data structure	If `on` for top model, must be `on` for referenced models.
Code Placement	**Use owner from data object for data definition placement**	Must be the same for top and referenced models.
	Signal display level	Must be the same for top and referenced models.
	Parameter tune level	Must be the same for top and referenced models.

Configuration Parameters Changed During Code Generation

For referenced models, if these **Configuration Parameters** > **Code Generation** > **Identifiers** parameters have settings that do not contain a $R token (which represents the name of the referenced model), code generation prepends the $R token to the identifier format:

- **Global variables** (`CustomSymbolStrGlobalVar`)
- **Global types** (`CustomSymbolStrType`)
- **Subsystem methods** (`CustomSymbolStrFcn`)

- **Constant macros** (`CustomSymbolStrMacro`)

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

1. In the Simulink Editor, from the **Modeling** tab, select **Model Advisor**.
2. Expand **By Task** > **Model Referencing**.
3. Run the **Check code generation identifier formats used for model reference** check.

If a script that operates on generated code uses identifier formats that code generation changes, then update the script to use the updated identifier format (which includes an appended `$R` token).

For more information about identifiers, see “Identifier Format Control” on page 55-17.

Naming Requirements

Within a model that uses model referencing, names of the constituent models cannot collide. When you generate code from a model that uses model referencing, the **Maximum identifier length** parameter must be large enough to accommodate the root model name and the name-mangling text. A code generation error occurs if **Maximum identifier length** is not large enough.

When a name conflict occurs between a symbol within the scope of a higher-level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. The code generator performs name mangling on the symbol from the higher-level model.

Embedded Coder Naming Requirements

The Embedded Coder product lets you control the formatting of generated symbols in much greater detail. When generating code with an ERT target from a model that uses model referencing:

- The `$R` token (in addition to the `$M` token) must be included in the **Identifier format control** parameter specifications except for **Shared utilities identifier format**.
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.

For more information, see “Model Configuration Parameters: Code Generation Identifiers”.

Custom Target Requirements

If you have Embedded Coder, a custom target must meet various requirements to support model referencing. For details, see “Support Model Referencing” on page 94-61.

Miscellaneous Considerations

Minimize occurrences of algebraic loops by selecting the **Minimize algebraic loop occurrences** parameter on the **Model Referencing** pane. The setting of this option affects only generation of code from the model. For more information about direct feed through, see “Algebraic Loop Concepts”.

Use the **Integer rounding mode** parameter on your model's blocks to simulate the rounding behavior of the C compiler that you intend to use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product, Matrix Multiply and n-D Lookup Table blocks.

For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the Simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see “Rounding” (Fixed-Point Designer).

See Also

Related Examples

- “Manage Configuration Sets for a Model”
- “Set Configuration Parameters for Model Hierarchies”

Convert Subsystem to Referenced Model and Generate Code

Converting subsystems to referenced models enhances incremental code generation, modular development of referenced models, and code reuse. Many large models use a combination of referenced models and subsystems.

To convert a Subsystem block to a Model block, use any of these methods:

- In the Simulink Editor, select a Subsystem block. On the Simulink Toolstrip, the **Subsystem Block** tab, select **Convert** > **Model Block**.
- In the MATLAB Command Window, use the `Simulink.SubSystem.convertToModelReference` function with the `'UseConversionAdvisor'` argument set to `true`.

Before you convert a subsystem to a referenced model:

1. Configure the Subsystem block as an atomic subsystem. In the Subsystem Block Parameters, select the **Treat as atomic unit** property.

   An atomic subsystem executes as a unit relative to the parent model. Subsystem execution does not interleave with parent block execution. This property makes it possible to extract subsystems for use as standalone models and as functions in the generated code.
2. Set the model configuration parameter **Signal resolution** to `Explicit only` or `None`.

You can map a storage class to a category of model data elements or to an individual element by using the Code Mappings editor or the Code Mappings API (`coder.mapping.api.CodeMapping`). Then, use the Code Mappings editor or the API to specify code configurations settings specific for that storage class, such as header files, definition files, and memory sections. If your model contains code mappings, you can migrate the code mappings when you convert a subsystem to a referenced model. For more information, see **Code Mappings Editor – C**.

Configure Code Mappings

1. Open the example model `RollAxisAutopilot`.

   ```
   openExample('RollAxisAutopilot')
   ```
2. On the Simulink toolstrip, the **Apps** tab, select the **Embedded Coder** or **Simulink Coder** app.
3. In the **C Code** tab, click **Code Interface** > **Default Code Mappings**. This option pins the Code Mappings editor.
4. Map a storage class to a category of model data element. In the Code Mappings editor, select the **Data Defaults** tab. For the Inports category, specify the **Storage Class** as `ExportedGlobal`.

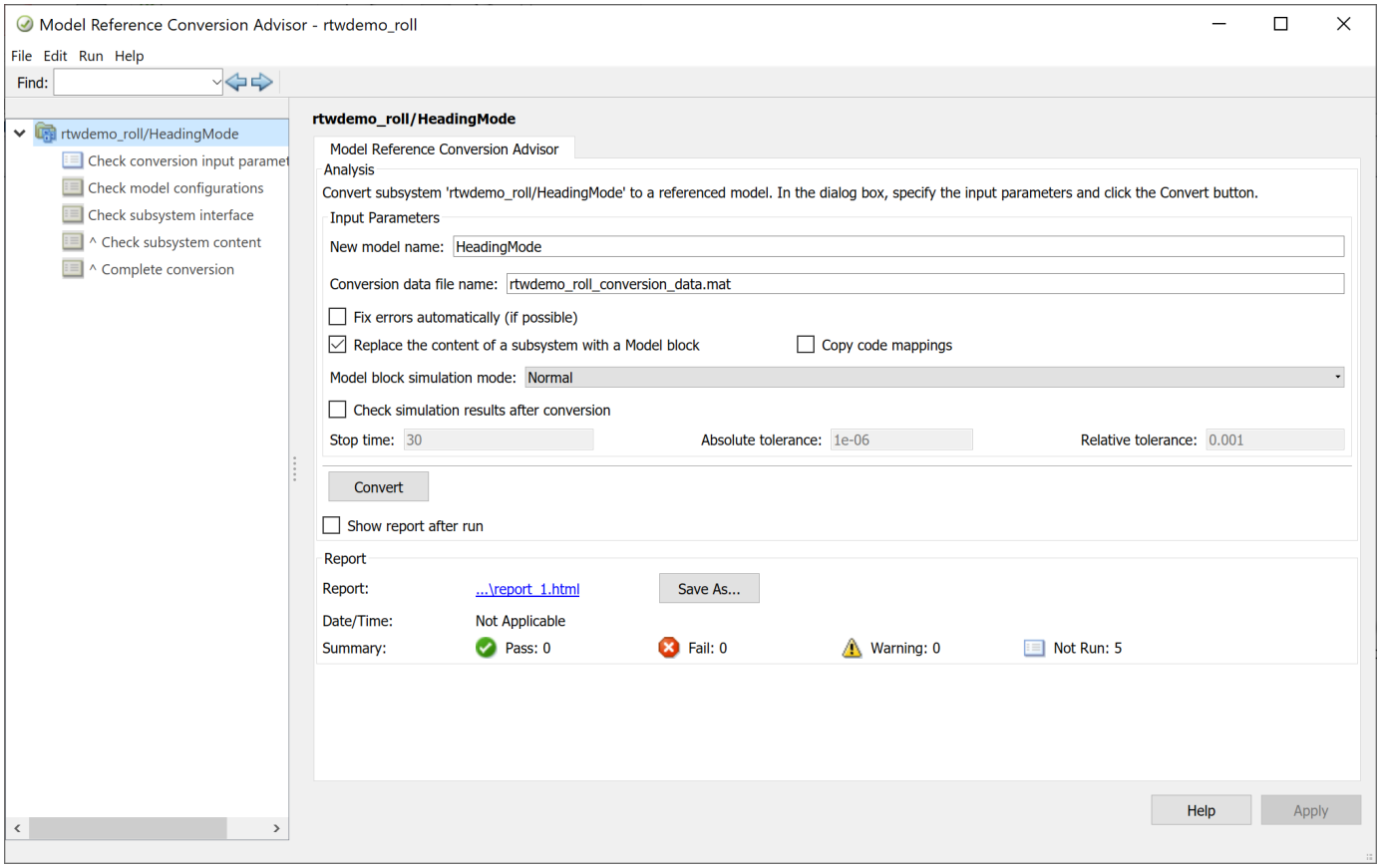
Alternatively, to configure the default mapping for Inports, in the MATLAB Command Window, enter:

```
roll_mapping = coder.mapping.api.get('RollAxisAutopilot');
setDataDefault(roll_mapping,'Inports','StorageClass','ExportedGlobal');
```

Convert Subsystem to Referenced Model

1. To convert a subsystem to a referenced model, open the **Model Reference Conversion Advisor**. Select the subsystem `HeadingMode`. On the **Subsystem Block** tab, select **Convert** > **Model Block**.

The Model Reference Conversion Advisor enables you to interactively specify conversion parameters and fix issues that the advisor finds.



2. Review the autopopulated model name. If needed, you can update this name.
3. To make the conversion process faster, select the **Fix errors automatically (if possible)**. This option fixes some conversion issues. You do not control the fixes.
4. To copy the code mappings specified for the model, select the **Copy code mappings** option. This option migrates the code mappings from the parent model to the newly converted referenced model. By default, this option is cleared. This option is not supported for models configured with an ERT-based system target file and service code interface.
5. Click **Convert**.

   The Model Reference Conversion Advisor steps through each of the checks. Address any issues that the advisor reports. After the Model Reference Conversion Advisor successfully completes the conversion, it replaces the Subsystem block with a Model block that references the new model.

Alternatively, to convert a subsystem to a referenced model, in the MATLAB Command Window, enter:

```
Simulink.SubSystem.convertToModelReference(...
   'RollAxisAutopilot/HeadingMode', ...
   'HeadingMode', ...
   'AutoFix',true,...
   'ReplaceSubsystem',true,...
   'CopyCodeMappings',true);
```

Copy Code Mappings After Subsystem Conversion

If you want to copy the code mappings from the parent model to the referenced model and you did not previously select the **Copy code mappings** option in the Model Reference Conversion Advisor, in the MATLAB Command Window, enter:

```
roll_mapping = coder.mapping.api.get('RollAxisAutopilot');
roll_sc = getDataDefault(roll_mapping,'Inports','StorageClass');

mode_mapping = coder.mapping.api.get('HeadingMode');
setDataDefault(mode_mapping,'Inports','StorageClass',roll_sc);
```

Review Generated Code

For the `RollAxisAutopilot`, model which contains the referenced model `HeadingMode`, open the **C Code** tab. To generate code for the model reference hierarchy, click **Build**.

The Simulink Coder build process generates and compiles code. The current folder now contains new files and a new folder.

Folder or File	Purpose
`RollAxisAutopilot_grt_rtw`	The build folder containing the generated code for the top model.
`slprj` > `grt` > `Heading Mode`	The build folder containing the generated code for the referenced model.
`RollAxisAutopilot.exe`	The executable created by the build process.
`RollAxisAutopilot.slxc`	The Simulink cache file for the top model.
`HeadingMode.slxc`	The Simulink cache file for the referenced model.

See Also

Related Examples

- “Generate Code for Model Reference Hierarchy” on page 6-2
- “Convert Subsystems to Referenced Models”

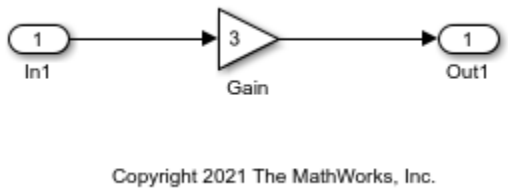
Generate Reusable Code from Model Reference Hierarchy

A reusable subsystem function that has input or output connected to the root Inport or Outport block of a referenced model can impact code reuse. It can prevent you from reusing atomic subsystems in a reference model context in the same way you might reuse the subsystems in a standalone model.

This example shows how to generate reusable code when you have a reusable subsystem referenced inside Model block.

Generate Reusable Code for Subsystem

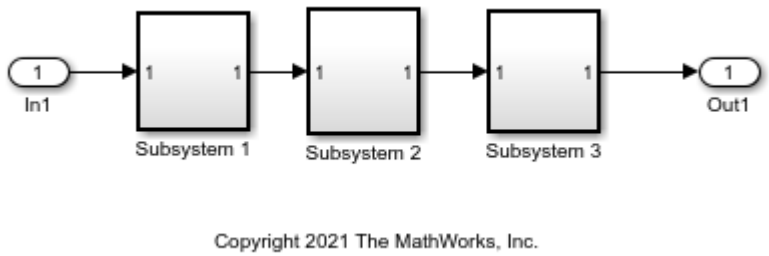
1. Convert these blocks into a subsystem. Drag a box to outline the subsystem that you want to create. From the **Multiple** tab on the Simulink Editor toolstrip, select **Create Subsystem**.



2. Make these changes to the Subsystem block parameters:

- Select “Treat as atomic unit”.
- On the **Code Generation** tab, set “Function packaging” to `Reusable function`.

3. Create a model that includes three instances of the preceding subsystem.



4. Set the model configuration parameter Default parameter behavior to `Inlined`.

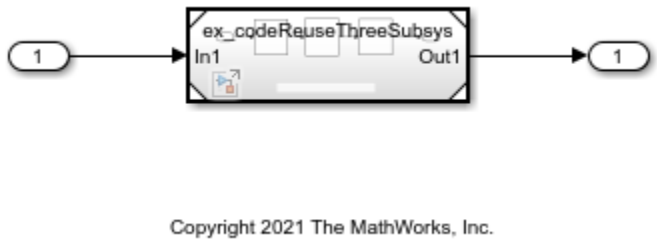
5. To build the model and generate code, press **Ctrl+B**.

The code generator optimizes the code by generating one copy of the function for the reused subsystem in the `model.c` file.

```
void ex_codeReuseThreeSubsys_Subsystem1(real_T rtu_In1,
  B_Subsystem1_ex_codeReuseThreeSubsys_T *localB)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain = 3.0 * rtu_In1;
}
```

Generate Reusable Code for Subsystem in Referenced Model

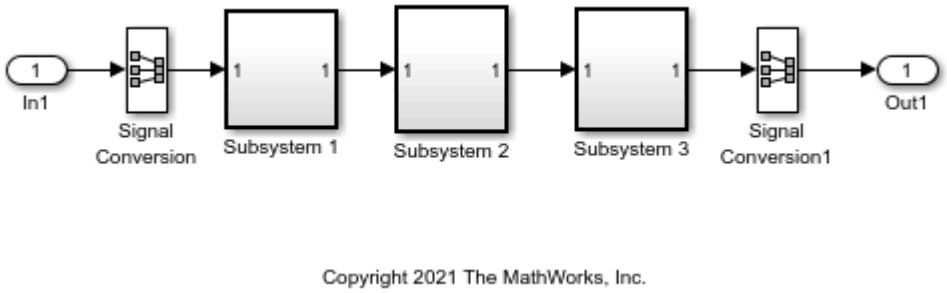
1. Create a top model by using a Model block that references the preceding model that has three instances of the reusable subsystem.



2. Generate code for the top model. The code generator generates three different function signatures in the `slprj` folder.

```
/* Output and update for atomic system: '<Root>/Subsystem 1' */
void ex_codeReuseThreeSubsys_Subsystem1(const real_T *rtu_In1,
  B_Subsystem1_ex_codeReuseThreeSubsys_T *localB)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain = 3.0 * *rtu_In1;
}

/* Output and update for atomic system: '<Root>/Subsystem 2' */
void ex_codeReuseThreeSubsys_Subsystem2(real_T rtu_In1,
  B_Subsystem2_ex_codeReuseThreeSubsys_T *localB)
{
  /* Gain: '<S2>/Gain' */
  localB->Gain = 3.0 * rtu_In1;
}

/* Output and update for atomic system: '<Root>/Subsystem 3' */
void ex_codeReuseThreeSubsys_Subsystem3(real_T rtu_In1, real_T *rty_Out1)
{
  /* Gain: '<S3>/Gain' */
  *rty_Out1 = 3.0 * rtu_In1;
}
```

3. To enable the subsystem functions for code reuse, insert Signal Conversion blocks into the referenced model. Place one block between the Inport and Subsystem 1 and another block between Subsystem 3 and the Outport.



If the subsystem has a Merge block with initial conditions, do not add a Signal Conversion block. Instead, add a Bias block to obtain a reusable function.

4. Select the model configuration parameter Pass fixed-size scalar root inputs by value for code generation.

5. Generate code for the top model. The code generator generates a single reusable function in the `slprj` folder.

```
void ex_codeReuseSubsysSigConvert_Subsystem1(real_T rtu_In1,
  B_Subsystem1_ex_codeReuseSubsysSigConvert_T *localB,
  P_Subsystem1_ex_codeReuseSubsysSigConvert_T *localP)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain = localP->Gain_Gain * rtu_In1;
}
```

See Also

Related Examples

- “Convert Subsystems to Referenced Models”
- “Reference Existing Models”
- “Generate Reusable Code from Library Subsystems Shared Across Models”

Storage Classes for Parameters and Signals Used in Model Blocks

To transfer data between your referenced model and top model, use signals and parameters. You can also use parameters to integrate external data into the model reference hierarchy. To include one model inside another model and create a model reference hierarchy, use a Model block.

You can apply a storage class to a data item in a model, such as signal or parameter, to:

- Make the data item appear in the generated code as a global symbol, typically a global variable.
- Prevent optimizations such as **Default parameter behavior** and **Signal storage reuse** from eliminating the data item from the generated code.

For examples, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

You can map a storage class to a category of model data elements or to an individual element by using the Code Mappings editor or the Code Mappings API (`coder.mapping.api.CodeMapping`). You can specify code configurations settings specific to that storage class, such as header files, definition files, and memory sections using the Code Mappings editor or the API. For more information, see **Code Mappings Editor – C**.

Storage Classes for Parameters in Model Blocks

Models containing Model blocks can use storage classes for parameters to interface with external code. Storage classes, with the exception of `Auto`, are tunable. The supported storage classes include:

- Predefined storage classes in Simulink Coder and Embedded Coder
- A new storage class created by using the Embedded Coder Dictionary

For limitations on Embedded Coder storage classes, see “Storage Class Limitations”.

The restrictions on parameters in referenced models are:

- Tunable parameters are not supported for noninlined S-functions.
- Tunable parameters set through the Model Parameter Configuration dialog box are ignored.

Concerning how global tunable parameters are declared, defined, and used in code generated for targets. consider:

- A global tunable parameter is a parameter in the base workspace that has a storage class other than `Auto`.
- An `extern` declaration is generated for models that use a given parameter.
- If a parameter is exported, the top model defines (allocates memory for) the parameter whether the top model uses the parameter or not.
- Global parameters are accessed directly as global memory. They are not passed as arguments to the functions that are generated for the referenced models.
- In a referenced model that sets the default storage class for a category of parameter data to `Default` in the Code Mapping editor, symbols for `Model default` parameters are generated by using unstructured variables (`rtP_xxx`) instead of being written into the *`model`*`_P` structure. You can then compile each referenced model independently.

Parameters that you use as Model block arguments must be defined in the referenced model workspace. For details, see “Parameterize Instances of a Reusable Referenced Model”.

Storage Classes for Signals in Model Blocks

Models containing Model blocks can use signals of storage class `Auto`.

A global signal is a signal that has a storage class other than `Auto`. The supported storage classes include:

- Predefined storage classes in Simulink Coder and Embedded Coder
- A new storage class created by using Embedded Coder Dictionary

If you have Embedded Coder, you can create a global signal by using a storage class defined in the Embedded Coder Dictionary that has **Storage Type** set as `Unstructured` or by creating a storage in the Embedded Coder Dictionary that has **Storage Type** set as `Unstructured`.

These signals are distinct from individual signals that use the storage class `Model default`. When you set the default storage class of the corresponding data category to `Default` in the Code Mapping editor, these signals are treated as test points that have the `Auto` storage class.

When you declare signals as global, be aware of how the signal data is handled. Global signals are declared, defined, and used as follows:

- An `extern` declaration is generated for models that use a global signal.

  As a result, if a signal crosses a Model block boundary, the top model and the referenced model both generate `extern` declarations for the signal.
- For an exported signal, the top model defines (allocates memory for) the signal, whether or not the top model itself uses the signal.
- Global signals used by a referenced model are accessed directly as global memory. They are not passed as arguments to the functions that are generated for the referenced models.

Storage classes that you create by using the Embedded Coder Dictionary also follow the preceding rules for the top model.

Note In a referenced model, you cannot use the storage class that you create by using the Embedded Coder Dictionary on root-level Inports and Outports.

Certain storage classes have limited support for use with model reference. See “Storage Class Limitations”.

Signal Name Mismatches Across Model Reference Boundary

Within a parent model, the name and storage class for a signal entering or leaving a Model block might not match those of the signal attached to the root inport or outport within that referenced model. Because referenced models are compiled independently without regard to a parent model, they cannot adapt to the possible variations in how parent models label and store signals.

The code generator accepts cases where input and output signals in a referenced model have the `Auto` storage class. When such signals are test pointed or are global, certain restrictions apply. This

table describes how mismatches in signal labels and storage classes between parent and referenced models are handled by the software.

Relationships of Signals and Storage Classes Across Model Reference Boundary

Storage Class in Referenced Model	Storage Class in Parent Model	Signal Passing Method	Signal Mismatch Checking
`Auto`	`Non-Auto`	Function argument	**Signal label mismatch**
`Model default` (when the Code Mapping editor specifies `Default` storage class) or resolved to Signal Object	`Non-Auto`	Function argument	**Signal label mismatch**
`Non-Auto`	`Auto`	Global variable	None
`Non-Auto`	`Non-Auto`	Global variable	One of the storage classes must be `Auto` (otherwise there is an error)

These signal resolution rules apply to code generation:

- If the storage class of a root input or output signal in a referenced model is `Auto` or `Model default` (when you set the storage class of the corresponding data category to `Default` in the Code Mapping editor), the signal is passed as a function argument. When such a signal is `Non-Auto` or resolves to a `Simulink.Signal` object in the parent model, the **Signal label mismatch** diagnostic is applied.
- If a root input or output signal in a referenced model is global, the signal is communicated by using direct memory access (global variable). The corresponding signal in the parent model must be `Auto`.
- You cannot use a storage class that you create by using the Embedded Coder Dictionary in a referenced model.

You can set the model configuration parameter **Signal label mismatch** diagnostic to `error`, `warning`, or `none`.

See Also

"Signal label mismatch"

Related Examples

- "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
- "C Code Generation Configuration for Model Interface Elements"
- "Establish Data Ownership in a Model Hierarchy" on page 37-10
- "Flexible Storage Class for Different Model Hierarchy Contexts" on page 32-34

Code Generation Limitations for Model Reference

Code generation limitations for model references influence how you properly set up the model reference hierarchy for code generation.

In addition to these limitations, a model reference hierarchy used for code generation must satisfy:

- The Simulink requirements and limitations listed in:
  - “Configuration Requirements for All Referenced Model Simulation”
  - “Model Reference Requirements and Limitations”
  - “Model Reference Interface and Boundary”
- The Simulink Coder requirements applicable to the code generation target, as listed in “Configuration Parameter Requirements” on page 6-8.

Storage Class Limitations

If you have Embedded Coder, some restrictions exist on grouped storage classes in referenced models. For details, see “Storage Class Limitations”.

Data Logging Limitations

- To Workspace blocks and Scope blocks are ignored when Simulink Coder generates code for a referenced model. The resulting code is the same as if the blocks did not exist. In some cases for models with Scope block and MAT-file logging enabled, the code generator produces an extra PWork in the model's DWork structure.
- Code generated for referenced models cannot log data to MAT-files. If you enable data logging for a referenced model, Simulink Coder disables the option before code generation and re-enables it afterward.
- If you log states for a model that contains referenced models, the ordering of the states in the output is determined by block sorted order, and might not match between simulation output and generated code MAT-file logging output.

State Initialization Limitations

When a top model uses the model configuration parameter **Initial state** to specify initial conditions, Simulink Coder does not initialize the discrete states of the referenced models during code generation.

Model Reuse Limitations

If a referenced model used for code generation has at least one of the following characteristics, the model must specify the model configuration parameter **Total number of instances allowed per top model** as `One`. Other instances of the model can exist in the hierarchy. If you do not set the parameter to `One`, or more than one instance of the model exists in the hierarchy, an error occurs. The characteristics are:

- The model references another model that has been set to a single instance.
- The model contains an internal signal or state that has a storage class that is not supported for multi-instance models. You must set the storage class to `Auto` or `Model default` for internal

signals and states. The default storage class for internal data must be a multi-instance storage class.

- The model uses exported Stateflow® graphical functions.
- The model contains an S-function that is:
  - Inlined but you have not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`
  - Not inlined
- The model contains a function-call subsystem that:
  - The Simulink engine forces to be a function
  - Is called by a wide signal

S-Functions Limitations

- A referenced model cannot use noninlined S-functions generated by Simulink Coder.
- The Simulink Coder S-function target does not support model referencing.

For additional information, see “S-Functions in Referenced Models”.

Subsystems Limitations

If you generate code for an atomic subsystem as a reusable function, inputs or outputs that connect the subsystem to a referenced model might prevent code reuse, as described in “Generate Reentrant Code from Subsystems”.

Targets Limitation

The Simulink Coder S-function target does not support model referencing.

Conditional Models Limitation

When a model contains a trigger or enable port, you cannot generate standalone Simulink Coder code.

Local Solver Limitation

You can not generate code for a model that uses a local solver. For more information on local solvers, see “Use Local Solvers in Referenced Models”.

See Also

Related Examples

- “Set Configuration Parameters for Code Generation of Model Hierarchies” on page 6-8
- “Set Configuration Parameters for Model Hierarchies”

Simulink Cache Files for Incremental Code Generation

Share the generated code for various target types between team members inside an organization by using Simulink cache files. When you generate code from a Simulink model hierarchy, each referenced model has an associated Simulink cache file. These files have an `.slxc` file extension. The cache files contain artifacts from the build that can reduce the time required for successive model simulation and code generation. Simulink cache files are generated when you simulate models in accelerator or rapid accelerator mode or generate code for models. For more information, see “Share Simulink Cache Files for Faster Simulation”.

Contents of Simulink Cache Files

At the end of a build, the code generator packages the artifacts and generated code for the model hierarchy in the Simulink cache files. The cached artifacts include:

- Shared utilities
- All files under the `slprj\`*`system_target_file`*`\`*`model_name`*
- *`model_name`*`_rtw_`*`system_target_file`* folder for the top model
- Any binaries generated for the top model

You cannot modify the contents of the Simulink cache files. If you update the build artifacts inside the `slprj` folder, the Simulink cache file picks up these updates during the next build cycle.

Share Code Generation Artifacts

If you are involved in team development of large models that reference other models, use Simulink cache files to synchronize builds and share generated code artifacts to enhance incremental code generation. You can leverage Simulink cache files to avoid unnecessary code generation.

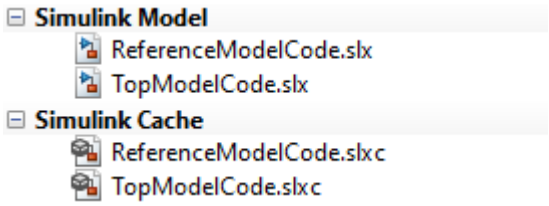
For example, assume that you collaborate with your team members by using a source control system and your models are built every night in a continuous integration system.

1. Open your model. For example, `TopModelCode`.

```
model = 'TopModelCode';
openExample('simulinkcoder/FilePackagingModelsCodeAndDataExample', ...
 'supportingFile', model);
```

2. Build the model. On the **C Code** tab, click **Build**.

   Observe the Simulink cache files generated for all referenced models. The `ReferenceModelCode` model is a referenced model in `TopModelCode`.



   Simulink stores cache files in the folder specified by the “Simulation cache folder” preference.

   To see all supported configurations, double-click the cache file. The Simulink cache report contains information related to each release, platform, and list of packaged targets.

Simulink cache for TopModelCode

This Simulink cache contains derived files for the following releases and platforms:

R2022b : win64

Code generation

Target	Build Type	Folder Structure
ert	Top model	Model specific
grt	Top model	Model specific

Alternatively, to get the contents of the Simulink cache file in a MATLAB table, use the `slxcinfo` function.

3. Submit your models to the source control system. The continuous integration system builds the models overnight and updates the existing Simulink cache files.

   Note Because Simulink cache files are derived files, do not submit them to source control. If you share Simulink cache files by storing them in a source control system, you cannot diff or merge different versions of these files.

4. Assume your team member made some changes to the model and submitted the changes to the source control system. When you want to work with the model hierarchy, fetch these files:
   - Design files (model and associated files) from the source control system.
   - Simulink cache files from the continuous integration system.
5. Make your changes to the model.
6. Build the model. The code generator extracts code generation artifacts stored in the Simulink cache files from the overnight build and avoids unnecessary rebuilds and code regeneration.

   The time to build the model and generate code is less than the first time that you built the model.

   Note To unpack the simulation and code generation targets from the Simulink cache files without updating, simulating, or generating code for the model hierarchy, use the `slxcunpack` function.

7. Submit your updated model files to the source control system.

See Also

`slxcinfo` | `slxcunpack`

Related Examples

- "Share Simulink Cache Files for Faster Simulation"
- "Reduce Build Time for Referenced Models by Using Parallel Builds" on page 59-51

More About

- “Pull, Push, and Fetch Files with Git”

External Websites

- Simulink Cache (1 min, 27 sec)

Specify Instance-Specific Parameter Values for Reusable Referenced Model

When you use model referencing to break a large system into components, each component is a separate model. You can reuse a component by referring to it with multiple Model blocks. Each Model block is an instance of the component. You can then configure a block parameter (such as the **Gain** parameter of a Gain block) to use either the same value or a different value for each instance of the component. To use different values, create and use a model argument to set the value of the block parameter.

When you generate code from a model hierarchy that uses model arguments, the arguments appear in the code as formal parameters of the referenced model entry-point functions, such as the output (`step`) function. The generated code then passes the instance-specific parameter values, which you specify in each Model block, to the corresponding function calls.

Whether you use or do not use model arguments, you can use storage classes to configure block parameters to appear in the generated code as tunable global variables. You can also use storage classes to generate tunable model argument values, which the generated code stores in memory and passes to the function calls. You can then change the values during execution.

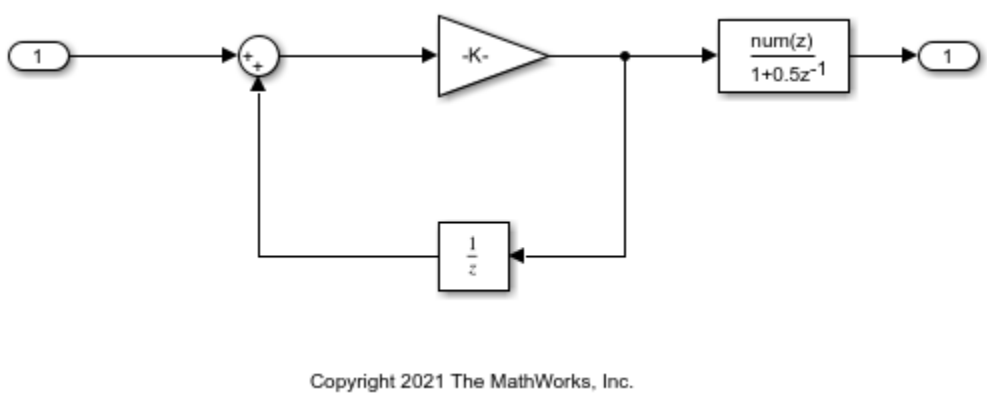
Pass Parameter Data to Referenced Model Entry-Point Functions as Arguments

Configure a referenced model to accept parameter data through formal parameters of the generated model entry-point function. This technique enables you to specify a different parameter value for each instance (Model block) of the referenced model.

Configure Referenced Model to Use Model Arguments

Open the model `ex_arg_code_ref`. This model represents a reusable algorithm.

```
open_system('ex_arg_code_ref')
```



On the **Modeling** tab, click **Model Data Editor**.

In the Model Data editor, in the data table, use the **Data Type** column to set the data type of the Inport block to `single`. Due to data type inheritance, the other signals in the model use the same data type.

Select the **Parameters** tab.

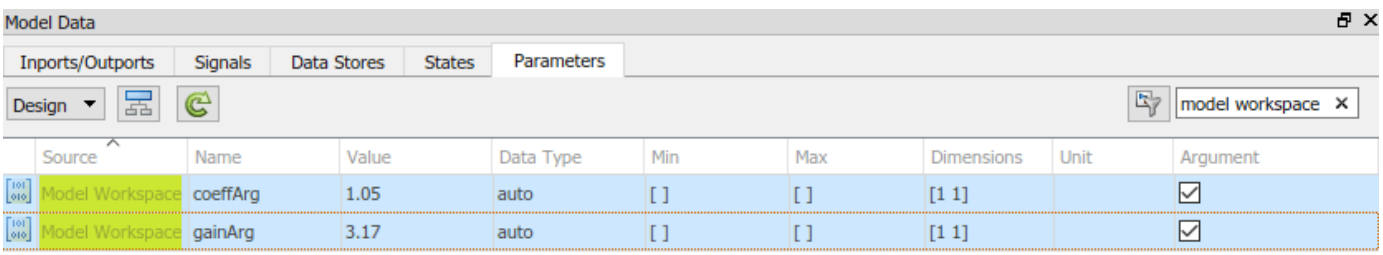
In the model, select the Gain block.

In the Model Data Editor, use the **Value** column to set the value of the **Gain** parameter to `gainArg`. `gainArg` is a `Simulink.Parameter` object in the model workspace whose value is `3.17`.

Use the Model Data Editor to set the **Numerator** parameter to the `Simulink.Parameter` object `coeffArg` whose value is `1.05`.

In the Model Data Editor, click the **Show/refresh additional information** button.

Use the **Filter contents** box to find each parameter object (`gainArg` and `coeffArg`). For each object, select the check box in the **Argument** column.



Save the `ex_arg_code_ref` model.

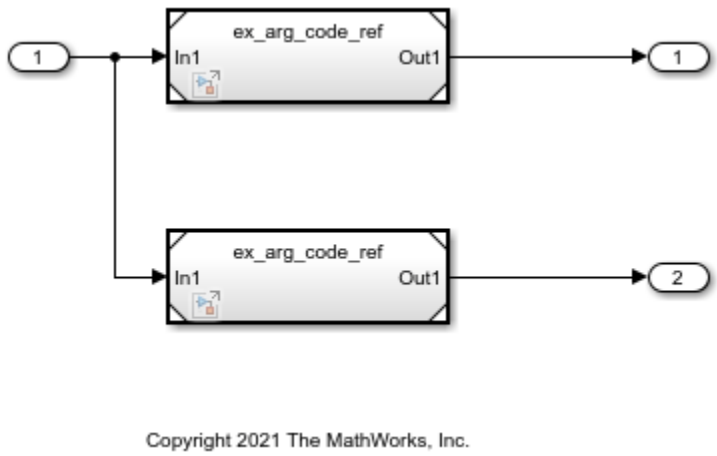
Alternatively, at the command prompt, you can use these commands to configure the blocks and the parameter objects:

```
set_param('ex_arg_code_ref/In1','OutDataTypeStr','single')

set_param('ex_arg_code_ref/Gain','Gain','gainArg')
modelWorkspace = get_param('ex_arg_code_ref','ModelWorkspace');
assignin(modelWorkspace,'gainArg',Simulink.Parameter(3.17));

set_param('ex_arg_code_ref/Discrete Filter','Numerator','coeffArg')
assignin(modelWorkspace,'coeffArg',Simulink.Parameter(1.05));

set_param('ex_arg_code_ref','ParameterArgumentNames','coeffArg,gainArg')

save_system('ex_arg_code_ref')
```

Specify Instance-Specific Parameter Values in Model Blocks

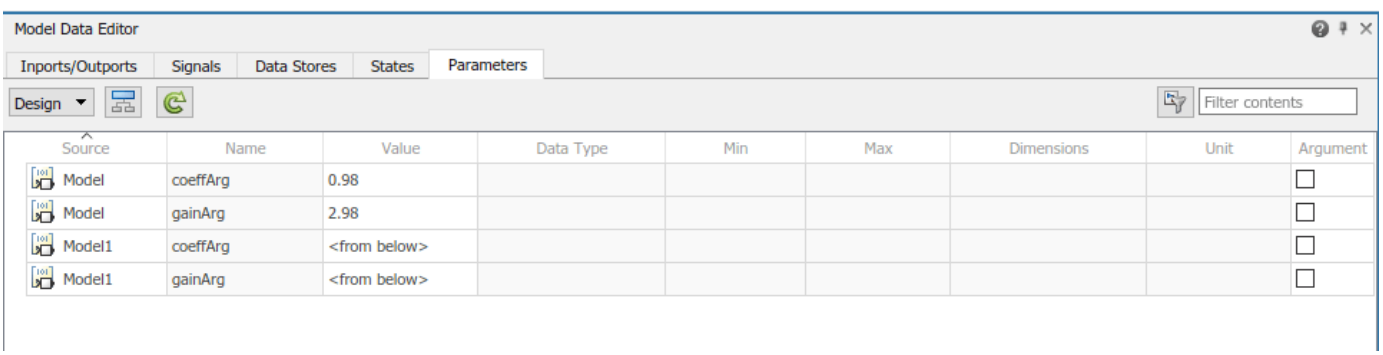
Open the model `ex_arg_code`. This model uses multiple instances (Model blocks) of the reusable algorithm.

```
open_system('ex_arg_code')
```



In the model, open the Model Data Editor **Parameters** tab (**Modeling > Model Data Editor**). The Model Data Editor shows four rows that correspond to the model arguments, `coeffArg` and `gainArg`, that you can specify for the two Model blocks.

Use the Model Data Editor to set values for the model arguments in `Model`. For example, use the values in the figure. For `Model1`, do not specify a value for the model arguments. By default, a model argument uses the last value specified below it in the model hierarchy (indicated by the value `from below`).



Alternatively, at the command prompt, you can use this command to set the values for `Model`:

```
instSpecParams = get_param('ex_arg_code/Model','InstanceParameters');
instSpecParams(1).Value = '.98';
instSpecParams(2).Value = '2.98';
set_param('ex_arg_code/Model','InstanceParameters',instSpecParams);
```

Generate code from the top model.

```
slbuild('ex_arg_code')
```

```
### Starting serial model reference code generation build.
### Successfully updated the model reference code generation target for: ex_arg_code_ref
### Starting build procedure for: ex_arg_code
### Successful completion of build procedure for: ex_arg_code

Build Summary
```

```
Code generation targets built:

Model            Action                        Rebuild Reason
==================================================================================
ex_arg_code_ref  Code generated and compiled.  ex_arg_code_ref.c does not exist.

Top model targets built:

Model        Action                        Rebuild Reason
==========================================================================================
ex_arg_code  Code generated and compiled.  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.148s
```

The file `ex_arg_code_ref.c` defines the referenced model entry-point function, `ex_arg_code_ref`. The function has two formal parameters, `rtp_coeffArg` and `rtp_gainArg`, that correspond to the model arguments, `coeffArg` and `gainArg`. The formal parameters use the data type `real32_T`, which corresponds to the data type `single` in Simulink®.

```
file = fullfile('slprj','grt','ex_arg_code_ref','ex_arg_code_ref.c');
rtwdemodbtype(file,'/* Output and update for referenced model:',...
    'real32_T rtp_gainArg)',1,1)
```

```
/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1, real32_T
                     rtp_coeffArg, real32_T rtp_gainArg, DW_ex_arg_code_ref_f_T *
                     localDW)
{
  real32_T rtb_Sum;

  /* Sum: '<Root>/Sum' incorporates:
   *  UnitDelay: '<Root>/Unit Delay'
   */
  rtb_Sum = *rtu_In1 + localDW->UnitDelay_DSTATE;

  /* Gain: '<Root>/Gain' */
  rtb_Sum *= rtp_gainArg;

  /* DiscreteFilter: '<Root>/Discrete Filter' */
  localDW->DiscreteFilter_tmp = rtb_Sum - 0.5F * localDW->DiscreteFilter_states;
  *rty_Out1 = rtp_coeffArg * localDW->DiscreteFilter_tmp;

  /* Update for UnitDelay: '<Root>/Unit Delay' */
  localDW->UnitDelay_DSTATE = rtb_Sum;

  /* Update for DiscreteFilter: '<Root>/Discrete Filter' */
  localDW->DiscreteFilter_states = localDW->DiscreteFilter_tmp;
}

/* Model initialize function */
void ex_arg_code_ref_initialize(const char_T **rt_errorStatus,
  RT_MODEL_ex_arg_code_ref_T *const ex_arg_code_ref_M, DW_ex_arg_code_ref_f_T
  *localDW)
{
  /* Registration code */
```

```
  /* initialize error status */
  rtmSetErrorStatusPointer(ex_arg_code_ref_M, rt_errorStatus);

  /* states (dwork) */
  (void) memset((void *)localDW, 0,
                sizeof(DW_ex_arg_code_ref_f_T));
}
```

The file `ex_arg_code.c` contains the definition of the top model entry-point function, `ex_arg_code`. This function calls the referenced model entry-point function, `ex_arg_code_ref`, and uses the model argument values that you specified (such as `1.11` and `3.34`) as the values of `rtp_coeffArg` and `rtp_gainArg`.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file,'/* ModelReference: ''<Root>/Model'' incorporates:',...
    '1.05F, 3.17F)',1,1)
```

```
  /* ModelReference: '<Root>/Model' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out1'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1, 0.98F, 2.98F,
                  &(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* ModelReference: '<Root>/Model1' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out2'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2, 1.05F, 3.17F,
                  &(ex_arg_code_DW.Model1_InstanceData.rtdw));

  /* Matfile logging */
  rt_UpdateTXYLogVars(ex_arg_code_M->rtwLogInfo,
                      (&ex_arg_code_M->Timing.taskTime0));

  /* signal main to stop simulation */
  {                                   /* Sample time: [0.2s, 0.0s] */
    if ((rtmGetTFinal(ex_arg_code_M)!=-1) &&
        !((rtmGetTFinal(ex_arg_code_M)-ex_arg_code_M->Timing.taskTime0) >
          ex_arg_code_M->Timing.taskTime0 * (DBL_EPSILON))) {
      rtmSetErrorStatus(ex_arg_code_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++ex_arg_code_M->Timing.clockTick0)) {
    ++ex_arg_code_M->Timing.clockTickH0;
  }
```

```
  ex_arg_code_M->Timing.taskTime0 = ex_arg_code_M->Timing.clockTick0 *
    ex_arg_code_M->Timing.stepSize0 + ex_arg_code_M->Timing.clockTickH0 *
    ex_arg_code_M->Timing.stepSize0 * 4294967296.0;
}

/* Model initialize function */
void ex_arg_code_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)ex_arg_code_M, 0,
                sizeof(RT_MODEL_ex_arg_code_T));
  rtmSetTFinal(ex_arg_code_M, 10.0);
  ex_arg_code_M->Timing.stepSize0 = 0.2;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    ex_arg_code_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogT(ex_arg_code_M->rtwLogInfo, "tout");
    rtliSetLogX(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogXFinal(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(ex_arg_code_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogMaxRows(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogDecimation(ex_arg_code_M->rtwLogInfo, 1);

    /*
     * Set pointers to the data and signal info for each output
     */
    {
      static void * rt_LoggedOutputSignalPtrs[] = {
        &ex_arg_code_Y.Out1,
        &ex_arg_code_Y.Out2
      };

      rtliSetLogYSignalPtrs(ex_arg_code_M->rtwLogInfo, ((LogSignalPtrsType)
        rt_LoggedOutputSignalPtrs));
    }

    {
      static int_T rt_LoggedOutputWidths[] = {
        1,
        1
      };
```

```
static int_T rt_LoggedOutputNumDimensions[] = {
  1,
  1
};

static int_T rt_LoggedOutputDimensions[] = {
  1,
  1
};

static boolean_T rt_LoggedOutputIsVarDims[] = {
  0,
  0
};

static void* rt_LoggedCurrentSignalDimensions[] = {
  (NULL),
  (NULL)
};

static int_T rt_LoggedCurrentSignalDimensionsSize[] = {
  4,
  4
};

static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
  SS_SINGLE,
  SS_SINGLE
};

static int_T rt_LoggedOutputComplexSignals[] = {
  0,
  0
};

static RTWPreprocessingFcnPtr rt_LoggingPreprocessingFcnPtrs[] = {
  (NULL),
  (NULL)
};

static const char_T *rt_LoggedOutputLabels[] = {
  "",
  "" };

static const char_T *rt_LoggedOutputBlockNames[] = {
  "ex_arg_code/Out1",
  "ex_arg_code/Out2" };

static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
  { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 },

  { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 }
};

static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
  {
    2,
    rt_LoggedOutputWidths,
```

```
        rt_LoggedOutputNumDimensions,
        rt_LoggedOutputDimensions,
        rt_LoggedOutputIsVarDims,
        rt_LoggedCurrentSignalDimensions,
        rt_LoggedCurrentSignalDimensionsSize,
        rt_LoggedOutputDataTypeIds,
        rt_LoggedOutputComplexSignals,
        (NULL),
        rt_LoggingPreprocessingFcnPtrs,

        { rt_LoggedOutputLabels },
        (NULL),
        (NULL),
        (NULL),

        { rt_LoggedOutputBlockNames },

        { (NULL) },
        (NULL),
        rt_RTWLogDataTypeConvert
      }
    };

    rtliSetLogYSignalInfo(ex_arg_code_M->rtwLogInfo, rt_LoggedOutputSignalInfo);

    /* set currSigDims field */
    rt_LoggedCurrentSignalDimensions[0] = &rt_LoggedOutputWidths[0];
    rt_LoggedCurrentSignalDimensions[1] = &rt_LoggedOutputWidths[1];
  }

  rtliSetLogY(ex_arg_code_M->rtwLogInfo, "yout");
}

/* states (dwork) */
(void) memset((void *)&ex_arg_code_DW, 0,
              sizeof(DW_ex_arg_code_T));

/* external inputs */
ex_arg_code_U.In1 = 0.0F;

/* external outputs */
(void)memset(&ex_arg_code_Y, 0, sizeof(ExtY_ex_arg_code_T));

/* Model Initialize function for ModelReference Block: '<Root>/Model' */
ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
  &(ex_arg_code_DW.Model_InstanceData.rtm),
  &(ex_arg_code_DW.Model_InstanceData.rtdw));

/* Model Initialize function for ModelReference Block: '<Root>/Model1' */
ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
  &(ex_arg_code_DW.Model1_InstanceData.rtm),
  &(ex_arg_code_DW.Model1_InstanceData.rtdw));

/* Matfile logging */
rt_StartDataLoggingWithStartTime(ex_arg_code_M->rtwLogInfo, 0.0, rtmGetTFinal
  (ex_arg_code_M), ex_arg_code_M->Timing.stepSize0, (&rtmGetErrorStatus
  (ex_arg_code_M)));
```

```
  /* SystemInitialize for ModelReference: '<Root>/Model' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* SystemInitialize for ModelReference: '<Root>/Model1' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model1_InstanceData.rtdw));
}

/* Model terminate function */
void ex_arg_code_terminate(void)
{
  /* (no terminate code required) */
}
```

The formal parameters use the data type `real32_T` (`single`) because:

1. The block parameters in `ex_arg_code_ref` determine their data types through internal rules. For example, in the Gain block dialog box, on the **Parameter Attributes** tab, **Parameter data type** is set to `Inherit: Inherit via internal rule` (the default). In this case, the internal rule chooses the same data type as the input and output signals, `single`.
2. The model arguments in the model workspace use context-sensitive data typing because the value of the `DataType` property is set to `auto` (the default). With this setting, the model arguments use the same data type as the block parameters, `single`.
3. The formal parameters in the generated code use the same data type as the model arguments, `single`.

Generate Tunable Argument Values

You can configure the instance-specific values in the Model blocks to appear in the generated code as tunable global variables. This technique enables you to store the parameter values for each instance in memory and to tune the values during code execution.

In the top model `ex_arg_code`, select the Model Data Editor **Parameters** tab.

Use the Model Data Editor to set the values of the model arguments according to this figure.

Source	Name	Value
Model	coeffArg	coeffForInst1
Model	gainArg	gainForInst1
Model1	coeffArg	coeffForInst2
Model1	gainArg	gainForInst2

View the contents of the `ex_arg_code_ref` model workspace in Model Explorer.

Copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace to the base workspace.

Rename `gainArg` as `gainForInst1`. Rename `coeffArg` as `coeffForInst1`.

```
gainForInst1 = getVariable(modelWorkspace,'gainArg');
gainForInst1 = copy(gainForInst1);
coeffForInst1 = getVariable(modelWorkspace,'coeffArg');
coeffForInst1 = copy(coeffForInst1);
```

Copy `gainForInst1` and `coeffForInst1` as `gainForInst2` and `coeffForInst2`.

```
gainForInst2 = copy(gainForInst1);
coeffForInst2 = copy(coeffForInst1);
```

Set the instance-specific parameter values by using the `Value` property of the parameter objects in the base workspace.

```
gainForInst1.Value = 2.98;
coeffForInst1.Value = 0.98;

gainForInst2.Value = 3.34;
coeffForInst2.Value = 1.11;
```

For each new parameter object, on the **Code Generation** tab, set `StorageClass` to `ExportedGlobal`. This setting causes the parameter objects to appear in the generated code as tunable global variables.

```
gainForInst1.StorageClass = 'ExportedGlobal';
coeffForInst1.StorageClass = 'ExportedGlobal';
gainForInst2.StorageClass = 'ExportedGlobal';
coeffForInst2.StorageClass = 'ExportedGlobal';

instSpecParams = get_param('ex_arg_code/Model','InstanceParameters');
instSpecParams(1).Value = 'coeffForInst1';
instSpecParams(2).Value = 'gainForInst1';
instSpecParams1 = get_param('ex_arg_code/Model1','InstanceParameters');
instSpecParams1(1).Value = 'coeffForInst2';
instSpecParams1(2).Value = 'gainForInst2';
set_param('ex_arg_code/Model','InstanceParameters',instSpecParams);
set_param('ex_arg_code/Model1','InstanceParameters',instSpecParams1);
```

Generate code from the top model.

```
slbuild('ex_arg_code')
```

```
### Starting serial model reference code generation build.
### Model reference code generation target for ex_arg_code_ref is up to date.
### Starting build procedure for: ex_arg_code
### Successful completion of build procedure for: ex_arg_code

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================
ex_arg_code  Code generated and compiled.  Generated code was out of date.

1 of 2 models built (1 models already up to date)
Build duration: 0h 0m 15.865s
```

The file `ex_arg_code.c` defines the global variables that correspond to the parameter objects in the base workspace.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Block states (default storage) */',1,0)
```

```
/* Exported block parameters */
```

```
real32_T coeffForInst1 = 0.98F;        /* Variable: coeffForInst1
                                        * Referenced by: '<Root>/Model'
                                        */
real32_T coeffForInst2 = 1.11F;        /* Variable: coeffForInst2
                                        * Referenced by: '<Root>/Model1'
                                        */
real32_T gainForInst1 = 2.98F;         /* Variable: gainForInst1
                                        * Referenced by: '<Root>/Model'
                                        */
real32_T gainForInst2 = 3.34F;         /* Variable: gainForInst2
                                        * Referenced by: '<Root>/Model1'
                                        */
```

In each call to `ex_arg_code_ref`, the top model algorithm uses the global variables to set the values of the formal parameters.

```
rtwdemodbtype(file,'/* ModelReference: ''<Root>/Model'' incorporates:',...
    'gainForInst2);',1,1)
```

```
  /* ModelReference: '<Root>/Model' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out1'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1, coeffForInst1,
                  gainForInst1, &(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* ModelReference: '<Root>/Model1' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out2'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2, coeffForInst2,
                  gainForInst2, &(ex_arg_code_DW.Model1_InstanceData.rtdw));

  /* Matfile logging */
  rt_UpdateTXYLogVars(ex_arg_code_M->rtwLogInfo,
                      (&ex_arg_code_M->Timing.taskTime0));

  /* signal main to stop simulation */
  {                                     /* Sample time: [0.2s, 0.0s] */
    if ((rtmGetTFinal(ex_arg_code_M)!=-1) &&
        !((rtmGetTFinal(ex_arg_code_M)-ex_arg_code_M->Timing.taskTime0) >
          ex_arg_code_M->Timing.taskTime0 * (DBL_EPSILON))) {
      rtmSetErrorStatus(ex_arg_code_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++ex_arg_code_M->Timing.clockTick0)) {
    ++ex_arg_code_M->Timing.clockTickH0;
```

```
  }

  ex_arg_code_M->Timing.taskTime0 = ex_arg_code_M->Timing.clockTick0 *
    ex_arg_code_M->Timing.stepSize0 + ex_arg_code_M->Timing.clockTickH0 *
    ex_arg_code_M->Timing.stepSize0 * 4294967296.0;
}

/* Model initialize function */
void ex_arg_code_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)ex_arg_code_M, 0,
                sizeof(RT_MODEL_ex_arg_code_T));
  rtmSetTFinal(ex_arg_code_M, 10.0);
  ex_arg_code_M->Timing.stepSize0 = 0.2;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    ex_arg_code_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogT(ex_arg_code_M->rtwLogInfo, "tout");
    rtliSetLogX(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogXFinal(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(ex_arg_code_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogMaxRows(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogDecimation(ex_arg_code_M->rtwLogInfo, 1);

    /*
     * Set pointers to the data and signal info for each output
     */
    {
      static void * rt_LoggedOutputSignalPtrs[] = {
        &ex_arg_code_Y.Out1,
        &ex_arg_code_Y.Out2
      };

      rtliSetLogYSignalPtrs(ex_arg_code_M->rtwLogInfo, ((LogSignalPtrsType)
        rt_LoggedOutputSignalPtrs));
    }

    {
      static int_T rt_LoggedOutputWidths[] = {
        1,
        1
      };
```

```
static int_T rt_LoggedOutputNumDimensions[] = {
  1,
  1
};

static int_T rt_LoggedOutputDimensions[] = {
  1,
  1
};

static boolean_T rt_LoggedOutputIsVarDims[] = {
  0,
  0
};

static void* rt_LoggedCurrentSignalDimensions[] = {
  (NULL),
  (NULL)
};

static int_T rt_LoggedCurrentSignalDimensionsSize[] = {
  4,
  4
};

static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
  SS_SINGLE,
  SS_SINGLE
};

static int_T rt_LoggedOutputComplexSignals[] = {
  0,
  0
};

static RTWPreprocessingFcnPtr rt_LoggingPreprocessingFcnPtrs[] = {
  (NULL),
  (NULL)
};

static const char_T *rt_LoggedOutputLabels[] = {
  "",
  "" };

static const char_T *rt_LoggedOutputBlockNames[] = {
  "ex_arg_code/Out1",
  "ex_arg_code/Out2" };

static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
  { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 },

  { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 }
};

static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
  {
    2,
```

```
        rt_LoggedOutputWidths,
        rt_LoggedOutputNumDimensions,
        rt_LoggedOutputDimensions,
        rt_LoggedOutputIsVarDims,
        rt_LoggedCurrentSignalDimensions,
        rt_LoggedCurrentSignalDimensionsSize,
        rt_LoggedOutputDataTypeIds,
        rt_LoggedOutputComplexSignals,
        (NULL),
        rt_LoggingPreprocessingFcnPtrs,

        { rt_LoggedOutputLabels },
        (NULL),
        (NULL),
        (NULL),

        { rt_LoggedOutputBlockNames },

        { (NULL) },
        (NULL),
        rt_RTWLogDataTypeConvert
      }
    };

    rtliSetLogYSignalInfo(ex_arg_code_M->rtwLogInfo, rt_LoggedOutputSignalInfo);

    /* set currSigDims field */
    rt_LoggedCurrentSignalDimensions[0] = &rt_LoggedOutputWidths[0];
    rt_LoggedCurrentSignalDimensions[1] = &rt_LoggedOutputWidths[1];
  }

  rtliSetLogY(ex_arg_code_M->rtwLogInfo, "yout");
}

/* states (dwork) */
(void) memset((void *)&ex_arg_code_DW, 0,
              sizeof(DW_ex_arg_code_T));

/* external inputs */
ex_arg_code_U.In1 = 0.0F;

/* external outputs */
(void)memset(&ex_arg_code_Y, 0, sizeof(ExtY_ex_arg_code_T));

/* Model Initialize function for ModelReference Block: '<Root>/Model' */
ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
  &(ex_arg_code_DW.Model_InstanceData.rtm),
  &(ex_arg_code_DW.Model_InstanceData.rtdw));

/* Model Initialize function for ModelReference Block: '<Root>/Model1' */
ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
  &(ex_arg_code_DW.Model1_InstanceData.rtm),
  &(ex_arg_code_DW.Model1_InstanceData.rtdw));

/* Matfile logging */
rt_StartDataLoggingWithStartTime(ex_arg_code_M->rtwLogInfo, 0.0, rtmGetTFinal
  (ex_arg_code_M), ex_arg_code_M->Timing.stepSize0, (&rtmGetErrorStatus
  (ex_arg_code_M)));
```

```
  /* SystemInitialize for ModelReference: '<Root>/Model' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* SystemInitialize for ModelReference: '<Root>/Model1' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model1_InstanceData.rtdw));
}

/* Model terminate function */
void ex_arg_code_terminate(void)
{
  /* (no terminate code required) */
}
```

The global variables in the generated code use the data type `real32_T` (`single`) because:

1. The parameter objects in the base workspace use context-sensitive data typing because the `DataType` property is set to `auto` (the default). With this setting, the parameter objects in the base workspace use the same data type as the model arguments, `single`.
2. The global variables in the generated code use the same data type as the parameter objects in the base workspace.

Group Multiple Model Arguments into Single Structure

Use the Model Explorer to copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace into the base workspace.

```
temp = getVariable(modelWorkspace,'gainArg');
gainArg = copy(temp);
temp = getVariable(modelWorkspace,'coeffArg');
coeffArg = copy(temp);
```

At the command prompt, combine these two parameter objects into a structure, `structArg`.

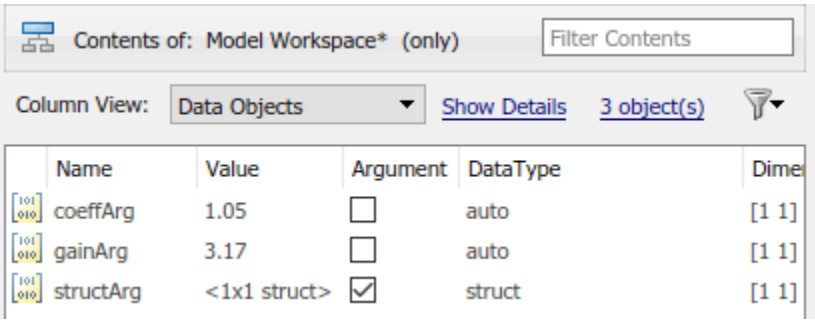
```
structArg = Simulink.Parameter(struct('gain',gainArg.Value,...
    'coeff',coeffArg.Value));
```

Use the Model Explorer to move `structArg` into the model workspace.

```
assignin(modelWorkspace,'structArg',copy(structArg));
clear structArg gainArg coeffArg
```

In the **Contents** pane, configure `structArg` as the only model argument.

```
set_param('ex_arg_code_ref','ParameterArgumentNames','structArg')
```



In the `ex_arg_code_ref` model, select the Model Data Editor **Parameters** tab.

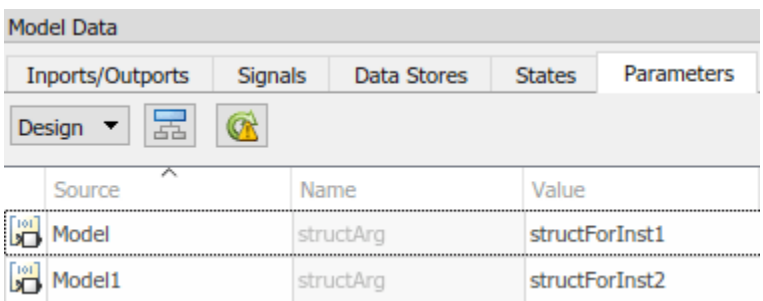
Use the Model Data Editor to set the value of the **Gain** parameter to `structArg.gain` and the value of the **Numerator** parameter to `structArg.coeff`. Save the model.

```
set_param('ex_arg_code_ref/Gain','Gain','structArg.gain')
set_param('ex_arg_code_ref/Discrete Filter',...
    'Numerator','structArg.coeff')
save_system('ex_arg_code_ref')
```

At the command prompt, combine the four parameter objects in the base workspace into two structures. Each structure stores the parameter values for one instance of `ex_arg_code_ref`.

```
structForInst1 = Simulink.Parameter(struct('gain',gainForInst1.Value,...
    'coeff',coeffForInst1.Value));

structForInst2 = Simulink.Parameter(struct('gain',gainForInst2.Value,...
    'coeff',coeffForInst2.Value));
```

In the top model ex_arg_code, use the Model Data Editor to set the argument values according to this figure.




```
instSpecParams = get_param('ex_arg_code/Model','InstanceParameters');
instSpecParams(1).Value = 'structForInst1';
instSpecParams1 = get_param('ex_arg_code/Model1','InstanceParameters');
instSpecParams1(1).Value = 'structForInst2';
set_param('ex_arg_code/Model','InstanceParameters',instSpecParams);
set_param('ex_arg_code/Model1','InstanceParameters',instSpecParams1);
```

Click the **Show/refresh additional information** button.

For the new parameter objects `structForInst1` and `structForInst2`, use the Model Explorer to apply the storage class `ExportedGlobal`.

```
structForInst1.StorageClass = 'ExportedGlobal';
structForInst2.StorageClass = 'ExportedGlobal';
```

At the command prompt, use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object. The hierarchy of elements in the object matches the hierarchy of the structure fields. The default name of the object is `slBus1`.

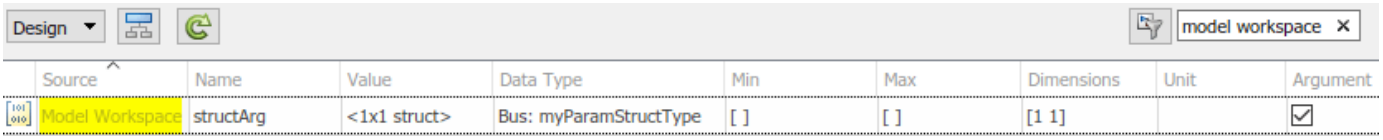
```
Simulink.Bus.createObject(structForInst1.Value);
```

Rename the bus object as `myParamStructType` by copying it.

```
myParamStructType = copy(slBus1);
```

In the Model Data Editor for `ex_arg_code`, use the Data Type column to set the data type of `structForInst1` and `structForInst2` to `Bus: myParamStructType`.

```
structForInst1.DataType = 'Bus: myParamStructType';
structForInst2.DataType = 'Bus: myParamStructType';
```




```
temp = getVariable(modelWorkspace,'structArg');
temp = copy(temp);
temp.DataType = 'Bus: myParamStructType';
assignin(modelWorkspace,'structArg',copy(temp));
```

Save the `ex_arg_code_ref` model.

```
save_system('ex_arg_code_ref')
```

When you use structures to group parameter values, you cannot take advantage of context-sensitive data typing to control the data types of the fields of the structures (for example, the fields of `structForInst1`). However, you can use the properties of the bus object to control the field data types.

At the command promt, set the data type of the elements in the bus object to `single`. The corresponding fields in the structures (such as `structForInst1` and `structArg`) use the same data type.

```
myParamStructType.Elements(1).DataType = 'single';
myParamStructType.Elements(2).DataType = 'single';
```

Generate code from the top model, `ex_arg_code`.

```
slbuild('ex_arg_code')
```

```
### Starting serial model reference code generation build.
### Successfully updated the model reference code generation target for: ex_arg_code_ref
### Starting build procedure for: ex_arg_code
### Successful completion of build procedure for: ex_arg_code

Build Summary

Code generation targets built:

Model            Action                        Rebuild Reason
==========================================================================================
ex_arg_code_ref  Code generated and compiled.  Model or library ex_arg_code_ref changed.

Top model targets built:

Model        Action                        Rebuild Reason
==============================================================================
ex_arg_code  Code generated and compiled.  Referenced models were updated.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 20.112s
```

The file `ex_arg_code_types.h` defines the structure type `myParamStructType`, which corresponds to the `Simulink.Bus` object.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code_types.h');
rtwdemodbtype(file,'typedef struct {','} myParamStructType;',1,1)
```

```
typedef struct {
  real32_T gain;
  real32_T coeff;
} myParamStructType;
```

In the file `ex_arg_code_ref.c`, the referenced model entry-point function has a formal parameter, `rtp_structArg`, that correspond to the model argument `structArg`.

```
file = fullfile('slprj','grt','ex_arg_code_ref','ex_arg_code_ref.c');
rtwdemodbtype(file,'/* Output and update for referenced model:',...
    '*rtp_structArg)',1,1)
```

```
/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1, const
                     myParamStructType *rtp_structArg, DW_ex_arg_code_ref_f_T
                     *localDW)
{
  real32_T rtb_Sum;

  /* Sum: '<Root>/Sum' incorporates:
   *  UnitDelay: '<Root>/Unit Delay'
   */
  rtb_Sum = *rtu_In1 + localDW->UnitDelay_DSTATE;

  /* Gain: '<Root>/Gain' */
  rtb_Sum *= rtp_structArg->gain;

  /* DiscreteFilter: '<Root>/Discrete Filter' */
  localDW->DiscreteFilter_tmp = rtb_Sum - 0.5F * localDW->DiscreteFilter_states;
  *rty_Out1 = rtp_structArg->coeff * localDW->DiscreteFilter_tmp;

  /* Update for UnitDelay: '<Root>/Unit Delay' */
  localDW->UnitDelay_DSTATE = rtb_Sum;

  /* Update for DiscreteFilter: '<Root>/Discrete Filter' */
  localDW->DiscreteFilter_states = localDW->DiscreteFilter_tmp;
}

/* Model initialize function */
void ex_arg_code_ref_initialize(const char_T **rt_errorStatus,
  RT_MODEL_ex_arg_code_ref_T *const ex_arg_code_ref_M, DW_ex_arg_code_ref_f_T
  *localDW)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatusPointer(ex_arg_code_ref_M, rt_errorStatus);

  /* states (dwork) */
  (void) memset((void *)localDW, 0,
```

```
                sizeof(DW_ex_arg_code_ref_f_T));
}
```

The file `ex_arg_code.c` defines the global structure variables that correspond to the parameter objects in the base workspace.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Block states (default storage) */',1,0)
```

```
/* Exported block parameters */
myParamStructType structForInst1 = {
  2.98F,
  0.98F
} ;                                   /* Variable: structForInst1
                                       * Referenced by: '<Root>/Model'
                                       */

myParamStructType structForInst2 = {
  3.34F,
  1.11F
} ;                                   /* Variable: structForInst2
                                       * Referenced by: '<Root>/Model1'
                                       */
```

The top model algorithm in the file `ex_arg_code.c` passes the addresses of the structure variables to the referenced model entry-point function.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file,'/* ModelReference: ''<Root>/Model'' incorporates:',...
    '&structForInst2);',1,1)
```

```
  /* ModelReference: '<Root>/Model' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out1'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1, &structForInst1,
                  &(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* ModelReference: '<Root>/Model1' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out2'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2, &structForInst2,
                  &(ex_arg_code_DW.Model1_InstanceData.rtdw));

  /* Matfile logging */
  rt_UpdateTXYLogVars(ex_arg_code_M->rtwLogInfo,
                      (&ex_arg_code_M->Timing.taskTime0));

  /* signal main to stop simulation */
  {                                   /* Sample time: [0.2s, 0.0s] */
    if ((rtmGetTFinal(ex_arg_code_M)!=-1) &&
        !((rtmGetTFinal(ex_arg_code_M)-ex_arg_code_M->Timing.taskTime0) >
          ex_arg_code_M->Timing.taskTime0 * (DBL_EPSILON))) {
      rtmSetErrorStatus(ex_arg_code_M, "Simulation finished");
```

```
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++ex_arg_code_M->Timing.clockTick0)) {
    ++ex_arg_code_M->Timing.clockTickH0;
  }

  ex_arg_code_M->Timing.taskTime0 = ex_arg_code_M->Timing.clockTick0 *
    ex_arg_code_M->Timing.stepSize0 + ex_arg_code_M->Timing.clockTickH0 *
    ex_arg_code_M->Timing.stepSize0 * 4294967296.0;
}

/* Model initialize function */
void ex_arg_code_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)ex_arg_code_M, 0,
                sizeof(RT_MODEL_ex_arg_code_T));
  rtmSetTFinal(ex_arg_code_M, 10.0);
  ex_arg_code_M->Timing.stepSize0 = 0.2;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    ex_arg_code_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(ex_arg_code_M->rtwLogInfo, (NULL));
    rtliSetLogT(ex_arg_code_M->rtwLogInfo, "tout");
    rtliSetLogX(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogXFinal(ex_arg_code_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(ex_arg_code_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogMaxRows(ex_arg_code_M->rtwLogInfo, 0);
    rtliSetLogDecimation(ex_arg_code_M->rtwLogInfo, 1);

    /*
     * Set pointers to the data and signal info for each output
     */
    {
```

```
  static void * rt_LoggedOutputSignalPtrs[] = {
    &ex_arg_code_Y.Out1,
    &ex_arg_code_Y.Out2
  };

  rtliSetLogYSignalPtrs(ex_arg_code_M->rtwLogInfo, ((LogSignalPtrsType)
    rt_LoggedOutputSignalPtrs));
}

{
  static int_T rt_LoggedOutputWidths[] = {
    1,
    1
  };

  static int_T rt_LoggedOutputNumDimensions[] = {
    1,
    1
  };

  static int_T rt_LoggedOutputDimensions[] = {
    1,
    1
  };

  static boolean_T rt_LoggedOutputIsVarDims[] = {
    0,
    0
  };

  static void* rt_LoggedCurrentSignalDimensions[] = {
    (NULL),
    (NULL)
  };

  static int_T rt_LoggedCurrentSignalDimensionsSize[] = {
    4,
    4
  };

  static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
    SS_SINGLE,
    SS_SINGLE
  };

  static int_T rt_LoggedOutputComplexSignals[] = {
    0,
    0
  };

  static RTWPreprocessingFcnPtr rt_LoggingPreprocessingFcnPtrs[] = {
    (NULL),
    (NULL)
  };

  static const char_T *rt_LoggedOutputLabels[] = {
    "",
    "" };
```

```
    static const char_T *rt_LoggedOutputBlockNames[] = {
      "ex_arg_code/Out1",
      "ex_arg_code/Out2" };

    static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
      { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 },

      { 0, SS_SINGLE, SS_SINGLE, 0, 0, 0, 1.0, 0, 0.0 }
    };

    static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
      {
        2,
        rt_LoggedOutputWidths,
        rt_LoggedOutputNumDimensions,
        rt_LoggedOutputDimensions,
        rt_LoggedOutputIsVarDims,
        rt_LoggedCurrentSignalDimensions,
        rt_LoggedCurrentSignalDimensionsSize,
        rt_LoggedOutputDataTypeIds,
        rt_LoggedOutputComplexSignals,
        (NULL),
        rt_LoggingPreprocessingFcnPtrs,

        { rt_LoggedOutputLabels },
        (NULL),
        (NULL),
        (NULL),

        { rt_LoggedOutputBlockNames },

        { (NULL) },
        (NULL),
        rt_RTWLogDataTypeConvert
      }
    };

    rtliSetLogYSignalInfo(ex_arg_code_M->rtwLogInfo, rt_LoggedOutputSignalInfo);

    /* set currSigDims field */
    rt_LoggedCurrentSignalDimensions[0] = &rt_LoggedOutputWidths[0];
    rt_LoggedCurrentSignalDimensions[1] = &rt_LoggedOutputWidths[1];
  }

  rtliSetLogY(ex_arg_code_M->rtwLogInfo, "yout");
}

/* states (dwork) */
(void) memset((void *)&ex_arg_code_DW, 0,
              sizeof(DW_ex_arg_code_T));

/* external inputs */
ex_arg_code_U.In1 = 0.0F;

/* external outputs */
(void)memset(&ex_arg_code_Y, 0, sizeof(ExtY_ex_arg_code_T));
```

```
  /* Model Initialize function for ModelReference Block: '<Root>/Model' */
  ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
    &(ex_arg_code_DW.Model_InstanceData.rtm),
    &(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* Model Initialize function for ModelReference Block: '<Root>/Model1' */
  ex_arg_code_ref_initialize(rtmGetErrorStatusPointer(ex_arg_code_M),
    &(ex_arg_code_DW.Model1_InstanceData.rtm),
    &(ex_arg_code_DW.Model1_InstanceData.rtdw));

  /* Matfile logging */
  rt_StartDataLoggingWithStartTime(ex_arg_code_M->rtwLogInfo, 0.0, rtmGetTFinal
    (ex_arg_code_M), ex_arg_code_M->Timing.stepSize0, (&rtmGetErrorStatus
    (ex_arg_code_M)));

  /* SystemInitialize for ModelReference: '<Root>/Model' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model_InstanceData.rtdw));

  /* SystemInitialize for ModelReference: '<Root>/Model1' */
  ex_arg_code_ref_Init(&(ex_arg_code_DW.Model1_InstanceData.rtdw));
}

/* Model terminate function */
void ex_arg_code_terminate(void)
{
  /* (no terminate code required) */
}
```

Control Data Types of Model Arguments and Argument Values

When you use model arguments, you can apply a data type to:

- The block parameters that use the arguments (for certain blocks, such as those in the Discrete library).
- The arguments in the referenced model workspace.
- The argument values that you specify in Model blocks.

To generate efficient code by eliminating unnecessary typecasts and C shifts, consider using inherited and context-sensitive data typing to match the data types.

- In the model workspace, use a MATLAB variable whose data type is `double` or a parameter object whose `DataType` property is set to `auto`. In this case, the variable or object uses the same data type as the block parameter.
- When you set the argument values in Model blocks, take advantage of context-sensitive data typing. To set an argument value, use an untyped value.

  - A literal number such as `15.23`. Do not use a typed expression such as `single(15.23)`.
  - A MATLAB variable whose data type is `double`.
  - A `Simulink.Parameter` object whose `DataType` property is set to `auto`.

  In these cases, the number, variable, or object uses the same data type as the model argument in the referenced model workspace. If you also configure the model argument to use context-sensitive data typing, you can control the data types of the block parameter, the argument, and the argument value by specifying the type only for the block parameter.

For basic information about controlling parameter data types, see “Parameter Data Types in the Generated Code” on page 36-131.

Use Model Argument in Different Data Type Contexts

If you use a model argument to set multiple block parameter values, and the data types of the block parameters differ, you cannot use context-sensitive data typing (`double` or `auto`) for the argument in the model workspace. You must explicitly specify a data type for the argument. For example, if the argument in the model workspace is a parameter object (such as `Simulink.Parameter`), set the `DataType` property to a value other than `auto`. For more information about this situation, see “Reuse Parameter Data in Different Data Type Contexts” on page 36-143.

In this case, you can continue to take advantage of context-sensitive data typing to control the data type of the argument values that you specify in Model blocks. Each argument value uses the data type that you specify for the corresponding argument in the model workspace.

See Also

Related Examples

- “Generate Code for Model Reference Hierarchy” on page 6-2
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Parameterize Instances of a Reusable Referenced Model”
- “Organize Data into Structures in Generated Code” on page 36-147
- “Parameter Data Types in the Generated Code” on page 36-131

Generated Functions in a Model Reference Hierarchy

When you generate code for a model, the code generator creates a subfolder `<model>_grt_rtw` (Simulink Coder) or a subfolder `<model>_ert_rtw` (Embedded Coder) in your current working folder and places source code files in this folder. The generated code is in these two primary files:

- `<model>.c` — This file contains the algorithm code. The generated code is mainly comprised of model data and model entry-point functions that are accessible to a caller by including the `<model>.h` file. You can call the generated model entry-point functions from external code or from a version of the generated `main` function that you modify.
- `<model>.h` — This file contains the `extern` declarations for block outputs, continuous states, model output, entry points, and timing data. For more information about data structures, see "Data Structures in the Generated Code" on page 36-31.

When you generate code for a referenced model hierarchy, the code generator places the code for the top model in the subfolder `<model>_grt_rtw` (Simulink Coder) or the subfolder `<model>_ert_rtw` (Embedded Coder) in your current working folder and places the code for referenced models in an `slprj` folder.

Generated Functions in a Top Model

In a model reference hierarchy, the code generator commonly produces these functions for a top model.

Function Name	Description
`<model>_step`	Execution function that contains the output and update code for the blocks in a Simulink model. This function calls the referenced model function. The code generator produces this function by default. To generate separate functions for output and update code, clear the model configuration parameter **Single output/update function**. For more information, see `model_step`.
`<model>_initialize`	Initialization function that executes the initialization code for a Simulink model. The code generator calls this function once at the start of the application code. This function might call the `<ref_model>_Init` function of a referenced model to initialize states and `<ref_model>_initialize` for zero initialization. For more information, see `model_initialize`.
`<model>_terminate`	Termination function that contains the code to turn off a system. For ERT-based models, you can suppress generation of this function by clearing the model configuration parameter **Terminate function required**. For more information, see `model_terminate`.

Generated Functions in a Referenced Model

The functions generated for a referenced model are internal functions used by the code generator. These functions are not intended to interface with external code. In a model reference hierarchy, the code generator commonly produces these functions for a referenced model.

Function Name	Description
`<ref_model>_Init`	Initialization function in a referenced model that is called from a top model entry-point function `<model>_initialize`. The `<model>_initialize` of the top model initializes the code for the model reference hierarchy and calls the `<ref_model>_Init` of the child model to initialize the states to their initial condition.
`<ref_model>`	Execution function that serves a similar purpose as `<model>_step` of a top model but is not an entry-point function for the external code. The two functions differ in terms of the scope. The `<ref_model>` function computes the outputs from internal state and updates that state. The `<model>_step` does the same but calls the individual `<ref_model>` function of the referenced models.
`<ref_model>_initialize`	Initialization function that executes the initialization code for a referenced model. The code generator calls this function from a top model entry-point function `<model>_initialize`. The `<model>_initialize` of the top model initializes the code for the model reference hierarchy and calls the `<ref_model>_initialize` of the child model to zero-initialize the referenced model code.

See Also

Related Examples

- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

Combined Models in Simulink Coder

- “Combine Code Generated for Multiple Models” on page 7-2
- “Function Reuse in Generated Code” on page 7-5
- “File Packaging for Models (Code and Data)” on page 7-10

Combine Code Generated for Multiple Models

To combine code, which the code generator produces for multiple models or multiple instances of a model, into one executable program, use these techniques:

- Model reference hierarchy. See “Model Reference Basics” and “Generate Code for Model Reference Hierarchy”.
- If you have Embedded Coder software, interface the code for multiple models to a common harness program. From the harness program, call the entry-point functions generated for each model. The `ert.tlc` system target file has restrictions, related to embedded processing that might be incompatible with your application.
- Generate reusable, multi-instance code that is reentrant. See “Combine Code Generated for Multiple Models or Multiple Instances of a Model” on page 7-3.

Note The S-function system target (`rtwsfcn.tlc`) does not support combining code generated for multiple models.

To combine models for simulation and code generation, consider using model referencing. Model referencing helps with:

- Symbol naming consistency
- Required scheduling of the overall algorithm
- Model configuration consistency

Considerations for Combining Code for Different Models

If you combine code generated for different models that is, without using referenced models, consider the following:

- Data is global. Symbol (name) clashes can result.
- Configuration parameter settings for the models must match, including settings such as hardware word sizes.
- Reuse and sharing of code can be suboptimal (for example, duplicate code for shared utility functions, scheduling, and solvers).
- Scheduling can be more complex (for example, models can have periodic sample times that are not multiples of each other, making scheduling from a common timer interrupt more complicated)
- For plant models that use continuous time and state, the continuous time signals connecting models are not handled by a single solver like continuous time signals within a model, which can lead to subtle, numeric differences.

Share Data Across Models

Use unidirectional signal connections between models. These connections affect the order in which models are called. For example, if you use an output signal from `modelA` as input to `modelB`, the `modelA` output computation is called first.

Timing Issues

When combining code generated for multiple models or multiple instances of a model:

- Configure the models by using the same solver mode (single-tasking or multitasking).
- If the models use continuous states, configure the models by using the same solver.

If the base rates for the models differ, the main program (such as `rt_malloc_main.c`) must set up the timer interrupt to occur at the greatest common divisor rate of the models. The main program calls each model at a time interval.

Data Logging and External Mode Support

A multiple-model program can log data to separate MAT-files for each model.

Only one of the models in a multiple-model program can use external mode.

Control Ownership of Data

If you have Embedded Coder software, you can specify an owner for individual data items such as signals, parameters, and states. The owner of a data item generates the definition, the memory allocation and initialization, for the data item. For example, if you apply a storage class to a `Simulink.Signal` object so that it appears as a global variable in the generated code, specify one of the combined models as the owner of the object. The code generated for that model defines the variable.

If you use model referencing, you can modularize the generated code and establish clear ownership of data when you work in a team.

If you do not use model referencing, you can prevent generation of duplicate definitions for a data item. For example, suppose you store a `Simulink.Parameter` object in the base workspace and apply the storage class `ExportedGlobal`. If you generate code from two separate models that use the object, each model generates a definition for the corresponding global variable. Instead, you can specify an owner for the object so that only the owner generates a definition.

To specify an owner for a data item:

1. Apply a storage class to the data item. See “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.
2. Configure the owner of the data item by specifying a value for the **Owner** property.
3. Select the model configuration parameter **Use owner from data object for data definition placement**.

For more information about controlling ownership and file placement of data definitions and declarations, see “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.

Combine Code Generated for Multiple Models or Multiple Instances of a Model

For each model that you are combining code for, generate the code.

1. Set the system target file to a GRT- or ERT-based system target file. The system target file for the models that you combine must be the same.
2. If you intend to have multiple instances of that model in the application, set the model configuration parameter **Code interface packaging** to `Reusable function`. If you specified

an ERT-based system target file, optionally, you can set the model configuration parameter **Use dynamic memory allocation for model initialization**, depending on whether you want to statically or dynamically allocate the memory for each instance of the model.

3. Generate source code. The code generator includes an allocation function in the generated file `model.c`. The allocation function dynamically allocates model data for each instance of the model.

After generating source code for each model:

1. Compile the code for each model that you are combining.
2. Combine the makefiles generated for the models into one makefile.
3. Create a combined simulation engine by modifying a main program, such as `rt_malloc_main.c`. The main program initializes and calls the code generated for each model.
4. Run the makefile. The makefile links the object files and the main program to an executable program.

See Also

Related Examples

- “Generate Code for Model Reference Hierarchy” on page 6-2
- “Generate Reusable Code from Model Reference Hierarchy” on page 6-16

Function Reuse in Generated Code

This example shows how to configure an atomic subsystem for generating reusable code. To specify that the code generated for a subsystem execute as an atomic unit, on the Block Parameters dialog box, select the **Treat as atomic unit** parameter. That parameter enables the **Function Packaging** parameter on the **Code Generation** tab. The **Function Packaging** parameter has these four settings:

- `Inline`: Inline the subsystem code
- `Nonreusable function`: Function with I/O passed as global data
- `Reusable function`: Function with I/O passed as function arguments
- `Auto`: Let Simulink Coder optimize based on context

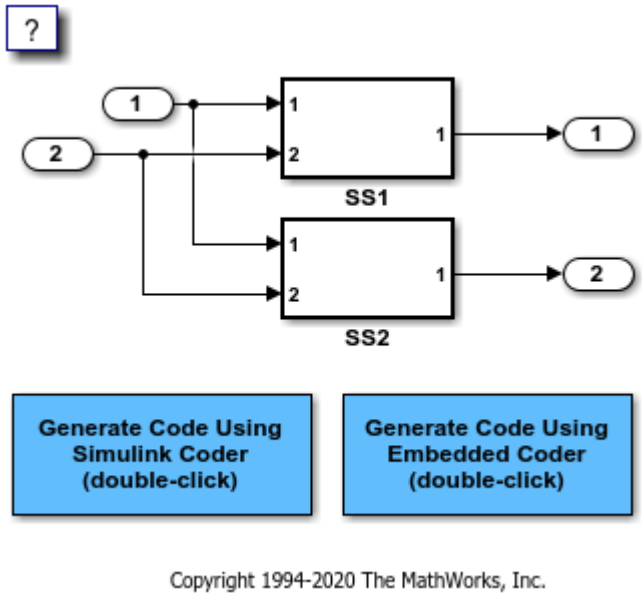
The `Reusable function` and `Auto` settings permit the code generator to reuse subsystem code. The `Reusable function` and `Nonreusable function` settings enable the **Function name options**, **Function name**, and **File name options** parameters.

If you have an Embedded Coder® license, you can configure a nonreusable subsystem to accept arguments.

Example Model

The `rtwdemo_ssreuse` model contains two identical subsystems, `SS1` and `SS2`. For these subsystems, the **Function packaging** parameter is set to `Reusable function`, and the **Function name** parameter is `myfun`. The subsystems are parameterized masked subsystems. To see the contents of the masked subsystems, right-click the subsystem blocks and select **Mask > Look Under Mask**.

```
model = 'rtwdemo_ssreuse';
open_system(model);
```



Generate and Inspect Code

Create a temporary folder for the build and inspection process. Open the **Simulink Coder** or **Embedded Coder** app. Then, generate and inspect the code.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================
rtwdemo_ssreuse  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.727s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_ssreuse_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   */
  myfun(rtwdemo_ssreuse_U.In1, rtwdemo_ssreuse_U.In2, rtwdemo_ssreuse_P.T1Data,
        rtwdemo_ssreuse_P.T1Break, &rtwdemo_ssreuse_B.SS1);

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outport: '<Root>/Out1' */
  rtwdemo_ssreuse_Y.Out1 = rtwdemo_ssreuse_B.SS1.LookupTable;

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   */
  myfun(rtwdemo_ssreuse_U.In1, rtwdemo_ssreuse_U.In2, rtwdemo_ssreuse_P.T2Data,
        rtwdemo_ssreuse_P.T2Break, &rtwdemo_ssreuse_B.SS2);

  /* End of Outputs for SubSystem: '<Root>/SS2' */

  /* Outport: '<Root>/Out2' */
  rtwdemo_ssreuse_Y.Out2 = rtwdemo_ssreuse_B.SS2.LookupTable;
}
```

In the model step function, there are two calls to the reusable function, `myfun`. The mask parameters, `T1Break`, `T1Data`, `T2Break`, and `T2Data`, are function arguments.

Change the **Function Packaging** parameter to `Inline`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Inline')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Inline')
```

Generate and inspect the code.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================
rtwdemo_ssreuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.379s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_ssreuse_step(void)
{
  real_T Out1_tmp;

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */
  /* Sum: '<S1>/Sum' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<S2>/Sum'
   */
  Out1_tmp = rtwdemo_ssreuse_U.In1 + rtwdemo_ssreuse_U.In2;

  /* End of Outputs for SubSystem: '<Root>/SS2' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Lookup_n-D: '<S1>/Lookup Table'
   *  Sum: '<S1>/Sum'
   */
  rtwdemo_ssreuse_Y.Out1 = look1_binlx(Out1_tmp, rtwdemo_ssreuse_P.T1Break,
    rtwdemo_ssreuse_P.T1Data, 10U);

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  /* Outport: '<Root>/Out2' incorporates:
   *  Lookup_n-D: '<S2>/Lookup Table'
   */
  rtwdemo_ssreuse_Y.Out2 = look1_binlx(Out1_tmp, rtwdemo_ssreuse_P.T2Break,
    rtwdemo_ssreuse_P.T2Data, 10U);
```

```
  /* End of Outputs for SubSystem: '<Root>/SS2' */
}
```

In the model step function, the subsystem code is inlined.

Change the **Function Packaging** parameter to `Nonreusable function`. For `SS2` , change the **Function name** parameter to `myfun2`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Nonreusable function')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Nonreusable function')
set_param('rtwdemo_ssreuse/SS2','RTWFcnName','myfun2')
```

Generate and inspect the code.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
================================================================================
rtwdemo_ssreuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.175s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_ssreuse_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */
  myfun();

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  myfun2();

  /* End of Outputs for SubSystem: '<Root>/SS2' */
}
```

The model step function contains calls to the functions `myfun` and `myfun2`. These functions have a void-void interface.

Change the **Function Packaging** parameter to `Auto`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Auto')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Auto')
```

For the `auto` setting, Simulink Coder chooses the optimal format. For this model, the optimal format is a reusable function.

Close the model and clean up.

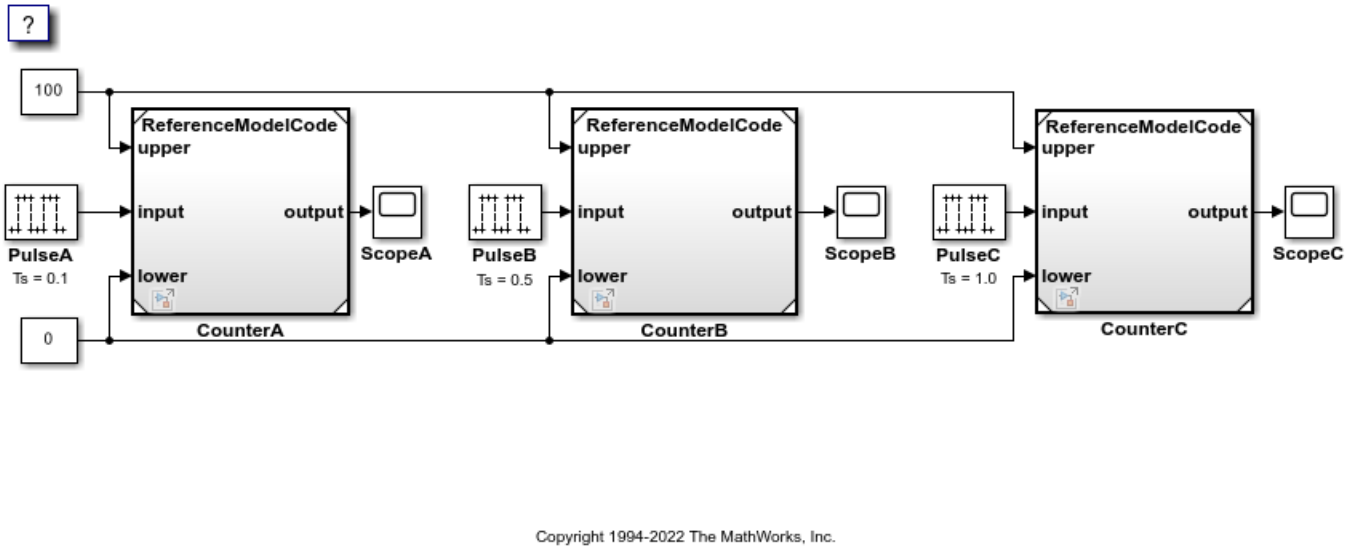
```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

File Packaging for Models (Code and Data)

This example shows how referenced models provide system interface encapsulation and incremental code generation. You can reference one model from another model (one or more times), and aspects of the referenced model are fixed: input/output signal types, parameter types, and sample times. Therefore, you can modularize your design and perform incremental code generation.

The code generator partitions data and functions of a referenced model into its own set of files, independent of its parent model. In this example, the referenced model `ReferenceModelCode` is referenced three times. For simulation and code generation, the model is incrementally generated, which means `ReferenceModelCode` builds the first time, but not on subsequent builds, unless you change `ReferenceModelCode`.

```
open_system('TopModelCode')
```



1. Generate and inspect the code.
2. Change the value of one of the constant inputs in `TopModelCode`.
3. Generate code. The code generator does not reproduce code for reference model `TopModelCode`.

```
bdclose('TopModelCode');
```

Code Reuse for Simulink Coder

What Is Code Reuse?

Code reuse is a programming technique that reduces time and resources to develop software. When you develop code for reuse, the code serves multiple purposes. The technique involves modularization, which enables multiple individuals to develop code for various system components independently and in parallel. The technique also simplifies software distribution.

A software library is an example of code reuse. A library can contain code for several functions, each having specific behavior. To use a library, you need to know only the interface, which is the specification for calling the library functions.

Reusable code can be reentrant. You can invoke, interrupt, and reinvoke reentrant code. Reentrant code resides in shared memory. Instances of data associated with each use of the code are unique and preserved. Multiple calls to a reusable, reentrant function, for example, can access the function code with each call maintaining a unique data set.

Applications of code reuse include:

- Team-based development.
- Switching between implementations of a function.
- Intellectual property protection.
- Overriding library behavior with a custom implementation.
- Implementing a library that is based on standard library functions.
- Unit testing.
- Execution speed improvement.

The code generator supports code reuse and reentrancy. You can use key Simulink componentization techniques to partition a model into design components that you simulate, generate code for, and verify independently. You can save individual components as subsystems, libraries, referenced models, or combinations of these elements from which you can generate code. For example, the code generator produces reusable function code from library subsystems and export-function models.

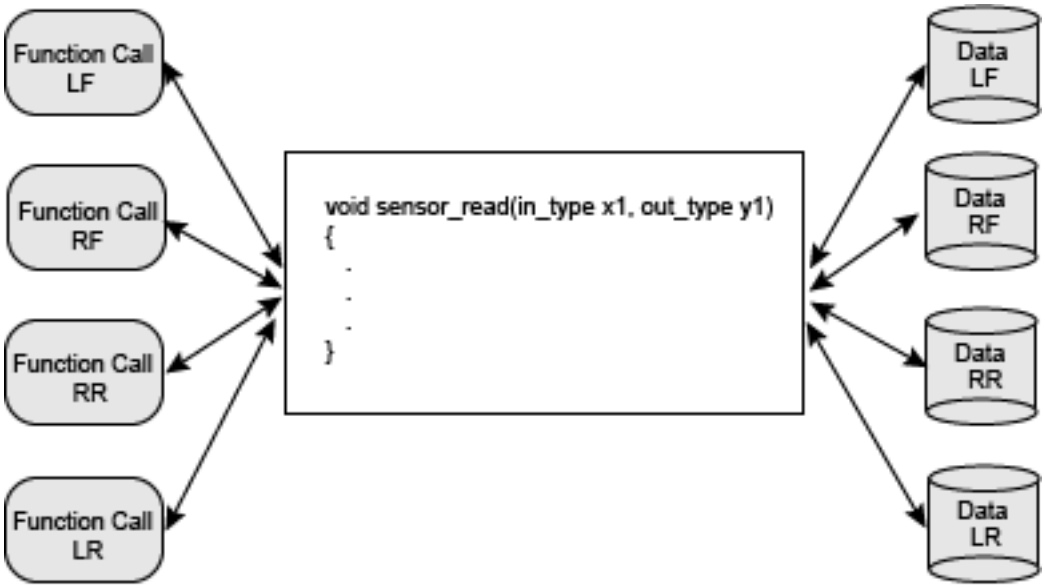
See Also

More About

- “What Is Reentrant Code?” on page 8-3
- “Choose a Componentization Technique for Code Reuse” on page 8-4

What Is Reentrant Code?

Reentrant (multi-instance) code is a reusable routine that multiple programs can invoke, interrupt, and reinvoke simultaneously. When you want to reuse code, but associate each instance of the shared code with unique, preserved data, use reentrant code. For example, consider this figure that shows four instances of function `sensor_read`. Although the function algorithm is the same in each case, the data associated with each instance varies and is relative to the sensor position.



To make generated code for a model component reentrant, you configure the component such that model entry-point functions receive root-level input and output data as arguments. How you configure the component depends on the modeling style or technique that you apply.

See Also

More About

- “What Is Code Reuse?” on page 8-2
- “Choose a Componentization Technique for Code Reuse” on page 8-4

Choose a Componentization Technique for Code Reuse

Key componentization techniques that you can use with Simulink and the code generator to produce reusable code include:

- Referenced models
- Subsystems
- Library subsystems
- Combinations of models, subsystems, and library subsystems

Choose componentization techniques based on your code reuse goals.

Goal	Referenced Model	Subsystem in Model	Subsystem in Library
Design for explicit code reuse.	√	√	√
Facilitate parallel team development.	√	√	√
Reuse function within a model or across models.	√		
Reduce build time by generating reusable code incrementally.	√		
Verify reusable code with SIL or PIL simulation.	√		
Optimize generated code by configuring code generator to detect opportunities for code reuse.	√	√	√
Maximize reuse with context-dependent behavior.		√	√
Develop a frequently used, and infrequently changed, utility function.			√

See Also

More About

- “Compare Capabilities of Model Components”
- “Generate Reentrant Code from Top Models” on page 8-13

- “Generate Reentrant Code from Subsystems” on page 8-28
- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36
- “Library-Based Code Generation for Reusable Library Subsystems” on page 9-2

Simulink Function Blocks and Code Generation

Why Generate Code from Simulink Function Blocks and Function Callers?

Simulink Function blocks provide a mechanism for generating C or C++ code for modeling components that represent shared resources. You define the logic as a resource in a Simulink Function block, which separates the function interface (name and arguments) from the implementation of the logic. Function callers (Function Caller blocks, MATLAB Function blocks, and Stateflow charts) can then reuse the function logic at different levels of the model hierarchy.

Simulink Function blocks provide an alternative to reusable subsystems. For example, a consideration for using a Simulink Function block instead of a subsystem block is that a Simulink Function block shares states between function callers. The code generator produces one function. If the Simulink Function block contains blocks that have states, such as a delay or memory, the states persist between function callers. The order of the function calls is an important consideration.

Reusable functions that the code generator produces from subsystems do not share states. The code generator produces one function for multiple instances of the subsystem as an optimization. If that subsystem contains blocks that have states, the code generator produces one function, but passes a different state variable to each instance. The instances do not share states.

Other uses of Simulink Function blocks and callers include:

- Nesting calls to a function.
- Calling a function defined in one modeling component from another modeling component.
- Generating functions that are globally accessible or scoped.
- Producing code for a client and server application.

Implementation Options

Choose how to implement Simulink functions and function callers based on your code generation requirements. Considerations include:

- How you represent a function and function callers in a model
- Scope of a function
- Whether to export a function from a model (requires Embedded Coder)
- Function code interface customizations (requires Embedded Coder)
- Code generation requirements
- Code generation limitations

Choose a Modeling Pattern

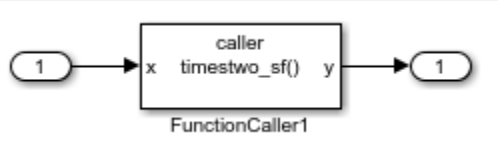
This table shows C code for a function that multiplies an input value times two and a Simulink Function block that can represent that function in a Simulink model.

Function Definition	Modeling Element

codegen-folder/*subsystem*.c
 	
`#include "timestwo_sf.h"`
 	
`#include "ex_slfunc_comp_sf.h"`
`#include "ex_slfunc_comp_sf_private.h"`
 	
`void timestwo_sf(real_T rtu_x, real_T *rty_y)`
`{`
`  *rty_y = 2.0 * rtu_x;`
`}`
	Simulink Function block

A Simulink function caller invokes a function defined with a Simulink Function block. From anywhere in a model or chart hierarchy, you can call a function defined with a Simulink Function block by using one of these modeling elements:

Function Call	Modeling Element

codegen-folder/*model*.c
 	
`void ex_slfunc_comp_sf_step(void)`
`{`
`  real_T rtb_FunctionCaller1;`
 	
`timestwo_sf(ex_slfunc_comp_sf_U.In1, &rtb_FunctionC`
 	
`ex_slfunc_comp_sf_Y.Out1 = rtb_FunctionCaller1;`
`.`
`.`
`.`
	Function Caller block
	
	 

codegen-folder/*model*.c
 	
`void ex_slfunc_comp_gf_step(void)`
`{`
`  real_T rtb_y1_l;`
 	
`  timestwo_gf(ex_slfunc_comp_gf_U.In4, &rtb_y1_l);`
 	
`  ex_slfunc_comp_gf_Y.Out4 = rtb_y1_l;`
`  .`
`  .`
`  .`
	Stateflow chart transition

codegen-folder/*model*.c
 	
`void ex_slfunc_comp_mf_step(void)`
`{`
`  real_T rtb_y;`
 	
`  timestwo_mf(ex_slfunc_comp_mf_U.In3, &rtb_y);`
 	
`  ex_slfunc_comp_mf_Y.Out3 = rtb_y;`
`  .`
`  .`
`  .`
	MATLAB Function block

For more information about modeling choices, see “Simulink Functions Overview”.

Specify Function Scope

A function that you define with a Simulink Function block can be global or scoped.

- Global---The code generator places code for a global function in source and header files that are separate from model code files (for example, *function*.c and *function*.h) . The separate files make the function code available for sharing between function callers.
- Scoped---The code generator places code for a scoped function in model code files (*model*.c and *model*.h). To call a scoped function in the context of a model, the function caller must be at the same level as the function in the model hierarchy, or one or more levels below.

  To create a library of functions that are accessible from anywhere in the generated model code, set up each function as a scoped Simulink Function block. Place each scoped function within a virtual subsystem at the root level of a model.

For more information, see and “Scoped, Global, and Port-Scoped Simulink Function Blocks Overview”.

Decide Whether to Generate Export Function Code

Although you can use Simulink Function blocks in a single top-model design, function code is more reusable when you generate it as standalone, atomic components. If you are using Embedded Coder, you can design functions in the context of export-function models.

For information, see “Generate Component Source Code for Export to External Code Base” on page 58-40 and “Export-Function Models Overview”.

Configure Function Code Interfaces

With Embedded Coder, simplify integration of generated code with external code by configuring generated function code interfaces for Simulink Function and Function Caller blocks. You can configure the function interfaces for:

- Global Simulink Function blocks
- Scoped Simulink Function blocks that are at the root level of a model

For more information, see “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2.

Uncalled Simulink Function Blocks

If you use a Simulink Function block in a rate-based model and do not call that function, the code generator treats the Simulink Function block as a constant and does not produce function code. For example, this can occur during model development when you are ready to define a function, but are not ready to identify a caller.

To identify such blocks in a rate-based model, display sample time colors during simulation (see “View Sample Time Information”). By default, Constant blocks appear magenta. Because the code generator considers uncalled Simulink Function blocks constants during simulation, they appear magenta.

Requirements

- Within a model hierarchy, function names are unique. If the code generator finds multiple functions with the same name, it issues an error. Change the name of one of the functions and delete the `slprj` folder.
- The signature (for example, the arguments and argument data types) for a function and function callers must match.
  - If the code generator finds the function first and the signature of a function caller does not match, the code generator issues an error. Change the function caller signature to match the signature of the Simulink Function block or delete the `slprj` folder.
  - If the code generator finds a function caller first and the signature of the function does not match, the code generator issues a warning message. Change the signature of the function or function caller so that the signatures match.
- In a Simulink Function block definition, do not define input and output signals for Argument Inport and Argument Outport blocks with a storage class.
- Do not specify Argument Inport and Argument Outport blocks as test points.
- If you specify the data type of input and output signals for Argument Inport and Argument Outport blocks as a `Simulink.IntEnumType`, `Simulink.AliasType`, or `Simulink.Bus`, set the `DataScope` property to `Imported` or `Exported`.
- A function interface and function callers must agree in data type, complexity, dimension, and number of arguments.

Limitations

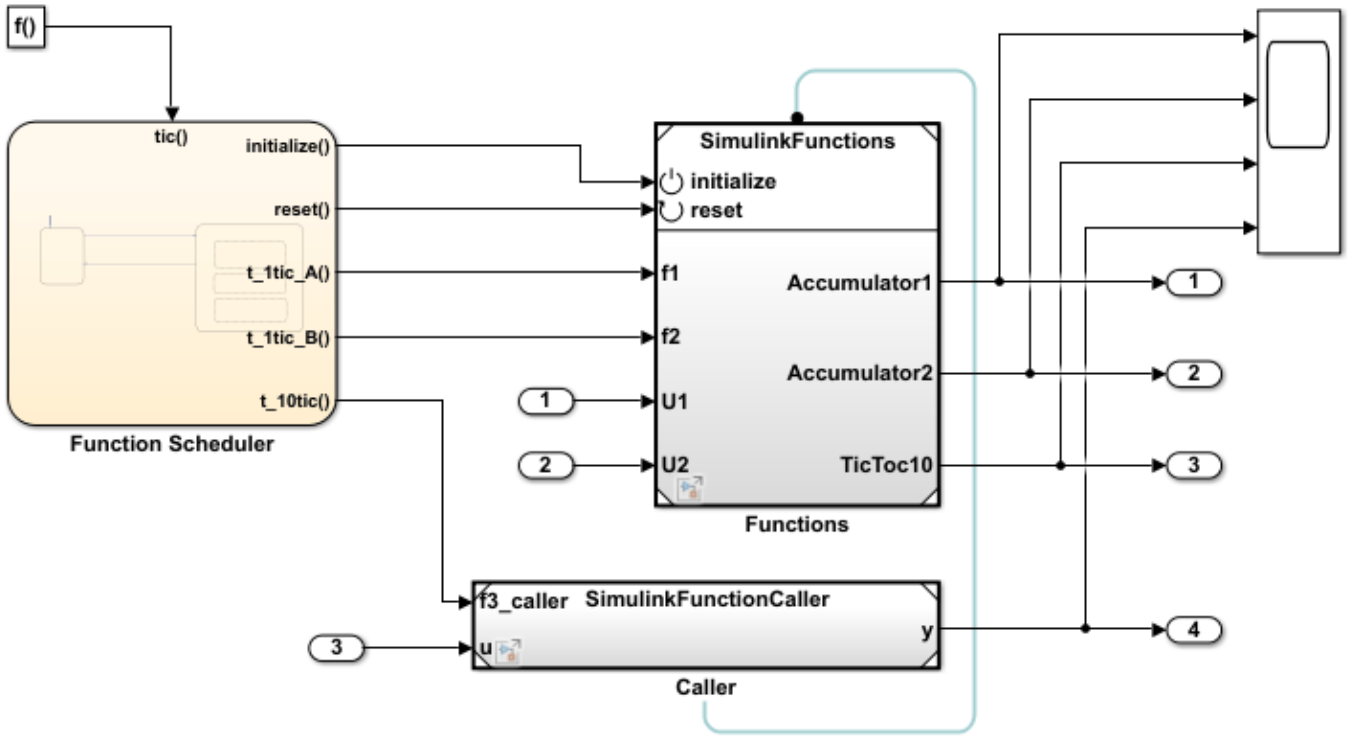
- Simulink functions and function callers do not honor the `MaxStackSize` parameter.
- To generate code from a model that includes scoped Simulink functions, the model must be an export-function model. The code generator does not support rate-based models that include scoped Simulink functions.
- You can use a Simulink Function block to define a scoped function in a referenced model. You cannot generate code for an export-function model that uses a Function Caller block in an atomic subsystem to invoke that function.
- You can invoke a C++ function that the code generator produces from a Simulink Function block by using code generated from a Stateflow chart. Due to current scope limitations for generated C ++ functions, you must invoke those functions with code generated from a Function Caller block.
- Code generation for a C++ class interface supports scoped Simulink functions only.

Generate Code for Simulink Function and Function Caller

This example shows how to generate C code for Simulink Function and Function Caller blocks and displays the relevant generated code.

Open the example model `SimulinkFunctionsTestHarness`. The model uses Stateflow software, but this example reviews only the code generated from the referenced models.

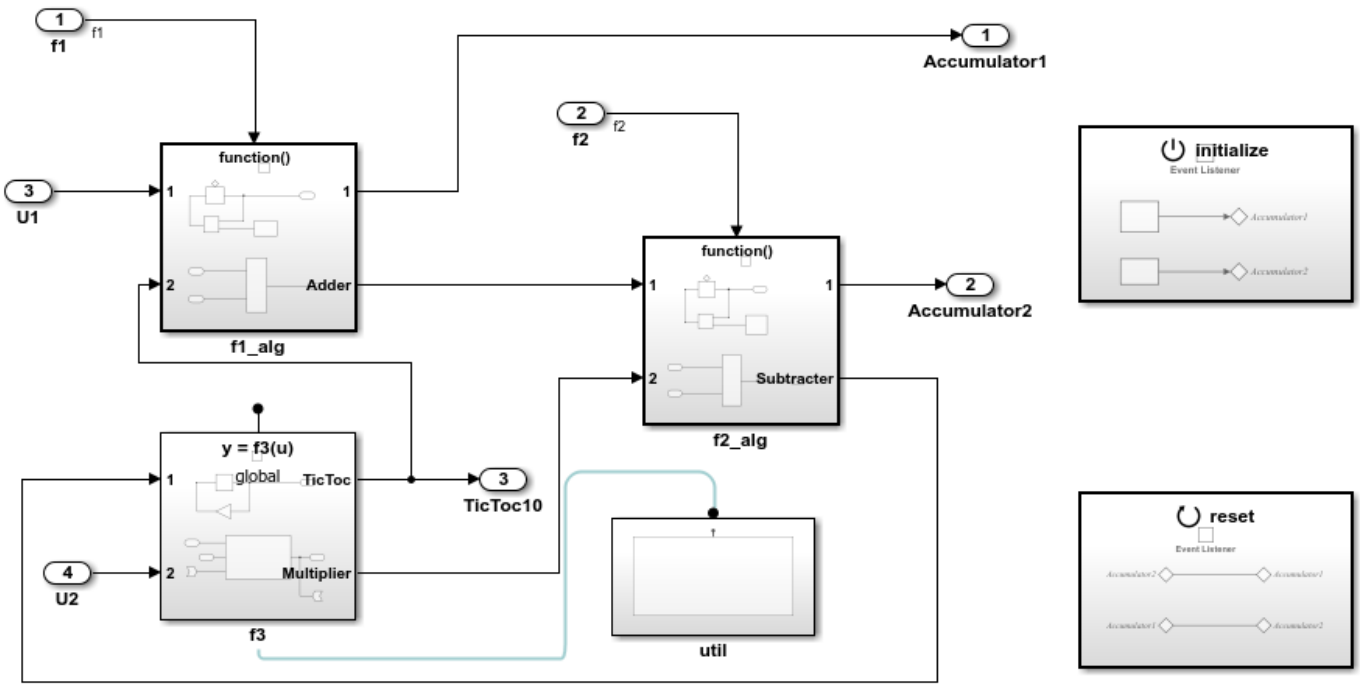
```
open_system('SimulinkFunctionsTestHarness')
```



Generate Code for Function Definition

Open model `SimulinkFunctions`. The Simulink Function block is the `f3` subsystem defined as `y = f3(u)`.

```
open_system('SimulinkFunctions')
```



Generate code. The code generator creates `SimulinkFunctions.c`. This file contains the function initialization code and function definition.

Initialization code for function `f3`:

```
void SimulinkFunctions_initialize(void)
{
  rtDWork.Delay_DSTATE = 1;
}
```

Code for function `f3`:

```
void f3(real_T rtu_u, real_T *rty_y)
{
  int8_T = rtb_Gain;

  rtY.TicToc10 = rtDWork.Delay_DSTATE;

  rtb_Gain = (int8_T)(int32_T)-(int32_T)rtY.TicToc10;

  adder(rtB.Subtract, rtU.U2, rtu_u, &rtB.FunctionCaller);

  *rty_y = rtB.FunctionCaller;

  rtDWork.Delay_DSTATE = rtb_Gain;
}

static void adder(real_T rtu_u1, real_T rtu_u2, real_T rtu_u3, real_T *rty_y)
{
  *rty_y = (rtu_u1 + rtu_u2) + rtu_u3;
}
```

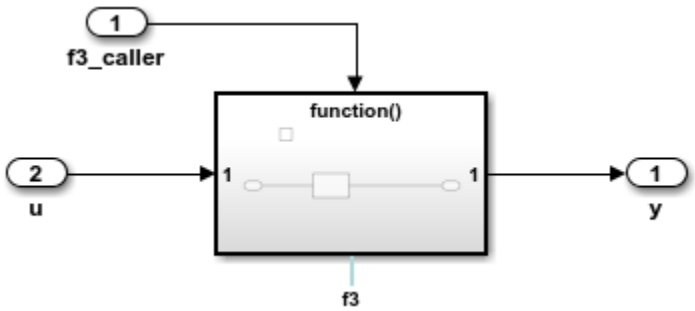
The shared header file `f3.h` contains the entry-point declaration for function `f3`.

```
#include "rtwtypes.h"

extern void f3(real_T rtu_u, real_T *rty_y);
```

Generate Code For Function Caller

`Open model SimulinFunctionCaller` to view the contents of the caller subsystem.

```
open_system('SimulinkFunctionCaller')
```



Generate code. The code generator creates the files `SimulinkFunctionCaller.h` and `SimulinkFunctionCaller.c` in the folder `SimulinkFunctionCaller_ert_rtw`.

`SimulinkFunctionCaller.h` includes the shared header file, `f3.h`, which contains the function entry-point declaration.

`SimulinkFunctionCaller.c` calls function `f3`.

```
void f3_caller(void)
{
  rtY.y = f3(rtu_u);
}
```

More About

- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Simulink Functions Overview”
- “Scoped Simulink Function Blocks in Models”
- “Simulink Function Blocks in Referenced Models”

See Also

Simulink Function | Function Caller

Related Examples

- “Simulink Functions Overview”
- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2
- “Client-Server Communication Interfaces” on page 3-65

Generate Reentrant Code from Top Models

By default, for top models, the code generator produces code that is not reentrant. Entry-point functions have a void-void interface. Code communicates with other code by sharing access to global data structures that reside in shared memory.

For applications that can benefit from reuse and require that each use or instance of the code maintains its own unique data, configure a model such that the code generator produces reentrant code. To generate reentrant code, set the model configuration parameter Code interface packaging to `Reusable function`. If you are using Embedded Coder and generating C++ code, alternatively, you can set the parameter to `C++ class`. In both cases, the code generator:

- Packages model data, such as block I/O, DWork vectors, and parameters, in the real-time model data structure (`rtModel`) .
- Passes the real-time model data structure as an input argument, by reference, to generated model entry-point functions.
- Passes root-level input and output arguments to generated model entry-point functions as individual arguments.
- Allocates memory for model data structures statically.
- Exports the real-time model data structure in the generated header file *`model`*`.h`.

Apply additional diagnostic and code generation control by setting these model configuration parameters:

- To select the severity level for diagnostic messages that the code generator displays when a model does not meet requirements for multi-instance code, set parameter Multi-instance code error diagnostic to `None`, `Warning`, or `Error`. Set the parameter to `Error` unless you need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
- To control how the generated code passes root-level model input and output to the reusable execution (step) function (requires Embedded Coder), set parameter Pass root-level I/O as to `Individual arguments`, `Structure reference`, or `Part of model data structure`.

  When you set Code interface packaging to `Reusable function`, the code generator packages model data (such as block I/O, Dwork, and parameters) into the real-time model data structure, and passes the model structure to generated model entry-point functions. If you set Pass root-level I/O as to `Part of model data structure`, the code generator packages root-level model input and output into the real-time model data structure also.
- To reduce memory usage by omitting the error status field from the real-time model data structure (requires Embedded Coder), select model configuration parameter **Remove error status field in real-time model data structure**.
- To include a function in the generated file *`model`*`.c` that uses `malloc` to dynamically allocate memory for model instance data (requires Embedded Coder), select model configuration parameter **Use dynamic memory allocation for model initialization**. If you do not select this parameter, the generated code statically allocates memory for model data structures.

Generate Reentrant, Multi-Instance Code

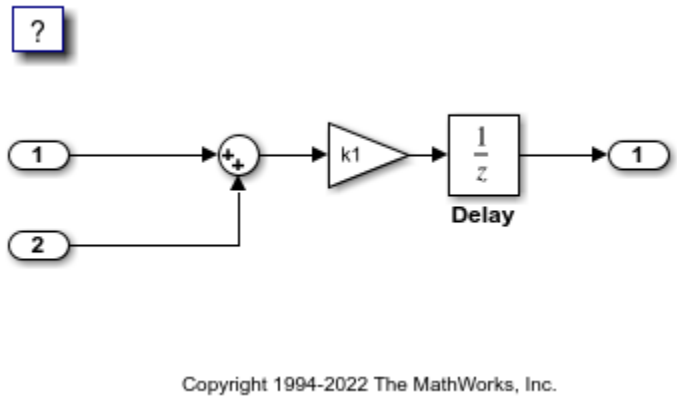
This example shows you how to configure a model for reentrant, multi-instance code generation. Multiple programs can use reentrant code simultaneously. When you configure a model for

reentrancy, the execution (step) entry-point function uses root-level input and output arguments instead of global data structures. After examining the configuration settings, generate and review the generated code.

Open the Model

Open the model `Reusable`. The model contains two root Inport blocks and a root Outport block.

```
model='Reusable';
open_system(model);
```



Examine Relevant Model Configuration Parameter Settings

1. Open the **Embedded Coder** app.

2. Open the Model Configuration Parameters dialog box.

3. Model configuration parameter **System target file** is set to `ert.tlc`. Although you can generate reentrant code for a model configured with **System target file** set to `grt.tlc`, ERT and ERT-based system target files provide more control over how the code passes root-level I/O.

4. Open the **Code Generation > Interface** pane and explore relevant model configuration parameter settings.

- **Code interface packaging** is set to `Reusable function`. This parameter setting instructs the code generator to produce reusable, multi-instance code.
- Setting of `Reusable function` displays parameter **Multi-instance code error diagnostic**. That parameter is set to `Error`, indicating that the code generator abort if the model violates requirements for generating multi-instance code.
- **Pass root-level I/O as** is set to `Part of model data structure`. This setting packages root-level model input and output into the real-time model data structure (`rtModel`), which is an optimized data structure that replaces `SimStruct` as the top-level data structure for a model.
- **Remove error status field in real-time model data structure** is selected. This parameter setting reduces memory usage by omitting the error status field from the generated real-time model data structure.

Generate and Review Code

```
slbuild(model);
```

```
### Starting build procedure for: Reusable
### Successful completion of build procedure for: Reusable

Build Summary

Top model targets built:

Model     Action                        Rebuild Reason
========================================================================================
Reusable  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.936s
```

Review the Generated Code

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point function `Reusable_step`. Use this file as a starting point for coding your execution framework.
- `Reusable.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `Reusable.h` declare model data structures and a public interface to the model entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework.

1. Include the generated header file by adding directive `#include Reusable.h`.

2. Write input data to the generated code for model Inport blocks.

3. Call the generated entry-point functions.

4. Read data from the generated code for the model Outport block.

Input ports:

- `<Root>/In1` of data `real_T` with dimension of 1
- `<Root>/In2` of data `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void Reusable_initialize(RT_MODEL *const rtM)`. At startup, call this function once.
- Output and update (step) entry-point function, `void Reusable_step(RT_MODEL *const rtM)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.

Output port:

- `<Root>/Out1` of data type `real_T` with dimension of 1

Examine the Step Function

Examine the |Reusable_step| function code in |Reusable.c|.

```
cfile = fullfile('Reusable_ert_rtw','Reusable.c');
rtwdemodbtype(cfile,'/* Model step function', '/* Model initialize function ', 1, 0);
```

```
/* Model step function */
void Reusable_step(RT_MODEL *const rtM)
{
  D_Work *rtDWork = rtM->dwork;
  ExternalInputs *rtU = (ExternalInputs *) rtM->inputs;
  ExternalOutputs *rtY = (ExternalOutputs *) rtM->outputs;

  /* Outport: '<Root>/Out1' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  rtY->Out1 = rtDWork->Delay_DSTATE;

  /* Gain: '<Root>/Gain' incorporates:
   *  Sum: '<Root>/Sum'
   *  UnitDelay: '<Root>/Delay'
   */
  rtDWork->Delay_DSTATE = (rtU->In1 + rtU->In2) * rtP.k1;
}
```

The code generator passes model data to the `Reusable_step` function as part of the real-time model data structure. Try different settings for model configuration parameters **Code interface packaging** and **Pass root-level I/O** and regenerate code. Observe how the function prototype changes.

Close the Model and Report

Close the model and the code generation report.

```
bdclose(model)
```

Share Data Between Instances

When your code calls a reentrant model entry-point function multiple times, each call represents an instance of the model. By default, the code generator generates code that assumes each instance reads from and writes to a separate copy of the signals, block states, and parameters in the model.

- To share a piece of parameter data between the instances (for example, to share a setpoint for a reusable PID control algorithm), use a parameter object, such as `Simulink.Parameter`. Then, configure the parameter with a storage class other than `Auto` or in the Code Mappings editor, set the default storage class for the corresponding category of parameter data `Default` (the default setting) to `Model default`. The parameter object appears in the code as a global symbol, such as a global variable, that the function accesses directly. For more information, see “C Code Generation Configuration for Model Interface Elements”.
- To share a piece of nonparameter data between the instances (for example, to share a fault indication or an accumulator), use a data store. You can configure the data store to appear in the code as a global symbol, such as a global variable, that the function accesses directly. Create a global data store by using a `Simulink.Signal` object or use a Data Store Memory block and select block parameter **Share across model instances**. For more information, see “Model Global Data by Creating Data Stores” and Data Store Memory.

See Also

More About

- “What Is Reentrant Code?” on page 8-3
- “Storage Classes and Reentrant, Multi-Instance Models and Components”

Generate Reentrant Code from Simulink Function Blocks

If you are using Embedded Coder, you can generate reusable, reentrant code by representing an algorithm as a scoped Simulink Function block. Examples of when to generate reentrant code from Simulink Function blocks are when a function shares state between function callers within a model or for client/server applications. You can generate code that is highly modularized by using multiple instances of a shared Simulink Function block in an export-function model. The code generator produces function code and associates each use or call to the function with instance-specific data. The scope of the function depends on whether you place the function at the root level of model or in a subsystem.

The code generator produces reentrant function code when you configure a:

- Top model with model configuration parameter Code interface packaging set to `Reusable function` or `C++ class` (C++ only).
- Referenced model with Total number of instances allowed per top model set to `Multiple`.

Call a function represented by a scoped Simulink Function block from one level above, at the same level, or from a level below the level of the function definition. You can scope a function in an atomic or nonvirtual subsystem, but function call accessibility is limited to the same level or below of the hierarchy. The function name does not have to be unique.

Identify Requirements

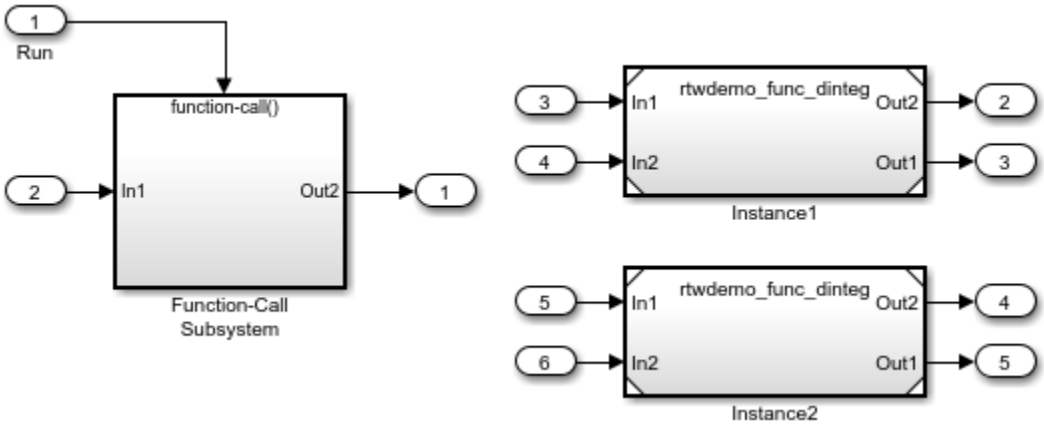
Before and while you design your model, consider:

- How many instances of each function are required?
- Do you need to restrict call sites for a function to the model containing the function definition?
- Do you need for a function to interface with signals in the local environment, but keep those signals hidden from callers?
- Do functions need to communicate directly with each other?
- Do functions need to connect to external I/O?
- Do you need to log function output?

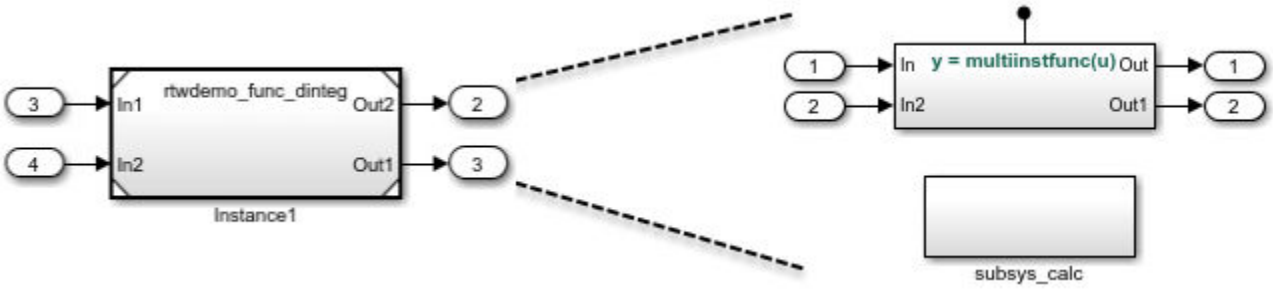
Create Model

Use the example model component `rtwdemo_comp` to see how to use Simulink Function blocks to generate reentrant C code. Use example model `rtwdemo_comp_cpp` if you prefer to generate C++ code. Open the model and examine the model hierarchy.

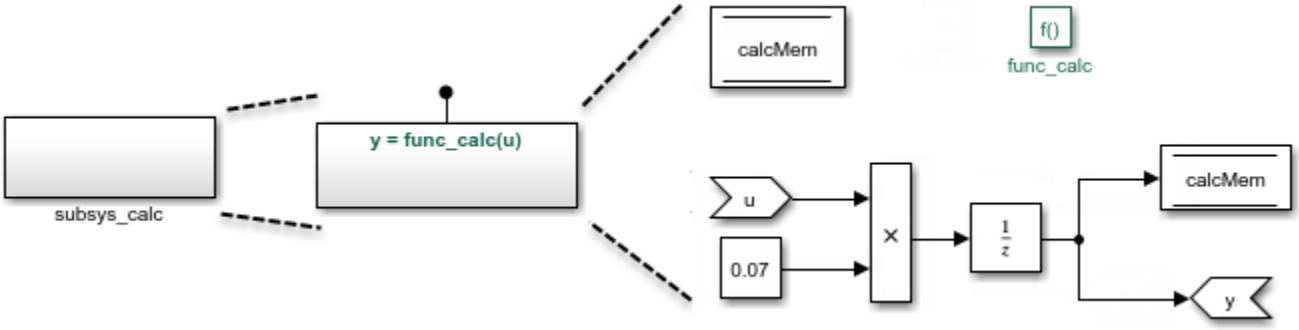
The top level of the model includes a function-call subsystem and two instances of a referenced model.



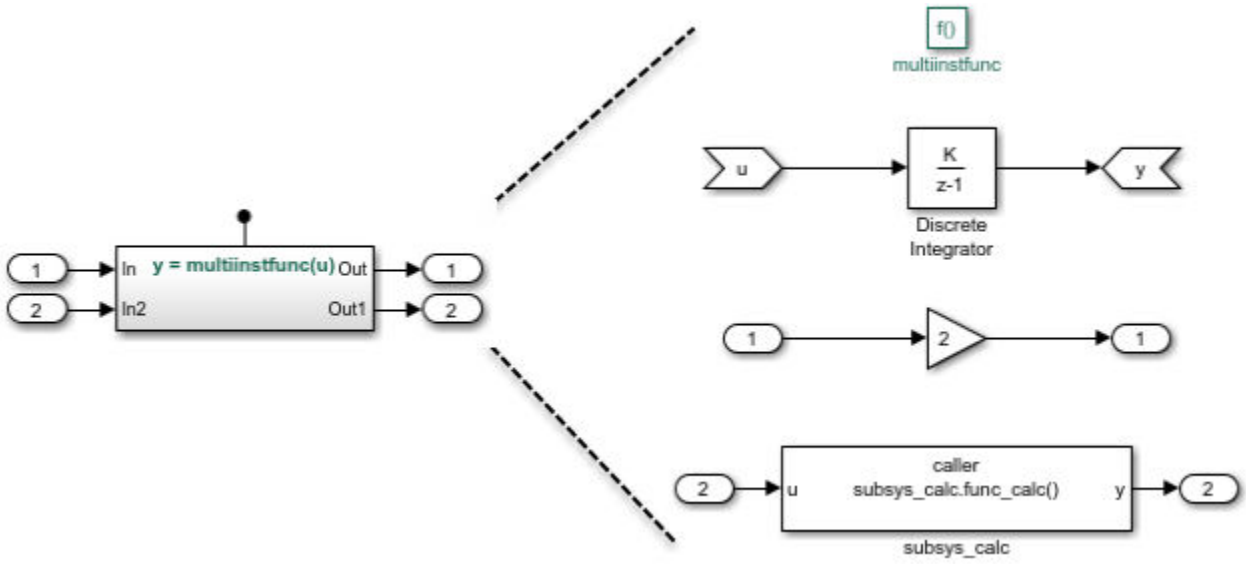
The referenced model, `rtwdemo_func_dinteg`, consists of a Simulink Function block that defines function `multiinstfunc` and subsystem `subsys_calc`.



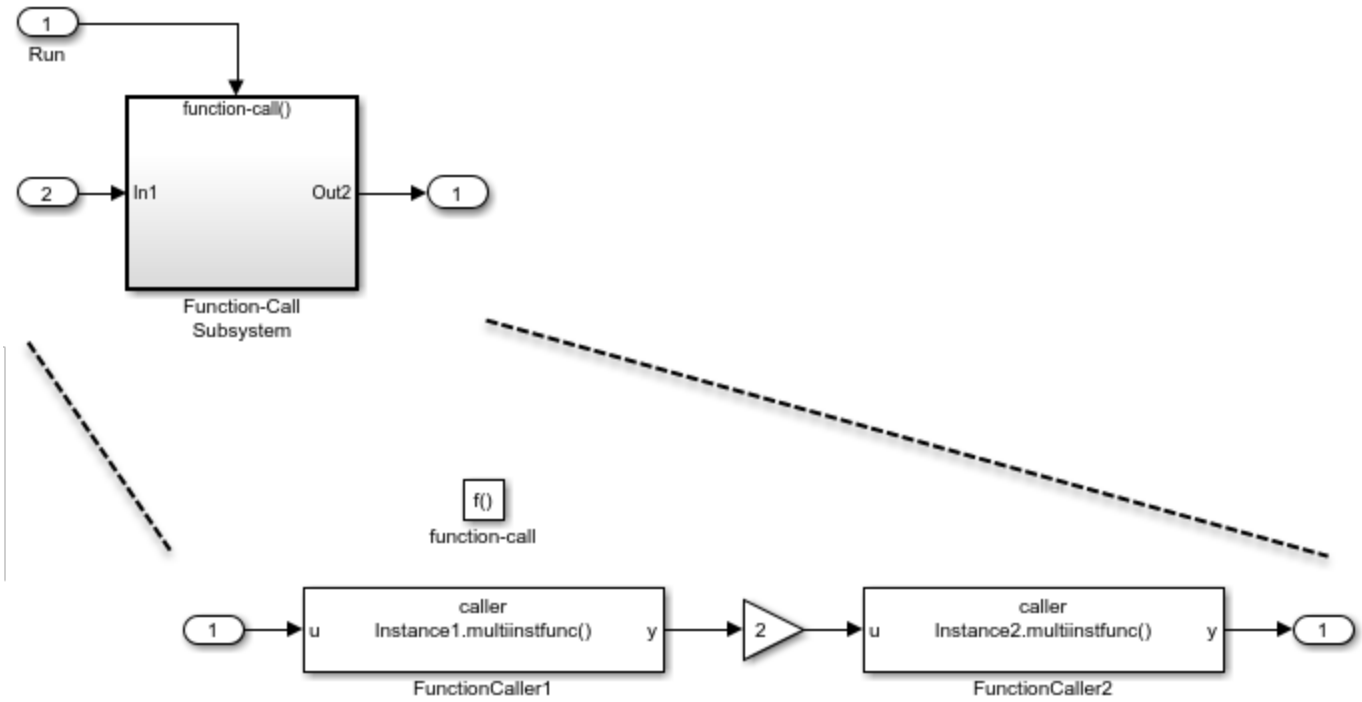
The subsystem consists of a Simulink Function block. This use of a Simulink Function block shows how you can limit the scope of the function that the block defines to the model that contains the subsystem. The code generator produces function code for `func_calc` and associates each call to the function with instance-specific data. The data includes states, such as data stored in memory.



The Simulink Function block that defines function `multiinstfunc` uses a Function Caller block to invoke function `func_calc`. That Simulink Function block also shows that it can interface to signals in the local environment of the block through Inport and Outport blocks.



At the top level of model `rtwdemo_comp`, the function-call subsystem uses Function Caller blocks to invoke the two instances of function `multiinstfunc`. The code generator produces function code and associates each call with instance-specific data.



Configure Model and Model Elements

Configure Simulink Function Blocks

Configure the Simulink Function blocks by setting parameters for the function Trigger Port block. For the code generator to produce reentrant code from Simulink Function blocks:

- Configure block instances with the same function name.
- Set block parameter **Function visibility** to `scoped`.

In the example, the function name for the Simulink Function blocks in the two instances of the referenced model `rtwdemo_func_dinteg` is specified as `multiinstfunc`.

Configure Function Callers

Configure blocks to call the Simulink Function block. You can call a Simulink Function block from a Function Caller block, Stateflow Chart block, or MATLAB Function block. For this example, use Function Caller blocks. For each of the blocks, set a value for the block parameter **Function prototype**. Start typing a prototype. For example, type `y`. Prototype options based on function definitions in the model appear in a selection list. Select the prototype that corresponds to each function call.

For this example, the prototypes are configured as follows:

- In the function-call subsystem, the prototypes for the function callers are configured as `y = Instance1.multiinstfunc(u)` and `y = Instance2.multiinstfunc(u)`. The `Instance`*n* prefix identifies each function invocation uniquely and associates the invocation with its own data set.
- The function caller in function `multiinstfunc` is configured with the prototype `y = subsys_calc.func_calc(u)`. The prefix `subsys_calc` identifies the subsystem that contains the function definition.

For the example, the input and output argument specifications and sample time retain default settings.

Subsystem Configuration

Configuration changes are not required for the subsystem in the example model. When you include a Simulink Function block in a subsystem, the code generator

- Scopes the function to the model that includes the subsystem.
- Treats the subsystem as an atomic unit.

Configure Referenced Model

Configure the referenced model that includes a Simulink Function block:

- In the Block Parameters dialog box, set parameter **Model name** to the referenced model file name. For this example, the model name is `rtwdemo_func_dinteg.slx`.
- Set model configuration parameter **Total number of instances allowed per top model** to `Multiple`.
- To generate a C++ class interface for the reference model, set model configuration parameter **Language** to `C++` and parameter **Code interface packaging** to `C++ class`.

Optionally, you can customize model entry-point function interfaces. You can specify entry-point function names. For the execution (step) entry-point function you can configure the function name and arguments. Custom entry-point function interfaces can minimize changes to existing external code that you integrate with the generated code. The example uses the default function interfaces. See “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2.

Configure Top Model

Configure the top model for a model component. If you want the model component (that is, the top model) to be reusable, set model configuration parameter **Code interface packaging** to `Reusable function`. If you are generating C++ code, you can set this parameter to `C++ class`. In either case, also:

- Set model configuration parameter **Multi-instance code error diagnostic** to `Error`.
- Set model configuration parameter **Pass root-level I/O as** to `Part of model data structure`.

Optionally, you can customize model entry-point function interfaces. You can specify entry-point function names. For the execution (step) entry-point function you can configure the function name and arguments. Custom entry-point function interfaces can minimize changes to existing external code that you integrate with the generated code. The example uses the default function interfaces. See “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2.

Generate and Inspect C Code

Generate C code for the model.

- Function code for multi-instance Simulink Function block

  When you place a scoped Simulink Function block in a referenced model that you use multiple times in another model, the code generator places the function code in the *`model`*`.c` file for the referenced model. For this example, the code generator places function code for `multiinstfunc` in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg.c`.

```
real_T rtwdemo_func_dinteg_multiinstfunc(RT_MODEL_rtwdemo_func_dinteg_T * const
  rtwdemo_func_dinteg_M, const real_T rtu_u)
{
  real_T rtb_TmpLatchAtIn0utport1;
  real_T rtb_TmpLatchAtIn2Outport1;
  real_T rty_y_0;

  rtb_TmpLatchAtIn0utport1 =
    *rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextInport.rtu_In1;

  rtb_TmpLatchAtIn2Outport1 =
    *rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextInport.rtu_In2;

  *rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtext0utport.rty_Out2 = 2.0 *
    rtb_TmpLatchAtIn0utport1;

  rtwdemo_func_dinteg_func_calc(rtwdemo_func_dinteg_M, rtb_TmpLatchAtIn2Outport1,
    rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtext0utport.rty_Out1);

  rty_y_0 = rtwdemo_func_dinteg_M->dwork.DiscreteIntegrator_DSTATE;

  rtwdemo_func_dinteg_M->dwork.DiscreteIntegrator_DSTATE += 0.1 * rtu_u;
  return rty_y_0;
}
```

- Function code for a Simulink Function block defined in a subsystem

  The code generator places the function code for a Simulink Function block that you define in a subsystem in the *`model`*`.c` file for the model that contains the subsystem. For this example, the code generator places the function code for `func_calc` in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg.c`.

```
void rtwdemo_func_dinteg_func_calc(RT_MODEL_rtwdemo_func_dinteg_T * const
  rtwdemo_func_dinteg_M, real_T rtu_u, real_T *rty_y)
{
  rtwdemo_func_dinteg_M->dwork.calcMem =
    rtwdemo_func_dinteg_M->dwork.UnitDelay_DSTATE;

  *rty_y = rtwdemo_func_dinteg_M->dwork.UnitDelay_DSTATE;

  rtwdemo_func_dinteg_M->dwork.UnitDelay_DSTATE = rtu_u * 0.07;
}
```

- Structure that stores multi-instance data for reusable functions

  The code generator uses a structure similar to the real-time model (`RT_MODEL`) data structure to store the multi-instance data associated with a reusable function. The code generator defines the structure in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg.h`.

```
typedef struct rtwdemo_func_dinteg_tag_RTM RT_MODEL_rtwdemo_func_dinteg_T;
```

- Initialization code for multi-instance referenced model

  For each instance of a referenced model that includes the same scoped Simulink Function block, the code generator produces initialization and startup function code. A single copy of the initialization code is defined in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg.c`.

```
void rtwdemo_func_dinteg_Start(RT_MODEL_rtwdemo_func_dinteg_T *const
  rtwdemo_func_dinteg_M, const real_T *rtu_In1, const real_T *rtu_In2, real_T
  *rty_Out2, real_T *rty_Out1)
{
  rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextInport.rtu_In1 = rtu_In1;
  rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextInport.rtu_In2 = rtu_In2;
  rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextOutport.rty_Out2 = rty_Out2;
  rtwdemo_func_dinteg_M->rtwdemo_func_dintegrtextOutport.rty_Out1 = rty_Out1;
}

void rtwdemo_func_dinteg_initialize(const char_T **rt_errorStatus,
  RT_MODEL_rtwdemo_func_dinteg_T *const rtwdemo_func_dinteg_M)
{
  {
    rtmSetErrorStatusPointer(rtwdemo_func_dinteg_M, rt_errorStatus);
  }
}
```

  The initialization code is called for each instance of a referenced model that contains the same Simulink Function block. That code is in file `rtwdemo_comp.c` in the build folder.

```
.
.
.
rtwdemo_func_dinteg_initialize(rtmGetErrorStatusPointer(rtwdemo_comp_M),
  (&(rtwdemo_comp_M->Instance1)));

rtwdemo_func_dinteg_initialize(rtmGetErrorStatusPointer(rtwdemo_comp_M),
  (&(rtwdemo_comp_M->Instance2)));

rtwdemo_func_dinteg_Start((&(rtwdemo_comp_M->Instance1)), &rtU->In2, &rtU->In3,
  &rtY->Out2, &rtY->Out3);

rtwdemo_func_dinteg_Start((&(rtwdemo_comp_M->Instance2)), &rtU->In4, &rtU->In5,
  &rtY->Out4, &rtY->Out5);
}
```

- Top model entry-point function declarations

  The model header file `rtwdemo_comp.h` includes `extern` declarations for top model initialize, terminate, and execution (run) entry-point functions.

```
extern void rtwdemo_comp_initialize(RT_MODEL_rtwdemo_comp_T *const
  rtwdemo_comp_M);
```

```
extern void Run(RT_MODEL_rtwdemo_comp_T *const rtwdemo_comp_M);
```

- Reference model entry-point function declarations

  Header file `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg.h` includes `extern` declarations for reference model entry-point functions.

```
extern void rtwdemo_func_dinteg_initialize(const char_T **rt_errorStatus,
  RT_MODEL_rtwdemo_func_dinteg_T *const rtwdemo_func_dinteg_M);
extern real_T rtwdemo_func_dinteg_multiinstfunc(RT_MODEL_rtwdemo_func_dinteg_T *
  const rtwdemo_func_dinteg_M, const real_T rtu_u);
extern void rtwdemo_func_dinteg_Start(RT_MODEL_rtwdemo_func_dinteg_T *const
  rtwdemo_func_dinteg_M, const real_T *rtu_In1, const real_T *rtu_In2, real_T
  *rty_Out2, real_T *rty_Out1);
```

Generate and Inspect C++ Code

Generate C++ code for the model.

- Function code for multi-instance Simulink Function block

  When you place a scoped Simulink Function block in a referenced model that you use multiple times in another model, the code generator places the function code in the *`model`*`.cpp` file for the referenced model. For this example, the code generator places function code for `multiinstfunc` in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg_cpp.cpp`.

```
real_T rtwdemo_func_dintegModelClass::multiinstfunc(const real_T rtu_u)
{
  real_T rtb_TmpLatchAtInOutport1;
  real_T rtb_TmpLatchAtIn2Outport1;
  real_T rty_y_0;

  rtb_TmpLatchAtInOutport1 = *rtwdemo_func_dinteg_cprtrtu_In1;

  rtb_TmpLatchAtIn2Outport1 = *rtwdemo_func_dinteg_cprtrtu_In2;

  *rtwdemo_func_dinteg_crtrty_Out2 = 2.0 * rtb_TmpLatchAtInOutport1;

  rtwdemo_func_dinteg_c_func_calc(rtb_TmpLatchAtIn2Outport1,
    rtwdemo_func_dinteg_crtrty_Out1);

  rty_y_0 = rtwdemo_func_dinteg_cpprtDW.DiscreteIntegrator_DSTATE;

  rtwdemo_func_dinteg_cpprtDW.DiscreteIntegrator_DSTATE += 0.1 * rtu_u;
  return rty_y_0;
}
```

- Function code for a Simulink Function block defined in a subsystem

  The code generator places the function code for a Simulink Function block that you define in a subsystem in *`model`*`.cpp` for the model that contains the subsystem. For this example, the code generator places the function code for `func_calc` in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg_cpp.cpp`.

```
void rtwdemo_func_dintegModelClass::rtwdemo_func_dinteg_c_func_calc(real_T rtu_u,
  real_T *rty_y)
{
  rtwdemo_func_dinteg_cpprtDW.calcMem =
    rtwdemo_func_dinteg_cpprtDW.UnitDelay_DSTATE;

  *rty_y = rtwdemo_func_dinteg_cpprtDW.UnitDelay_DSTATE;

  rtwdemo_func_dinteg_cpprtDW.UnitDelay_DSTATE = rtu_u * 0.07;
}
```

- Structure that stores multi-instance data for reusable functions

  The code generator uses a structure similar to the real-time model (`RT_MODEL`) data structure to store the multi-instance data associated with a `Reusable function`. The code generator defines the structure in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg_cpp.h`.

  ```
  typedef struct rtwdemo_func_dinteg_cpp_tag_RTM rtwdemo_func_dinteg_cp_RT_MODEL;
  ```

- Initialization code for multi-instance referenced model

  For each instance of a referenced model that includes the same scoped Simulink Function block, the code generator produces initialization and startup function code. A single copy of the initialization code is defined in `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg_cpp.cpp`.

  ```
  void rtwdemo_func_dintegModelClass::start(real_T *rtu_In1, real_T *rtu_In2,
    real_T *rty_Out2, real_T *rty_Out1)
  {
    rtwdemo_func_dinteg_cprtrtu_In1 = rtu_In1;
    rtwdemo_func_dinteg_cprtrtu_In2 = rtu_In2;
    rtwdemo_func_dinteg_crtrty_Out2 = rty_Out2;
    rtwdemo_func_dinteg_crtrty_Out1 = rty_Out1;
  }
  .
  .
  .
  void rtwdemo_func_dintegModelClass::initializeRTM()
  {
    (void) memset((void *)((&rtwdemo_func_dinteg_cpprtM)), 0,
                  sizeof(rtwdemo_func_dinteg_cp_RT_MODEL));
  }
  ```

  The initialization code is called for each instance of a referenced model that contains the same Simulink Function block. That code is in file `rtwdemo_comp_cpp.cpp` in the build folder.

  ```
  .
  .
  .

  Instance1MDLOBJ0.initializeRTM();

  Instance1MDLOBJ0.setErrorStatusPointer(rtmGetErrorStatusPointer((&rtM)));
  Instance1MDLOBJ0.initialize();

  Instance2MDLOBJ1.initializeRTM();

  Instance2MDLOBJ1.setErrorStatusPointer(rtmGetErrorStatusPointer((&rtM)));

  Instance1MDLOBJ0.start(&rtU.In2, &rtU.In3, &rtY.Out2, &rtY.Out3);

  Instance2MDLOBJ1.start(&rtwU.In4, &rtwU.In5, &rtY.Out4, &rtY.Out5);
  }
  ```

- Top model class declaration

  The model header file `rtwdemo_comp_cpp.h` includes the class declaration for the top model.

  ```
  class rtwdemo_compModelClass {
   public:
    ExtU rtU;

    ExtY rtY;

    void initialize();

    void Run();

    rtwdemo_compModelClass();
  ```

```
  ~rtwdemo_compModelClass();

  RT_MODEL * getRTM();

 private:
  RT_MODEL rtM;

  rtwdemo_func_dintegModelClass Instance1MDLOBJ0;

  rtwdemo_func_dintegModelClass Instance2MDLOBJ1;
};
```

- Reference model class declaration

  Header file `slprj/ert/rtwdemo_func_dinteg/rtwdemo_func_dinteg_cpp.h` includes the class declaration for the reference model.

```
class rtwdemo_func_dintegModelClass {
 public:
  real_T multiinstfunc(const real_T rtu_u);

  void start(real_T *rtu_In1, real_T *rtu_In2, real_T *rty_Out2, real_T
              *rty_Out1);

  rtwdemo_func_dintegModelClass();

  ~rtwdemo_func_dintegModelClass();

  rtwdemo_func_dinteg_cp_RT_MODEL * getRTM();

  void initializeRTM();

  void setErrorStatusPointer(const char_T **rt_errorStatus);

 private:
  rtwdemo_func_dinteg_cpp_DW rtwdemo_func_dinteg_cpprtDW;

  const real_T *rtwdemo_func_dinteg_cprtrtu_In1;
  const real_T *rtwdemo_func_dinteg_cprtrtu_In2;
  real_T *rtwdemo_func_dinteg_crtrty_Out2;
  real_T *rtwdemo_func_dinteg_crtrty_Out1;

  rtwdemo_func_dinteg_cp_RT_MODEL rtwdemo_func_dinteg_cpprtM;

  void rtwdemo_func_dinteg_c_func_calc(real_T rtu_u, real_T *rty_y);
};
```

Limitations

These code generation limitations apply to export-function models that include multiple instances of a Simulink Function block.

- You must set the model configuration parameter **Pass root-level I/O as** to `Part of model data structure`.
- Generated code is compatible with single-threaded execution only. To avoid race conditions for shared signal data, invoke instances of the function code from the same execution thread.
- You cannot:
  - Generate code if the export-function model includes model variants.
  - Enable the external mode data exchange interface.

See Also

More About

- “Simulink Functions Overview”
- “Simulink Functions”
- “Simulink Function Blocks in Referenced Models”
- “Storage Classes and Reentrant, Multi-Instance Models and Components”
- “Simulink Function Blocks and Code Generation” on page 8-6

Generate Reentrant Code from Subsystems

You can reduce the amount of code that the code generator produces for identical atomic subsystems by configuring the subsystems as reusable functions that pass data as arguments (for example, `rtB_*` for block input and output, `rtDW_*` for continuous states, and `rtP_*` for parameters). By default, the code generator produces subsystem code that communicates with other code by sharing access to global data structures that reside in shared memory. By passing data as arguments, the code can be reentrant. Each instance of the code maintains its own unique data.

To configure subsystems for reusability and reentrancy, configure mask and block parameters of the identical subsystems the same way. The code generator performs a checksum to determine whether subsystems are identical and whether the code is reusable. If the code is reusable, the code generator produces a single instance of reusable, reentrant function code.

To configure the subsystem block parameters:

1. If a subsystem is virtual, select block parameter “Treat as atomic unit” to enable function packaging parameters.
2. Set block parameter “Function packaging” to `Reusable function`. The code generator produces a separate function with arguments for each subsystem. Selecting `Reusable function` also enables additional parameters that you can use to control the names of the function and file that the code generator produces for the subsystem code.
3. Select a value for block parameter “Function name options”. To generate reusable, reentrant code, specify the same setting for identical subsystems.

   - To let the code generator determine the function name, select `Auto`.
   - To use the subsystem name or, for a library block, the name of the library block, select `Use subsystem name`.
   - To display block parameter “Function name” and enter a unique, valid C/C++ function name, select `User specified`.

   If multiple instances of an identical subsystem exist within a model reference hierarchy, select `Auto`.

   If you are using Embedded Coder, you can use identifier format controls. See “Identifier Format Control” on page 55-17.
4. Set “File name options”. To generate reusable, reentrant code, specify the same setting for identical subsystems.

   - To let the code generator determine file naming select `Auto`.
   - To use the subsystem name or, for a library block, the name of the library block select `Use subsystem name`.
   - To use the function name, as specified by block parameter **Function name options** select `Use function name`.
   - To display block parameter “File name (no extension)” and enter a file name, excluding the extension (for example, `.c` or `.cpp`) select `User specified`.

   Other considerations:

   - If multiple instances of an identical subsystem exist within a model reference hierarchy, select `Auto`.

- If the code generator does not generate a separate file for a subsystem, the function code is placed in the file generated from the subsystem's parent system. If the parent is the model itself, the code generator places the function code in *model*.c or *model*.cpp.
- If your generated code is under source control, specify a value other than Auto. This specification prevents the generated file name from changing when you modify and rebuild the model.
- If you select Use subsystem name, the code generator mangles the subsystem file name if the model contains Model blocks, or if a model reference target is being generated for the model. In these situations, the code generator uses the file name *modelsubsystem*.c.

If subsystem A has mask parameter b and K and subsystem B has mask parameters c and K, code reuse is not possible. The code generator produces separate functions for subsystems A and B. If you set block parameters for subsystems A and B differently, code reuse is not possible.

Function Reuse in Generated Code

This example shows how to configure an atomic subsystem for generating reusable code. To specify that the code generated for a subsystem execute as an atomic unit, on the Block Parameters dialog box, select the **Treat as atomic unit** parameter. That parameter enables the **Function Packaging** parameter on the **Code Generation** tab. The **Function Packaging** parameter has these four settings:

- Inline: Inline the subsystem code
- Nonreusable function: Function with I/O passed as global data
- Reusable function: Function with I/O passed as function arguments
- Auto: Let Simulink Coder optimize based on context

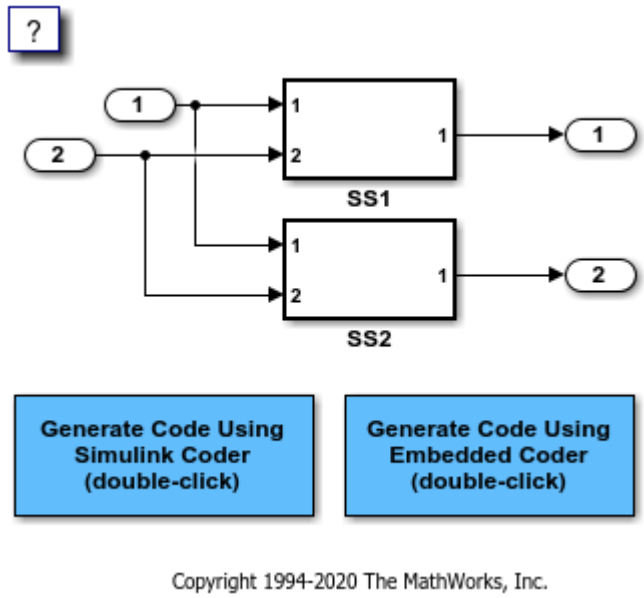
The Reusable function and Auto settings permit the code generator to reuse subsystem code. The Reusable function and Nonreusable function settings enable the **Function name options**, **Function name**, and **File name options** parameters.

If you have an Embedded Coder® license, you can configure a nonreusable subsystem to accept arguments.

Example Model

The rtwdemo_ssreuse model contains two identical subsystems, SS1 and SS2. For these subsystems, the **Function packaging** parameter is set to Reusable function, and the **Function name** parameter is myfun. The subsystems are parameterized masked subsystems. To see the contents of the masked subsystems, right-click the subsystem blocks and select **Mask > Look Under Mask**.

```
model = 'rtwdemo_ssreuse';
open_system(model);
```



Generate and Inspect Code

Create a temporary folder for the build and inspection process. Open the **Simulink Coder** or **Embedded Coder** app. Then, generate and inspect the code.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================
rtwdemo_ssreuse  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.727s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_ssreuse_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   */
```

```
  myfun(rtwdemo_ssreuse_U.In1, rtwdemo_ssreuse_U.In2, rtwdemo_ssreuse_P.T1Data,
        rtwdemo_ssreuse_P.T1Break, &rtwdemo_ssreuse_B.SS1);

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outport: '<Root>/Out1' */
  rtwdemo_ssreuse_Y.Out1 = rtwdemo_ssreuse_B.SS1.LookupTable;

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   */
  myfun(rtwdemo_ssreuse_U.In1, rtwdemo_ssreuse_U.In2, rtwdemo_ssreuse_P.T2Data,
        rtwdemo_ssreuse_P.T2Break, &rtwdemo_ssreuse_B.SS2);

  /* End of Outputs for SubSystem: '<Root>/SS2' */

  /* Outport: '<Root>/Out2' */
  rtwdemo_ssreuse_Y.Out2 = rtwdemo_ssreuse_B.SS2.LookupTable;
}
```

In the model step function, there are two calls to the reusable function, `myfun`. The mask parameters, `T1Break`, `T1Data`, `T2Break`, and `T2Data`, are function arguments.

Change the **Function Packaging** parameter to `Inline`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Inline')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Inline')
```

Generate and inspect the code.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
================================================================================
rtwdemo_ssreuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.379s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_ssreuse_step(void)
{
  real_T Out1_tmp;

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */
```

```
  /* Sum: '<S1>/Sum' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<S2>/Sum'
   */
  Out1_tmp = rtwdemo_ssreuse_U.In1 + rtwdemo_ssreuse_U.In2;

  /* End of Outputs for SubSystem: '<Root>/SS2' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Lookup_n-D: '<S1>/Lookup Table'
   *  Sum: '<S1>/Sum'
   */
  rtwdemo_ssreuse_Y.Out1 = look1_binlx(Out1_tmp, rtwdemo_ssreuse_P.T1Break,
    rtwdemo_ssreuse_P.T1Data, 10U);

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  /* Outport: '<Root>/Out2' incorporates:
   *  Lookup_n-D: '<S2>/Lookup Table'
   */
  rtwdemo_ssreuse_Y.Out2 = look1_binlx(Out1_tmp, rtwdemo_ssreuse_P.T2Break,
    rtwdemo_ssreuse_P.T2Data, 10U);

  /* End of Outputs for SubSystem: '<Root>/SS2' */
}
```

In the model step function, the subsystem code is inlined.

Change the **Function Packaging** parameter to `Nonreusable function`. For `SS2` , change the **Function name** parameter to `myfun2`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Nonreusable function')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Nonreusable function')
set_param('rtwdemo_ssreuse/SS2','RTWFcnName','myfun2')
```

Generate and inspect the code.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_ssreuse
### Successful completion of build procedure for: rtwdemo_ssreuse

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
================================================================================
rtwdemo_ssreuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.175s
```

```
cfile=fullfile(cgDir, 'rtwdemo_ssreuse_grt_rtw', 'rtwdemo_ssreuse.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
```

```
void rtwdemo_ssreuse_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/SS1' */
  myfun();

  /* End of Outputs for SubSystem: '<Root>/SS1' */

  /* Outputs for Atomic SubSystem: '<Root>/SS2' */
  myfun2();

  /* End of Outputs for SubSystem: '<Root>/SS2' */
}
```

The model step function contains calls to the functions `myfun` and `myfun2`. These functions have a void-void interface.

Change the **Function Packaging** parameter to `Auto`.

```
set_param('rtwdemo_ssreuse/SS1','RTWSystemCode','Auto')
set_param('rtwdemo_ssreuse/SS2','RTWSystemCode','Auto')
```

For the `auto` setting, Simulink Coder chooses the optimal format. For this model, the optimal format is a reusable function.

Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Limitations

Code reuse for subsystems is an optimization feature and is not guaranteed. The following list of limitations on code reuse in subsystems is nonexhaustive.

- The code generator uses a checksum to determine whether subsystems are identical and reusable. Subsystem code is not reused if:
  - In blocks and data objects, you use symbols to specify dimensions.
  - A port has different sample times, data types, complexity, frame status, or dimensions across subsystems.
  - The output of a subsystem is marked as a global signal.
  - Subsystems contain identical blocks with different names or parameter settings.
  - The output of a subsystem is connected to a Merge block, and the output of the Merge block is configured with a custom storage class that is implemented in the C code as nonaddressable memory (for example, `BitField`).
  - The input of a subsystem is nonscalar and is configured with a custom storage class that is implemented in the C code as nonaddressable memory.
  - A masked subsystem has a parameter that is nonscalar and is configured with a custom storage class that is implemented in the C code as nonaddressable memory.
  - A function-call subsystem uses mask parameters when you set the model configuration parameter Default parameter behavior to `Tunable`. To reuse the masked function-call

subsystem, place it inside a new atomic subsystem without a mask. Then move the Trigger block from the masked subsystem into the atomic subsystem.

- A block in a subsystem uses a partially tunable expression. Some partially tunable expressions disable code reuse.

  Partially tunable expressions are expressions that contain one or more tunable variables and an expression that is not tunable. For example, suppose that you create the tunable variable `K` with value `15.23` and the tunable variable `P` with value `[5;7;9]`. The expression `K+P'` is a partially tunable expression because the expression `P'` is not tunable. For more information about tunable expression limitations, see “Tunable Expression Limitations”.

- For subsystems that contain S-function blocks that are reusable, the blocks must meet the requirements listed in “S-Functions for Code Reuse” on page 13-81.
- If you select `Reusable function`, and the code generator determines that you cannot reuse the code for a subsystem, it generates a separate function that is not reused. The code generation report might show that the separate function is reusable, even if only one subsystem uses it. If you prefer that the code generator inline subsystem code in such cases (rather than deployed as functions), set “Function packaging” to `Auto`.
- If a reusable subsystem uses a shared local data store and you configure default mapping for model data elements, leave the default storage class mapping for category **Shared local data stores** set to `Default`.
- Use of these blocks in a subsystem can prevent the subsystem code from being reused:
  - Scope blocks (with data logging enabled)
  - S-Function blocks that fail to meet certain criteria (see “S-Functions for Code Reuse” on page 13-81)
  - To File blocks (with data logging enabled)
  - To Workspace blocks (with data logging enabled)
- For reusable library subsystems (subsystems shared across reference models), the code generator uses a checksum to determine whether subsystems are identical. The code generator places the reusable library subsystem code in the shared utilities folder, and the reusable code is independent of the generated code of the top model or the reference model. For example, the reusable library subsystem code does not include *`model`*`.h` or *`model`*`_types.h`.

  Reusable code that the code generator places in the shared utilities folder and is dependent on the model code does not compile. If the code generator determines that the reusable library subsystem code is dependent on the model code, the reusable subsystem code is not placed in the shared utilities folder. The code generator produces code that is dependent on the model code when the reusable library subsystem:

  - Contains a block that uses time-related functionality, such as a Step block, or continuous time or multirate blocks.
  - Contains one or more Model blocks.
  - Contains a subsystem that is not inlined or a reusable library subsystem.
  - Contains a signal or parameter with a storage class that places the data declaration in a file under *`model`*`_ert_rtw` (For example, `ExportedGlobal` storage class places data in *`model`*`.c`)
  - Contains a user-defined storage class like Enumeration, Simulink.Signal, Simulink.Parameter, and so on where **Data Scope** is not set to `Exported`. The code generator might place the type definition in *`model`*`_types.h`.

  - Is a variant subsystem that generates preprocessor conditionals. The code generator places preprocessor directives that define the variant objects in `model_types.h`
- You cannot reuse a subsystem if there are multiple instances of the same subsystem in the model and one subsystem is fed by a Constant block but the other subsystem is not. To work around this limitation, insert a Signal Conversion block before the subsystems.
- Two reusable subsystems that have different checksums cannot share the same function in the generated code. When you specify the same function name for both subsystems in **Block Parameters (Subsystem) > Code Generation > Function name**:
  - If the subsystems are library-linked, the code generator mangles the function name to generate unique function names. The code generator uses the subsystem naming rules to generate the function names.
  - If the subsystems are not library-linked, the code generator issues an error.

The preceding limitations also apply for library-based code generation. For more information, see “Library-Based Code Generation for Reusable Library Subsystems” on page 9-2.

See Also

More About

- “What Is Reentrant Code?” on page 8-3
- “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36
- “Generate Subsystem Code as Separate Function and Files” on page 4-7
- “Storage Classes and Reentrant, Multi-Instance Models and Components”

Generate Reusable Code from Library Subsystems Shared Across Models

What Is a Reusable Library Subsystem?

A reusable library subsystem is a subsystem included in a library that is configured for reuse. You must define a subsystem in a library and configure it for reuse to reuse the subsystem across models.

To reuse common functionality, you can include multiple instances of a subsystem:

- Within a single model, which is a top model or part of model reference hierarchy
- Across multiple referenced models in a model reference hierarchy
- Across multiple top models that contain Model blocks
- Across multiple top models that do not include Model blocks

The code generator uses checksums to determine reusability. There are cases when the code generator cannot reuse subsystem code.

For incremental code generation, if the reusable library subsystem changes, a rebuild of itself and its parents occurs. During the build, if a matching function is not found, a new instance of the reusable function is generated into the shared utilities folder. If a different matching function is found from previous builds, that function is used, and a new reusable function is not emitted.

For subsequent builds, unused files are not replaced or deleted from your folder. During development of a model, when many obsolete shared functions exist in the shared utilities folder, you can delete the folder and regenerate the code. If all instances of a reusable library subsystem are removed from a model reference hierarchy and you regenerate the code, the obsolete shared functions remain in the shared utilities folder until you delete them.

If a model changes such that the change might cause different generated code for the subsystem, a new reusable function is generated. For example, model configuration parameters that modify code comments might cause different generated code for the subsystem even if the reusable library subsystem did not change.

Embedded Coder users can generate code from a library that contains subsystems are configured for reuse. For more information, see “Library-Based Code Generation for Reusable Library Subsystems” on page 9-2.

Reusable Library Subsystem Code Placement and Naming

The generated code of a reusable library subsystem is independent of the generated code of the model. Code for the reusable library subsystem is generated to the shared utility folder, `slprj/target/_sharedutils`, instead of the model reference hierarchy folders. The generated code for the supporting types, which are generated to the `.h` file, are also in the shared utilities folder.

For unique naming, reusable function names have a checksum appended to the reusable library subsystem name. For example, the code and files for a subsystem, `SS1`, which links to a reusable library subsystem, `RLS`, might be:

- Function name: `RLS_mgdjlngd`

- File name: `RLS_mgdjlnd.c` and `RLS_mgdjlnd.h`

To configure a reusable library subsystem, set the Subsystem parameters as listed here:

- Select **Treat as an atomic unit**.
- On the **Code Generation** tab:
  - Set **Function packaging** to `Reusable function`.
  - Set the **Function name options** and **File name options** parameters based on this table. This table summarizes how the code generator produces code and highlights the **File name options** in bold that enable you to generate code from reusable library subsystems. For the purpose of this table, assume that the Subsystem block is named `mySubsystem` and the `File name` value for the `User specified` option is `myFunction`.

Function Name Options	File Name Options	Generated Folder Location	Generated Function Name	Generated File Names
Auto	**Auto**	/slprj/*target*/_sharedutils	mySubsystem_*checksum*	mySubsystem_*checksum*.c, mySubsystem_*checksum*.h
	Use subsystem name	/slprj/*target*/_sharedutils	mySubsystem_*checksum*	mySubsystem_*checksum*.c, mySubsystem_*checksum*.h
	Use function name	/slprj/*target*/_sharedutils	mySubsystem_*checksum*	mySubsystem_*checksum*.c, mySubsystem_*checksum*.h
	User specified	*model*_target_rtw	*model*_Subsystem	myFile.h
Use subsystem name	Auto	*model*_target_rtw	*model*_Subsystem	Generated code included in *model*.h
	Use subsystem name	/slprj/*target*/_sharedutils	mySubsystem_*checksum*	mySubsystem_*checksum*.c, mySubsystem_*checksum*.h
	Use function name	/slprj/*target*/_sharedutils	mySubsystem_*checksum*	mySubsystem_*checksum*.c, mySubsystem_*checksum*.h
	User specified	*model*_target_rtw	*model*_Subsystem	myFile.h

Function Name Options	File Name Options	Generated Folder Location	Generated Function Name	Generated File Names
`User specified`	**`Auto`** (This specification is supported for one unique instance of the subsystem in the model.)	`/slprj/`*`target`*`/_sharedutils`	`myFunction`	`myFunction.c`, `myFunction.h`
	`Use subsystem name`	*`model_target`*`_rtw`	`myFunction`	`mySubsystem.c`, `mySubsystem.h`
	`Use function name` (This specification is supported for one unique instance of the subsystem in the model.)	`/slprj/`*`target`*`/_sharedutils`	`myFunction`	`myFunction.c`, `myFunction.h`
	`User specified` (This specification is supported if the user-specified function name and file name are same and only for one unique instance of the subsystem in the model.)	`/slprj/`*`target`*`/_sharedutils`	`myFunction`	`myFunction.c`, `myFunction.h`

For more information about the Subsystem block parameters, see Subsystem.

Configure Models That Include Reusable Library Subsystems

For a model to use a reusable library subsystem, you must configure the model differently depending on whether the model is a reference model or top model. If the subsystem is in a referenced model hierarchy, set the configuration parameter, “Shared code placement” to `Auto`. Otherwise, for each model that uses the subsystem, set the model configuration parameter **Shared code placement** to `Shared location`.

If a reusable library subsystem uses a shared local data store and you configure default mapping for model data elements, leave the default storage class mapping for category **Shared local data stores** set to **Default**.

In a model reference hierarchy, if an instance of the reusable library subsystem is in the top model, then on the **Model Referencing** pane of the Configuration Parameters dialog box, you must select the **Pass fixed-size scalar root input by value for code generation** parameter. If you do not select the parameter, a separate shared function is generated for the reusable library subsystem instance in the top model, and a reusable function is generated for instances in the referenced models.

If a reusable library subsystem is connected to the root Outport, reuse does not happen with identical subsystems that are not connected to the root Outport. However, you can set **Pass reusable system outputs as** to `Individual arguments` on the **Optimizations** pane to make sure that reuse occurs between these subsystems. This parameter requires an Embedded Coder license.

For more information on creating a library, see “Custom Libraries”. For an example of creating a reusable library subsystem, see “Generate Reusable Code for Subsystems Shared Across Models”.

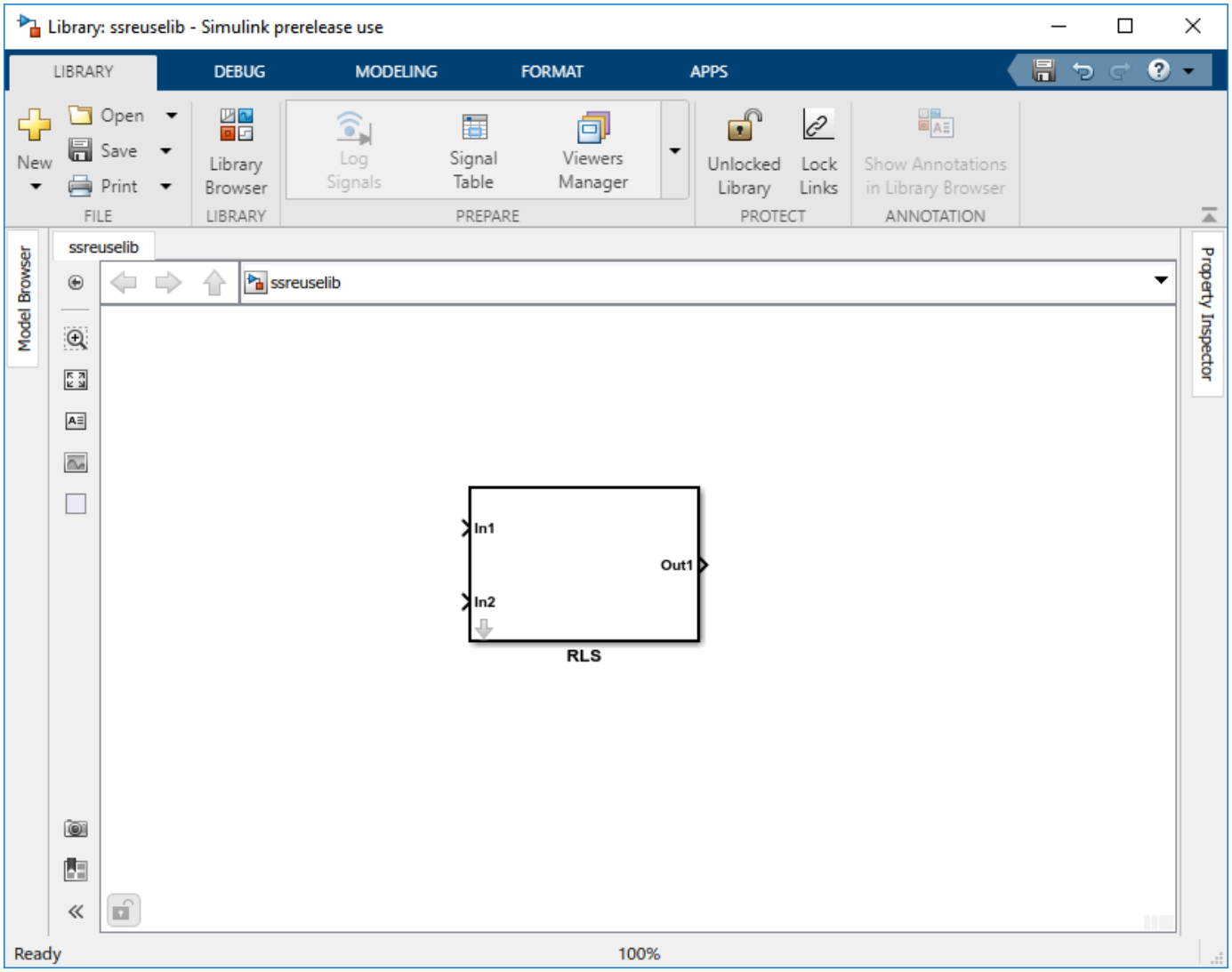
Generate Reusable Code for Subsystems Shared Across Models

This example shows how to configure a reusable library subsystem and generate a reusable function for a subsystem shared across referenced models. The result is reusable code for the subsystem, which is generated to the shared utility folder (`slprj/`*`target`*`/_sharedutils`).

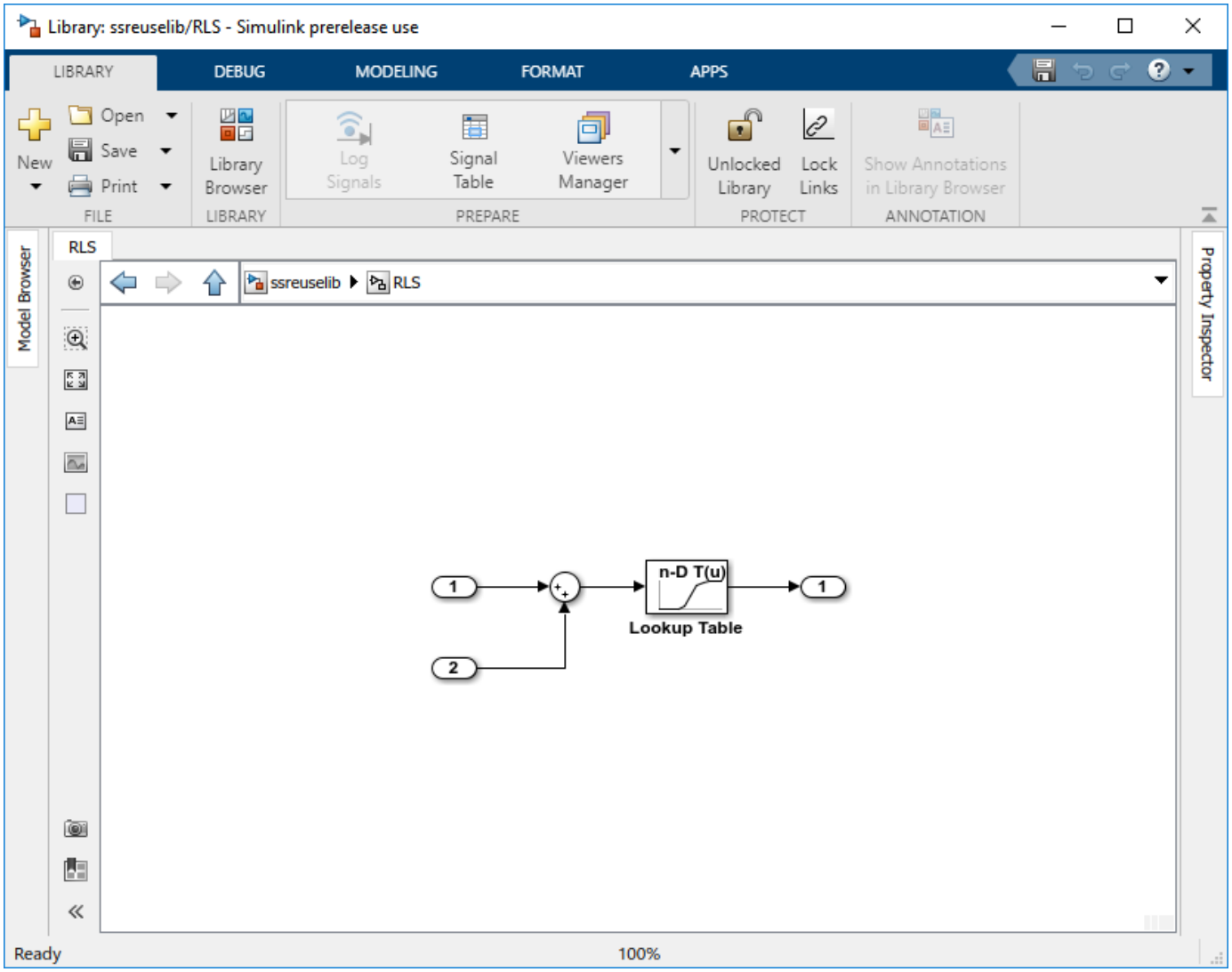
- “Create a reusable library subsystem.” on page 8-39
- “Create the example model.” on page 8-42
- “Set configuration parameters of the top model.” on page 8-43
- “Create and propagate a configuration reference.” on page 8-43
- “Generate and view the code.” on page 8-44

Create a reusable library subsystem.

1. In the Simulink Editor, on the **Simulation** tab, select **New > Library**. Select **Blank Library** from the Simulink Start Page window. Open `rtwdemo_ssreuse` to copy and paste subsystem `SS1` into the Library Editor. This action loads the variables for `SS1` into the base workspace. Rename the subsystem block to `RLS`.



2 Click the Subsystem block and press **Ctrl+U** to view the contents of subsystem RLS.



3 To configure the subsystem, in the Library editor, right-click RLS. In the context menu, select **Block Parameters (Subsystem)**. In the Subsystem Parameters dialog box, choose the following options:

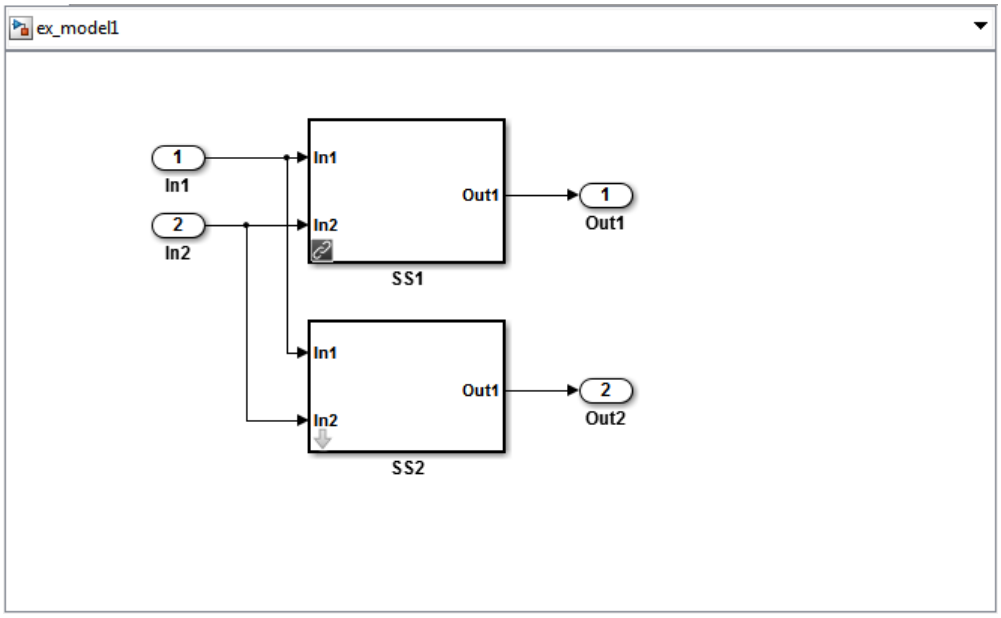
- Select **Treat as an atomic unit**.
- On the **Code Generation** tab:
  - Set **Function packaging** to `Reusable function`.
  - Set **Function name options** to `User specified` and verify that the **Function name** is set to `myfun`.
  - Set **File name options** to `Auto`.

4 Click **Apply** and **OK**.

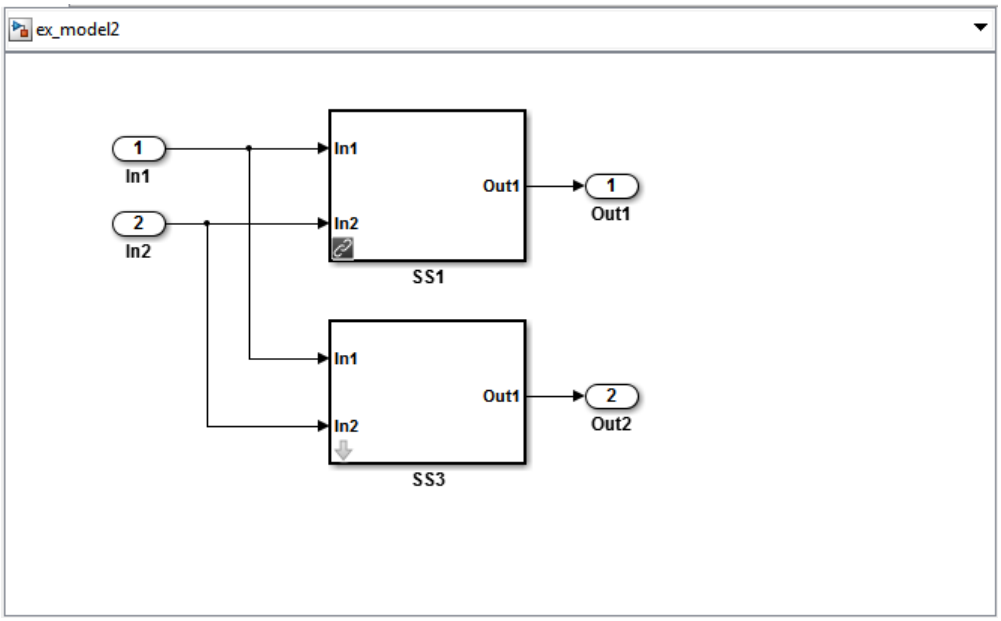
5 Save the reusable library as `ssreuselib`, which creates a file, `ssreuselib.slx`.

Create the example model.

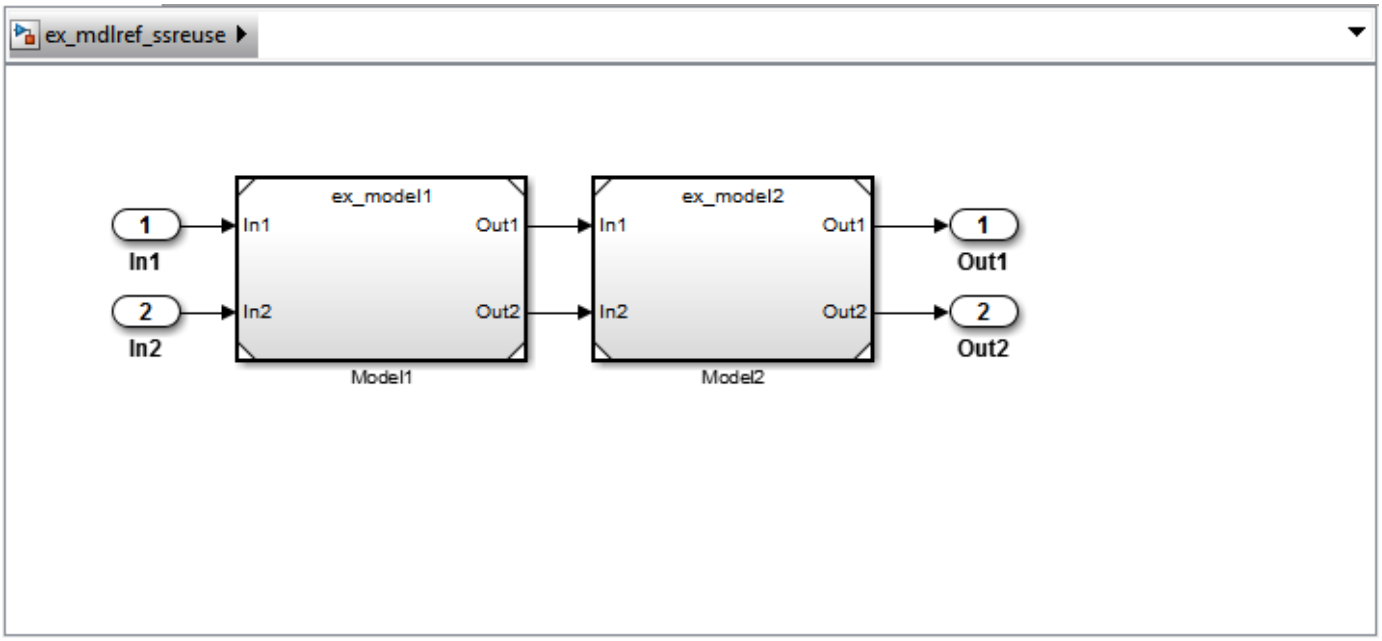
1 Create a model which includes one instance of RLS from `ssreuselib`. Name this subsystem SS1. Add another subsystem and name it SS2. Name the model `ex_model1`.



2 Create another model which includes one instance of RLS from `ssreuselib`. Name this subsystem SS1. Add another subsystem and name it SS3. Name the model `ex_model2`.



3 Create a top model with two model blocks that reference `ex_model1` and `ex_model2`. Save the top model as `ex_mdlref_ssreuse`.



Set configuration parameters of the top model.

1. With model `ex_mdlref_ssreuse` open in the Simulink Editor, on the **C Code** tab, click **Settings** to open the Configuration Parameters dialog box.
2. On the **Solver** pane, specify the **Type** as `Fixed-step`.
3. On the **Model Referencing** pane, click the ellipsis to view the advanced parameters. Then, select **Pass fixed-size scalar root inputs by value for code generation**.
4. On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
5. On the **Code Generation > Interface** pane, set the "Shared code placement" to `Shared location`.
6. On the **Code Generation > Identifiers** pane, set the **Maximum identifier length** to `256`. This step is optional.
7. Click **Apply** and **OK**.

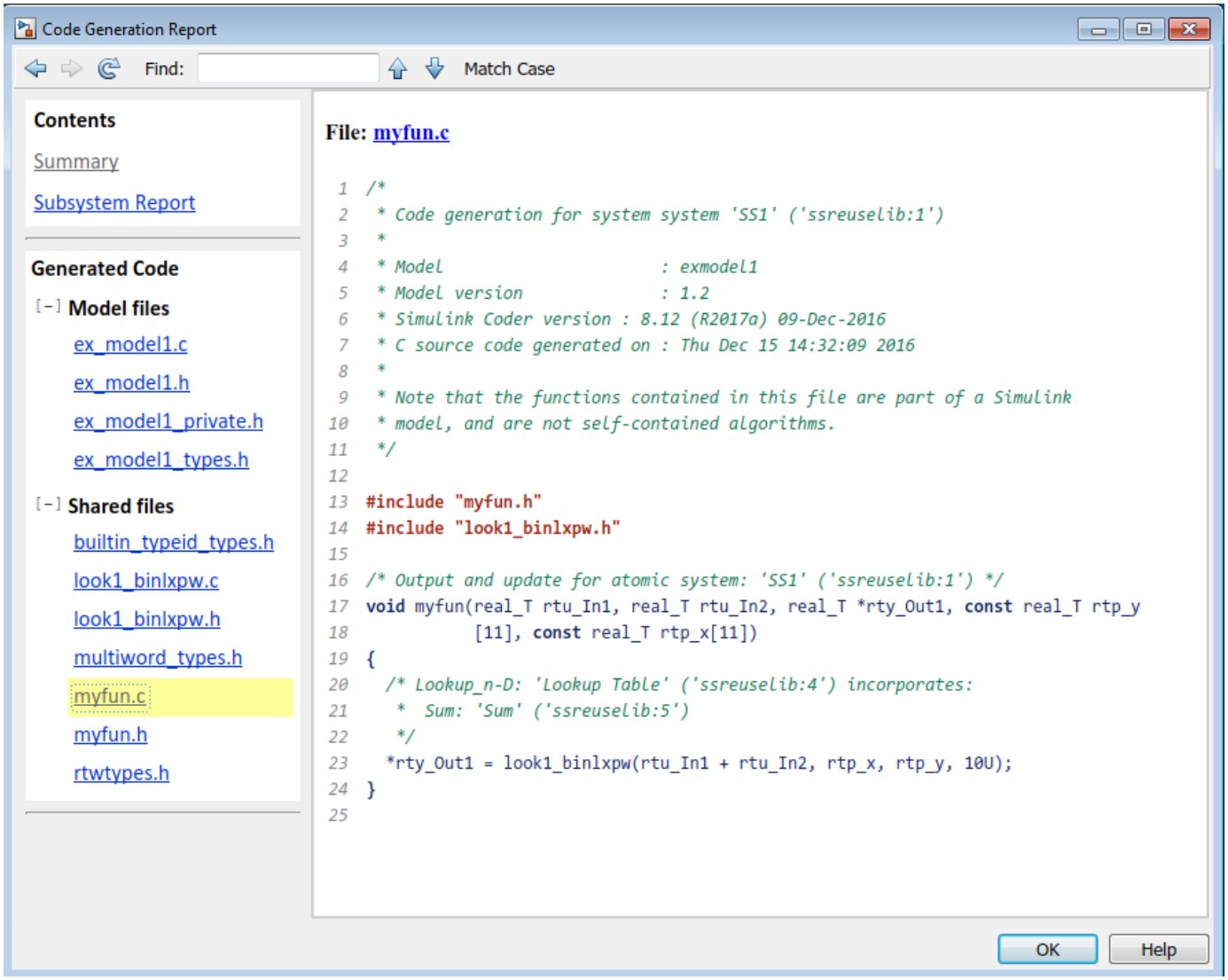
Create and propagate a configuration reference.

1. In the Simulink Editor, on the **Modeling** tab, click **Model Explorer** to open the Model Explorer. In the left navigation column of the Model Explorer, expand the `ex_mdlref_ssreuse` node.
2. Select the Configurations node below the `ex_mdlref_ssreuse` node. In the **Contents** pane, right-click **Configuration** and select `Convert to Configuration Reference`.
3. In the Convert Active Configuration to Reference dialog box, click **OK**. This action converts the model configuration set to a configuration reference, `Simulink.ConfigSetRef`, and creates the configuration reference object, `configSetObj`, in the base workspace.
4. In the left navigation column, right-click **Reference (Active)** and select `Propagate to Referenced Models`.
5. In the Configuration Reference Propagation to Referenced Models dialog box, select the referenced models in the list. Click **Propagate**.

Now, the top model and referenced models use the same configuration reference, `Reference (Active)`, which points to a model configuration reference object, `configSetObj`, in the base workspace. When you save your model, you also need to export the `configSetObj` to a MAT-file.

Generate and view the code.

1. To generate code, in the Simulink Editor, press **Ctrl+B**. After the code is generated, the code generation report opens.
2. To view the code generation report for a referenced model, in the left navigation pane, in the **Referenced Models** section, select `ex_model1`. The code generation report displays the generated files for `ex_model1`.
3. In the left navigation pane, expand the **Shared files**. The code generator uses the reusable library subsystem name. The code for subsystem `SS1` is in `myfun.c` and `myfun.h`.



4. Click **Back** and navigate to the `ex_model2` code generation report. `ex_model2` uses the same source code, `myfun.c` and `myfun.h`, as the code for `ex_model1`. Your subsystem function and file names will be different.

See Also

More About

- “Custom Libraries”
- “Generate Reentrant Code from Subsystems” on page 8-28
- “Generate Subsystem Code as Separate Function and Files” on page 4-7

Generate Reusable Code from Stateflow Atomic Subcharts

Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1. Open the library model that contains your atomic subchart.
2. Unlock the library.
3. Right-click the library chart and select **Block Parameters**.
4. In the dialog box, specify the following parameters:
   a. On the **Main** tab, select parameter **Treat as atomic unit**.
   b. On the **Code Generation** tab, set parameter **Function packaging** to `Reusable function`.
   c. Set **File name options** to `User specified`.
   d. For **File name**, enter the name of the file without an extension.
   e. Click **OK** to apply the changes.
5. (OPTIONAL) Customize the generated function names for atomic subcharts:
   a. Set model configuration parameter **System target file** to `ert.tlc`.
   b. For model configuration parameter **Subsystem methods**, specify the format of the function names using a combination of the following naming rule tokens:
      - `$R` — root model name
      - `$F` — type of interface function for the atomic subchart
      - `$N` — block name
      - `$H` — subsystem index
      - `$M` — name-mangling text
   c. Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

1. In your chart, right-click the atomic subchart and select **Properties**.
2. In the dialog box, specify the following properties:
   a. Set property **Code generation function packaging** to `Reusable function`.
   b. Set **Code generation file name options** to `User specified`.
   c. For **Code generation file name**, enter the name of the file without extension.
   d. Click **OK** to apply the changes.
3. (OPTIONAL) Customize the generated function names for atomic subcharts:

a Set model configuration parameter **System target file** to `ert.tlc`.

b For model configuration parameter **Subsystem methods**, specify the format of the function names using a combination of the following naming rule tokens:

- `$R` — root model name
- `$F` — type of interface function for the atomic subchart
- `$N` — block name
- `$H` — subsystem index
- `$M` — name-mangling text

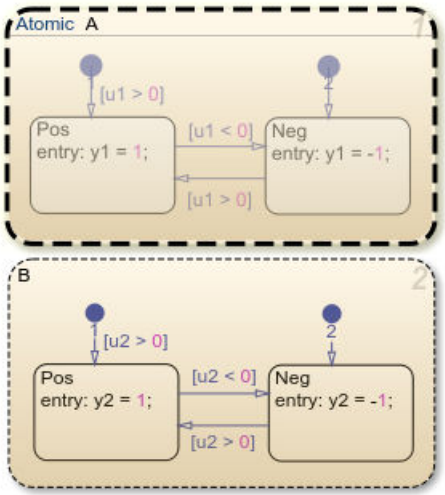
c Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for the atomic subchart. For more information, see “Generate Code from Atomic Subcharts” on page 12-5.

Generate Reusable Code for Unit Testing

Convert a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Group & Subchart** > **Atomic Subchart**. State A changes to an atomic subchart:



Set Up a Standalone C File for the Atomic Subchart

1 Open the properties dialog box for A.

2 Set property **Code generation function packaging** to `Reusable function`.

3 Set **Code generation file name options** to `User specified`.

4 For **Code generation file name**, enter `saturator` as the name of the file.

5 Click **OK**.

Set Up the Code Generation Report

1 Set model configuration parameter **System target file** to `ert.tlc`.

2. Select model configuration parameters **Create code generation report** and **Open report automatically**.
3. Select parameters **Code-to-model** and **Model-to-code**.
4. Click **Apply**.

Customize the Generated Function Names

1. Set model configuration parameter **Subsystem methods** to the format scheme `$R$N$M$F`, where:

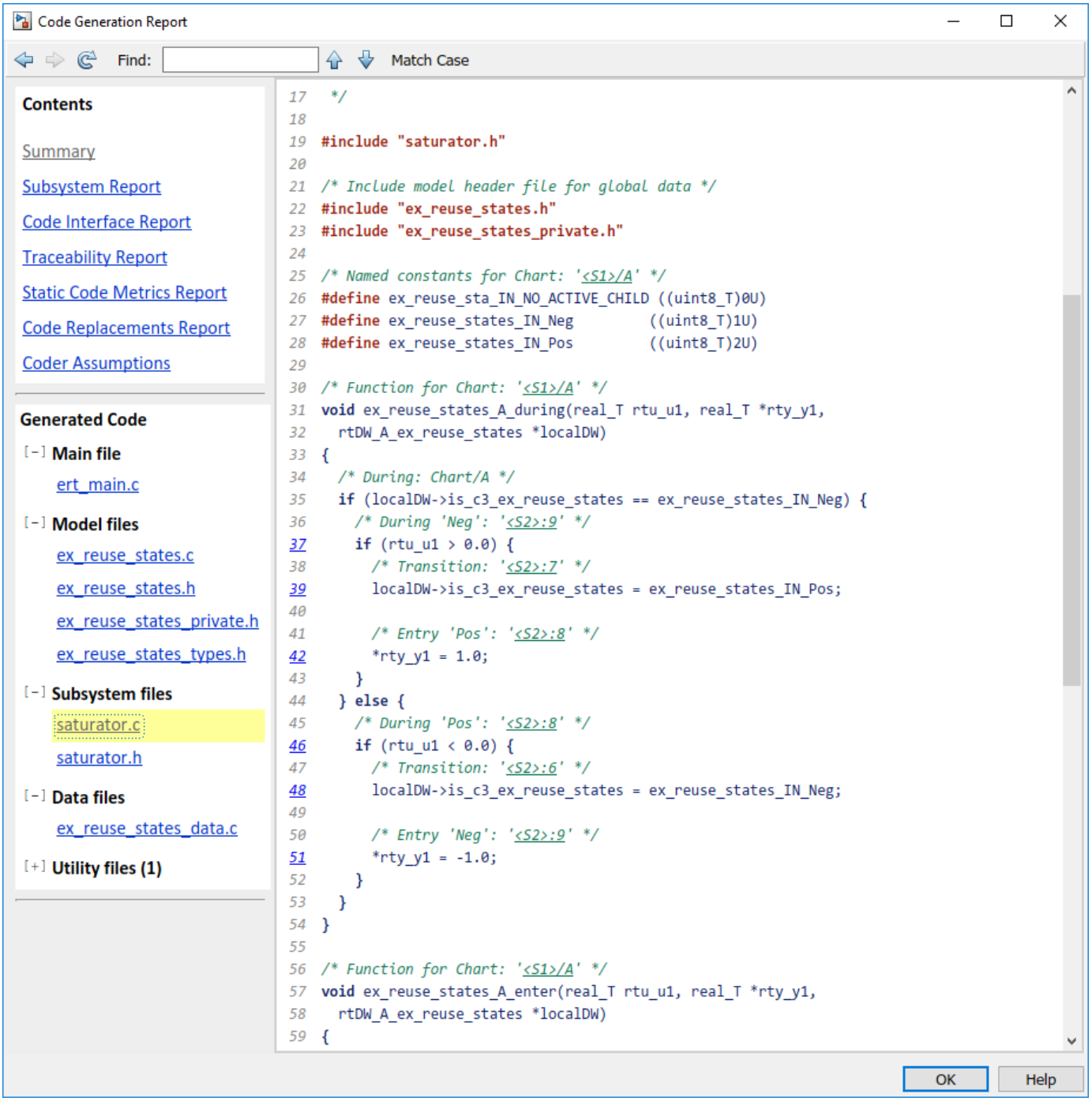
   - `$R` is the root model name.
   - `$N` is the block name.
   - `$M` is the mangle token.
   - `$F` is the type of interface function for the atomic subchart.

   For more information, see Subsystem methods.
2. Click **Apply**.

Generate Code for Only the Atomic Subchart

To generate code for your model, press **Ctrl+B**. In the code generation report that appears, you see a separate file that contains the generated code for the atomic subchart.

To inspect the code for `saturator.c`, click the hyperlink in the report to see the following code:



Line 28 shows that the `during` function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format scheme `$R$N$M$F` specified for **Subsystem methods**:

- `$R` is the root model name, `ex_reuse_states`.
- `$N` is the block name, `A`.
- `$M` is the mangle token, which is empty.

- `$F` is the type of interface function for the atomic subchart, `during`.

Note The line numbers shown can differ from the numbers that appear in your code generation report.

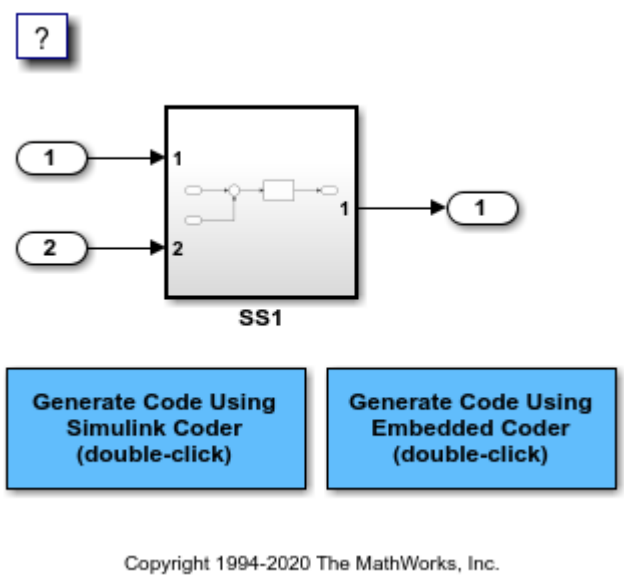
File Packaging for Atomic Subsystems (Code Only)

This model shows how to partition an atomic subsystem into its own source file. Once a subsystem is specified as atomic, on the Subsystem Block Parameter dialog, on the **Code Generation** tab, you can specify how the subsystem is represented in the generated code using the **Function packaging** parameter:

- `Inline`: an Inlined function
- `Function`: function with I/O passed as global data
- `Reusable function`: function with I/O passed as function arguments
- `Auto`: Simulink® Coder™ optimizes based on context

With the `Function` and `Reusable function` options, you can specify a function and file name with **Function name** and **File name**, respectively. In the model, `rtwdemo_filepart`, `SS1` subsystem is configured as `Reusable function` with function name `myfun` and file name `myfile`.

```
open_system('rtwdemo_filepart')
```



This capability only partitions the code for a subsystem. The data is declared and owned by the parent model. For full system encapsulation (i.e., data and functions), use Model Reference.

```
bdclose('rtwdemo_filepart');
```

See Also

More About

- “Model Finite State Machines by Using Stateflow Charts” (Stateflow)
- “Encapsulate Modal Logic by Using Subcharts” (Stateflow)

Generate Shared Utility Code

When to Generate Shared Utility Code

Blocks in a model can require common functionality to implement their algorithms. Consider modularizing this functionality into standalone support or helper functions. This approach can be more efficient than inlining the code for the functionality for each block instance. To decide about using a library or a shared utility, consider:

- Packaging functions that can have multiple callers into a library when the functions are defined statically. That is, before you use the code generator to produce code for your model, the function source code exists in a file.
- Packaging functions that can have multiple callers as shared utilities (produced during code generation) when the functions cannot be defined statically. For example, several model- and block-specific properties specify which functions are used and their behavior. Also, these properties determine type definitions (for example, `typedef`) in shared utility header files. The number of possible combinations of properties that determine unique behavior make it impractical to define statically the possible function files before code generation.

Configure Naming of Generated Functions

Configure a default naming rule for generated shared utility functions by using one of these methods.

Configuration	Action
A new model or an existing model for which you have mapped default customization settings by using the **Code Mappings Editor – C** or code mappings programming interface	In the **Embedded Coder Dictionary**, define a function customization template that specifies a custom function naming rule. Then, in the Code Mappings editor or by using the code mappings programming interface, map the **Shared utility** function category to that template. For more information, see “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15.
A model created in a release earlier than R2018a and that has not been configured with the **Code Mappings Editor – C** or code mappings programming interface	Set the model configuration parameter **Shared utilities identifier format** (`CustomSymbolsStrUtil`) to a custom function naming rule. For more information, see “Identifier Format Control” on page 55-17.

Once you use the **Code Mappings Editor – C** or code mappings programming interface to configure a model, setting model configuration parameter **Shared utilities identifier format** does not impact code generation. This setting is not applicable for models configured with these specifications.

System Target File	Programming Language
`ert.tlc`	C++
`autosar.tlc`	C
`autosar_adaptive.tlc`	C++

The naming rule for shared utility functions must include the conditional checksum token `$C`. To customize the length of the checksum that the code generator produces, use parameter **Shared**

checksum length (`SharedChecksumLength`) . Increasing the length of the checksum reduces the probability of clashes.

This example shows the customization of shared utility function names for C code generation.

1. Open the example model `rtwdemo_configdefaults` and the **Embedded Coder** app.
2. Set model configuration parameter **Shared code placement** to `Shared location`.
3. Open the Embedded Coder Dictionary. On the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.
4. On the **Function Customization Template** tab, below the table, click **Create** to add a new function customization template.
5. For the function customization template, specify these configurations:
   - **Name**: `exSharedUtility`
   - **Function Name**: `mymodel_$N$C`

     You can customize this format by using a string. You must include token `$C`.
6. Open the Code Mappings editor. On the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
7. On the **Function Defaults** tab, for the **Shared Utility** category, configure **Function Customization Template** as `exSharedUtility`.
8. Generate code and a code generation report.
9. In the left navigation pane of the report, review the list of generated files for shared utilities under **Shared files**.

Control Placement of Shared Utility Code

Control placement of shared utility code with the model configuration parameter **Shared code placement**. The default option value is `Auto`. For this setting, the code generator places required code for fixed-point and other utilities in the *`model`*`.c` file, the *`model`*`.cpp` file, or a separate file in the build folder (for example, `vdp_grt_rtw`) if a model does not contain existing shared utility code or one of these blocks:

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

If a model contains one or more of the preceding blocks, the code generator creates and uses a shared utilities folder within `slprj`. The naming convention for shared utility folders is `slprj/`*`target`*`/_sharedutils`. *`target`* is `sim` for simulations with Model blocks or the name of the system target file for code generation.

```
slprj/sim/_sharedutils      % folder used with simulation
slprj/grt/_sharedutils      % folder used with grt.tlc STF
slprj/ert/_sharedutils      % folder used with ert.tlc STF
slprj/mytarget/_sharedutils % folder used with mytarget.tlc STF
```

To force a model build to use the `slprj` folder for shared utility generation, even when the current model does not contain existing shared utility code or one of the preceding blocks, set parameter **Shared code placement** to `Shared location`. The code generator places utilities under the `slprj` folder rather than in the normal build folder. This setting is useful when you are manually combining code from several models, because it prevents symbol collisions between the models.

Control Placement of rtwtypes.h for Shared Utility Code

When the configuration parameter **Data type replacement** is set to `Use coder typedefs`, the generated `rtwtypes.h` header file provides fundamental type definitions, `#define` statements, and enumerations. For more information, see “rtwtypes.h” on page 52-40.

Control placement of rtwtypes.h file by selecting whether the build process uses the shared utilities folder. If the model build uses a shared utilities folder, the build process places `rtwtypes.h` in `slprj/`*`target`*`/_sharedutils`. Otherwise, the software places `rtwtypes.h` in *`model_target`*`_rtw`.

Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `rtwtypes.h` file during code generation. If updates occur, recompile and, depending on your development process, reverify previously generated code. To minimize updates to the `rtwtypes.h` file, make the following changes in the Configuration Parameters dialog box:

- Select model configuration parameter **Support: complex numbers** even if the model does not currently use complex data types. Selecting this parameter protects against a future requirement to add support for complex data types when integrating code.
- Clear parameter **Support non-inlined S-functions**. If you use noninlined S-functions in the model, this option generates an error.
- Clear parameter **Classic call interface**. This parameter setting disables use of the GRT system target file.

Avoid Duplicate Header Files for Exported Data

Exported header files appear in the shared utility folder when:

- You control the file placement of declarations for signals, parameters, and states by defining and applying storage classes.
- The code generator places utility code in a shared location.

For example, you can specify a header file for data by using:

- A storage class that you define in the **Embedded Coder Dictionary** and map to categories of model data in the **Code Mappings Editor – C**.
- The **Code Generation** tab in a Signal Properties dialog box.
- The `HeaderFile` property of a data object. Data objects are objects of the classes `Simulink.Signal` and `Simulink.Parameter`.

If you want the declaration to appear in the file *`model`*`.h`, it is a best practice to leave the header file name unspecified. By default, the code generator places data declarations in *`model`*`.h`.

If you specify *`model`*`.h` as the header file name, and if the code generator places utility code in a shared location, you cannot generate code from the model. The code generator cannot create the file *`model`*`.h` in both the model build folder and the shared utility folder.

Reduce Shared Utility Code Generation with Incremental Builds

You can specify that the model build generates C source files in a shared utilities folder. See “Control Placement of Shared Utility Code” on page 8-53. These files include C source files that contain

function definitions and header files that contain macro definitions. For this discussion, the term functions means functions and macros.

Blocks within the same model and blocks in different models can use a shared function when you use model reference or when you build multiple standalone models from the same start build folder. The code generator produces the code for a given function only once for the block that first triggers code generation. As the product determines the requirement to generate function code for subsequent blocks, it performs a file existence check. If the file exists, the model build does not regenerate the function. The shared utility code mechanism requires that a given function and file name represent the same functional behavior regardless of which block or model generates the function. To satisfy this requirement:

- Model properties that determine function behavior contribute to the shared utility checksum or determine the function and file name.
- Block properties that determine the function behavior also determine the function and file name.

During compilation, makefile rules for the shared utilities folder select compilation of only new C files and incrementally archive the object file into the shared utility library, `rtwshared.lib`, or `rtwshared.a`. Incremental compilation also occurs.

Manage the Shared Utility Code Checksum

When you set the model configuration parameter **Shared code placement** to `Shared location` or when the model contains Model blocks, the code generator places shared code in the shared utilities folder. The build process generates a shared utilities checksum of the code generation configuration for the shared code.

During subsequent code generation, if the checksum file `slprj/`*`target`*`/_sharedutils/checksummap.mat` exists relative to the current folder, the code generator reads that file. The code generator verifies that the current model that you are building has configuration properties that match the checksum of properties from the shared utility model. If mismatches occur between the properties defined in `checksummap.mat` and the current model properties, you see an error. Use the error message to manage the checksum (for example, diagnose and resolve the configuration issues with the current model).

View the Shared Utility Checksum Hash Table

It is helpful to view the property values that contribute to the checksum. This example uses the `rtwdemo_lct_start_term.slx` model. To load the `checksum.mat` file into MATLAB and view the `targetInfoStruct` that defines the checksum-related properties:

1. Open the `rtwdemo_lct_start_term.slx` model. In the Command Window, type:

   ```
   rtwdemo_lct_start_term
   ```

2. Create and move to a new working folder.

   ```
   mkdir C:\Temp\demo
   cd C:\Temp\demo
   ```

3. Save a copy of the model in the folder.
4. Build the model, which is set up to produce shared utilities.
   a. On the **Apps** tab, under **Code Generation**, click **Simulink Coder**.
   b. On the **C Code** tab, in the **Generate Code** section, click the **Build** button.

5. Move to the `_sharedutils` folder created by the build process.

```
cd C:\Temp\demo\slprj\grt\_sharedutils
```

6. Load the `checksummap.mat` file into MATLAB.

```
load checksummap
```

7. Display the contents of `hashTbl.targetInfoStruct` and examine the checksum-related property values.

```
hashTbl.targetInfoStruct
```

For this example, the Command Window displays `hashTbl.targetInfoStruct` for the shared utilities that you generated from the model:

```
ans =

  struct with fields:

                  ShiftRightIntArith: 'on'
              ProdShiftRightIntArith: 'on'
                           Endianess: 'LittleEndian'
                        ProdEndianess: 'LittleEndian'
                          wordlengths: '8,16,32,32,64,32,64,64,64,64'
                      Prodwordlengths: '8,16,32,32,64,32,64,64,64,64'
                       TargetWordSize: '64'
                         ProdWordSize: '64'
                  TargetHWDeviceType: 'Custom Processor->MATLAB Host Computer'
                    ProdHWDeviceType: 'Intel->x86-64 (Windows64)'
                 TargetIntDivRoundTo: 'Zero'
                   ProdIntDivRoundTo: 'Zero'
   UseDivisionForNetSlopeComputation: 'off'
                  PurelyIntegerCode: 'off'
                  PortableWordSizes: 'off'
              SupportNonInlinedSFcns: ''
                 RTWReplacementTypes: ''
                           MaxIdInt8: 'MAX_int8_T'
                           MinIdInt8: 'MIN_int8_T'
                          MaxIdUint8: 'MAX_uint8_T'
                          MaxIdInt16: 'MAX_int16_T'
                          MinIdInt16: 'MIN_int16_T'
                         MaxIdUint16: 'MAX_uint16_T'
                          MaxIdInt32: 'MAX_int32_T'
                          MinIdInt32: 'MIN_int32_T'
                         MaxIdUint32: 'MAX_uint32_T'
                          MaxIdInt64: 'MAX_int64_T'
                          MinIdInt64: 'MIN_int64_T'
                         MaxIdUint64: 'MAX_uint64_T'
                        BooleanTrueId: 'true'
                       BooleanFalseId: 'false'
    TypeLimitIdReplacementHeaderFile: ''
                SharedCodeRepository: ''
                          TargetLang: 'C'
                        TargetLangStd: 'C89/C90 (ANSI)'
                 InstructionSetExtions: 'None'
             PreserveExternInFcnDecls: 'on'
              ImplementImageWithCVMat: 'off'
              EnableSignedRightShifts: 'on'
               EnableSignedLeftShifts: 'on'
```

```
                        TflName: 'None'
                    TflCheckSum: [4.2298e+09 3.9634e+09
                                  2.9213e+09 173860989]
      UtilMemSecPreStatement: ''
     UtilMemSecPostStatement: ''
           UtilMemSecComment: ''
        CodeCoverageChecksum: [3.6498e+09 78774415
                                  2.5508e+09 2.1183e+09]
TargetLargestAtomicInteger: 'Char'
  TargetLargestAtomicFloat: 'None'
  ProdLargestAtomicInteger: 'Char'
    ProdLargestAtomicFloat: 'Float'
               LongLongMode: 'on'
          ProdLongLongMode: 'off'
             HardwareBoard: 'None'
            toolchainOrTMF: 'Microsoft Visual C++ 2017 v15.0 |
                             nmake (64-bit Windows)'
```

Relate the Shared Utility Checksum to Configuration Parameters

Examine the `targetInfoStruct` hash table from the shared utility model. Some key-value pairs relate directly to a model property. Other pairs relate to groups of properties.

The following table describes the key-value pairs.

Key Names	Model Properties
`ShiftRightIntArith`	`TargetShiftRightIntArith`
`ProdShiftRightIntArith`	`ProdShiftRightIntArith`
`Endianess`	`TargetEndianess`
`ProdEndianess`	`ProdEndianess`
`wordlengths`	`TargetBitPerChar`, `TargetBitPerShort`, `TargetBitPerInt`, `TargetBitPerLong`, `TargetBitPerLongLong`, `TargetBitPerFloat`, `TargetBitPerDouble`,`TargetWordSize`, `TargetBitPerPointer`, `TargetBitPerSizeT`, `TargetBitPerPtrDiffT`
`Prodwordlengths`	`ProdBitPerChar`, `ProdBitPerShort`, `ProdBitPerInt`, `ProdBitPerLong`, `ProdBitPerLongLong`, `ProdBitPerFloat`, `ProdBitPerDouble`,`ProdWordSize`, `ProdBitPerPointer`, `ProdBitPerSizeT`, `ProdBitPerPtrDiffT`
`TargetWordSize`	`TargetWordSize`
`ProdWordSize`	`ProdWordSize`
`TargetHWDeviceType`	`TargetHWDeviceType`
`ProdHWDeviceType`	`ProdHWDeviceType`
`TargetIntDivRoundTo`	`TargetIntDivRoundTo`
`ProdIntDivRoundTo`	`ProdIntDivRoundTo`

Key Names	Model Properties
UseDivisionForNetSlopeComputation	UseDivisionForNetSlopeComputation
PurelyIntegerCode	PurelyIntegerCode
PortableWordSizes	PortableWordSizes
SupportNonInlinedSFcns	SupportNonInlinedSFcns
RTWReplacementTypes	EnableUserReplacementTypes, ReplacementTypes
MaxIdInt8	MaxIdInt8
MinIdInt8	MinIdInt8
MaxIdUint8	MaxIdUint8
MaxIdInt16	MaxIdInt16
MinIdInt16	MinIdInt16
MaxIdUint16	MaxIdUint16
MaxIdInt32	MaxIdInt32
MinIdInt32	MinIdInt32
MaxIdUint32	MaxIdUint32
MaxIdInt64	MaxIdInt64
MinIdInt64	MinIdInt64
MaxIdUint64	MaxIdUint64
BooleanTrueId	BooleanTrueId
BooleanFalseId	BooleanFalseId
TypeLimitIdReplacementHeaderFile	TypeLimitIdReplacementHeaderFile
SharedCodeRepository	Reserved (internal use only)
TargetLang	TargetLang
TargetLangStd	TargetLangStandard
InstructionSetExtions	InstructionSetExtensions
PreserveExternInFcnDecls	PreserveExternInFcnDecls
ImplementImageWithCVMat	ImplementImageWithCVMat
EnableSignedRightShifts	EnableSignedRightShifts
EnableSignedLeftShifts	EnableSignedLeftShifts
TflName	CodeReplacementLibrary
TflCheckSum	Reserved (internal use only)
UtilMemSecPreStatement	MemSecFuncSharedUtil, MemSecPackage
UtilMemSecPostStatement	MemSecFuncSharedUtil, MemSecPackage
UtilMemSecComment	MemSecFuncSharedUtil, MemSecPackage
CodeCoverageChecksum	Reserved (internal use only)
TargetLargestAtomicInteger	TargetLargestAtomicInteger
TargetLargestAtomicFloat	TargetLargestAtomicFloat

Key Names	Model Properties
`ProdLargestAtomicInteger`	`ProdLargestAtomicInteger`
`ProdLargestAtomicFloat`	`ProdLargestAtomicFloat`
`LongLongMode`	`TargetLongLongMode`
`ProdLongLongMode`	`ProdLongLongMode`
`HardwareBoard`	`HardwareBoard`
`toolchainOrTMF`	Name of: • Toolchain if build process uses either a toolchain (`Toolchain`) or a template makefile that is associated with a toolchain. • Template makefile if build process uses template makefile (`TemplateMakefile`).

See Also

More About

- “Generate Shared Utility Code for Fixed-Point Functions”
- “Generate Shared Utility Code for Custom Data Types”
- “Cross-Release Shared Utility Code Reuse”
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

Generate Shared Utility Code for Fixed-Point Functions

An important set of generated functions that the model build places in the shared utility folder are the fixed-point support functions. Based on model and block properties, there are many possible versions of fixed-point utilities functions that make it impractical to provide a complete set as static files. Generating only the required fixed-point utility functions during the code generation process is an efficient alternative.

The shared utility checksum mechanism makes sure that several critical properties are identical for models that use the shared utilities. For the fixed-point functions, there are additional properties that determine function behavior. The mechanism codes these properties into the functions and file names to maintain requirements. The additional properties include:

Category	Function/Property
Block properties	• Fixed-point operation that the block performs • Fixed-point data type and scaling (`Slope`, `Bias`) of function inputs and outputs • Overflow handling mode (`Saturation`, `Wrap`) • Rounding Mode (`Floor`, `Ceil`, `Zero`)
Model properties	`get_param(bdroot, 'NoFixptDivByZeroProtection')`

The property-based naming convention for the fixed-point utilities is as follows:

```
operation + [zero protection] + output data type + output bits +
[input1 data] + input1 bits + [input2 data type + input2 bits] +
[shift direction] + [saturate mode] + [round mode]
```

The file names shown are examples of generated fixed-point utility files. The function or macro names in the file are identical to the file name without the extension.

```
FIX2FIX_U12_U16.c
FIX2FIX_S9_S9_SR99.c
ACCUM_POS_S30_S30.h
MUL_S30_S30_S16.h
div_nzp_s16s32_floor.c
div_s32_sat_floor.c
```

For these examples, the table shows how the respective fields correspond.

The `ACCUM_POS` example uses the output variable as one of the input variables. So, the file and macro name only contain the output and second input.

The second `div` example has identical data type and bits for both inputs and the output. So, the file and function name only include the output.

Operation	FIX2FIX	FIX2FIX	ACCUM_POS	MUL	div	div
Zero protection	`NULL`	`NULL`	`NULL`	`NULL`	`_nzp`	`NULL`
Output data type	`_U`	`_S`	`_S`	`_S`	`_s`	`_s`
Output bits	`12`	`9`	`30`	`30`	`16`	`32`
Input data type	`_U`	`_S`	`_S`	`_S [and _S]`	`s`	`NULL`
Input bits	`16`	`9`	`30`	`30 [and 16]`	`32`	`NULL`

Operation	FIX2FIX	FIX2FIX	ACCUM_POS	MUL	div	div
Shift direction	NULL	SR99	NULL	NULL	NULL	NULL
Saturate mode	NULL	NULL	NULL	NULL	NULL	_sat
Round mode	NULL	NULL	NULL	NULL	_floor	_floor

See Also

More About

- “Generate Shared Utility Code”
- “Generate Shared Utility Code for Custom Data Types”
- “Cross-Release Shared Utility Code Reuse”

Generate Shared Utility Code for Custom Data Types

By default, if a model employs a custom data type (such as a `Simulink.AliasType` object or an enumeration class), the code generator places the corresponding type definition (`typedef`) in the *`model`*`_types.h` file. When you generate code from multiple models, each model duplicates the type definition. These duplicate definitions can prevent you from compiling the bodies of generated code together.

However, you can configure the code generator to place a single type definition in a header file in the `_sharedutils` folder. Then, when you generate code from a model, if the type definition already exists in the `_sharedutils` folder, the code generator does not duplicate the definition, but instead reuses it through inclusion (`#include`).

Through this mechanism, you can share:

- Simulink data type objects that you instantiate from the classes `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType`. For basic information about creating and using these objects, see “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2 and `Simulink.Bus`.
- Enumerations that you define, for example, by authoring an `enum` class in a script file or by using the function `Simulink.defineIntEnumType`. For basic information about defining enumerations in Simulink, see “Use Enumerated Data in Simulink Models”.

To share a custom data type across multiple models:

1. Define the data type. For example, create the `Simulink.AliasType` object.
2. Set data scope and header file properties to specific values that enable sharing.

   For a data type object, set the `DataScope` property to `'Exported'` and, optionally, specify the header file name through the `HeaderFile` property.

   For an enumeration that you define as an `enum` class in a script file, implement the `getDataScope` method (with return value `'Exported'`) and, optionally, implement the `getHeaderFile` method.

   For an enumeration that you define by using the `Simulink.defineIntEnumType` function, set the `'DataScope'` pair argument to `'Exported'` and, optionally, specify the `'HeaderFile'` pair argument
3. Use the data type in the models.
4. Before generating code from each model, set the model configuration parameter **Shared code placement** to `Shared location`.
5. Generate code from the models.

Note You can configure the definition of the custom data type to appear in a header file in the `_sharedutils` folder. The shared utility functions that the model build generates into the `_sharedutils` folder do not use the custom data type name. Only model code located in code folders for each model uses the custom data type name.

See Also

More About

- “Generate Shared Utility Code”
- “Generate Shared Utility Code for Fixed-Point Functions”
- “Cross-Release Shared Utility Code Reuse”
- “Control File Placement of Custom Data Types” on page 38-21

Shared Constant Parameters for Code Reuse

You can share the generated code for constant parameters across models if either of the following conditions applies:

- Constant parameters are shared in a model reference hierarchy.
- The model configuration parameter “Shared code placement” is set to `Shared location`.

If you do not want to generate shared constants, and model configuration parameter **Shared code placement** is set to `Shared location`, set parameter `GenerateSharedConstants` to `off`. For example, to turn off shared constants for the current model, in the Command Window, type the following.

```
set_param(gcs,'GenerateSharedConstants','off');
```

When the model configuration parameter `GenerateSharedConstants` is set to `off`, the code generator generates the constant parameters shared across the model reference hierarchy in a shared location, in the `slprj/`*`target`*`/_sharedutils` folder but outside the `const_params.c` file. The code generator defines the individual constant parameters in the reusable library subsystem files individually.

When the model configuration parameter `GenerateSharedConstants` is set to `on`, the code generator produces shared constant parameters individually and places them in the file `const_params.c`. The code generator places that file in the shared utilities folder `slprj/`*`target`*`/_sharedutils`.

For example, if a constant has multiple uses within a model reference hierarchy where the top model is named `topmod`, the code for the shared constant is as follows:

- In the shared utility folder, `slprj/grt/_sharedutils`, the constant parameters are defined in `const_params.c` and named `rtCP_pooled_` appended to a unique checksum:

  ```
  extern const real_T rtCP_pooled_lfcjjmohiecj[7];
  const real_T rtCP_pooled_lfcjjmohiecj[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

  extern const real_T rtCP_pooled_ppphohdbfcba[7];
  const real_T rtCP_pooled_ppphohdbfcba[7] = { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 };
  ```

- In *`top_model`*`_private.h` or in a referenced model, *`ref_model`*`_private.h`, for better readability, the constants are renamed as follows:

  ```
  extern const real_T rtCP_pooled_lfcjjmohiecj[7];
  extern const real_T rtCP_pooled_ppphohdbfcba[7];

  #define rtCP_Constant_Value   rtCP_pooled_lfcjjmohiecj  /* Expression: [1 2 3 4 5 6 7]
                                                           * Referenced by: '<Root>/Constant'*/
  #define rtCP_Gain_Gain        rtCP_pooled_ppphohdbfcba  /* Expression: [7 6 5 4 3 2 1]
                                                           * Referenced by: '<Root>/Gain' */
  ```

- In `topmod.c` or `refmod.c`, the call site might be:

  ```
  for (i = 0; i < 7; i++) {
     topmod_Y.Out1[i] = (topmod_U.In1 + rtCP_Constant_Value[i]) * rtCP_Gain_Gain[i];
  }
  ```

The code generator attempts to generate constants as individual constants to the `const_params.c` file in the shared utilities folder. If a model contains a parameter that the code generator cannot inline directly (for example, an array parameter), the code defines a structure type that represents the data. This structure uses the `const` storage type qualifier, so some build toolchains can optimize the assembly code further.

The declaration of the structure, `ConstParam_`*`model`*, is in *`model`*`.h`.

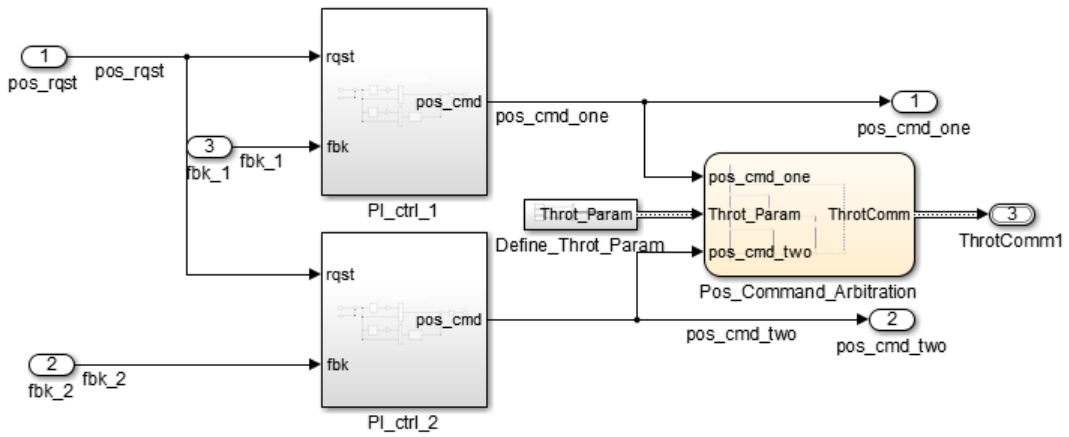
Suppress Shared Constants in the Generated Code

You can choose whether the code generator produces shared constants and shared functions. You might want to be able to keep the code and data separate between subsystems, or you might find that sharing constants results in a memory shortage during code generation.

You can change this setting programmatically by using the parameter `GenerateSharedConstants` with `set_param` and `get_param`.

In the following example, when `GenerateSharedConstants` is set to `on`, the code generator defines the constant values in the `_sharedutils` folder in the `const_params.c` file. When `GenerateSharedConstants` is set to `off`, the code generator defines the constant values in a nonshared area, in the `model_ert_rtw` file in the `model_data.c` file.

Open the model `rtwdemo_throttlecntrl`:



Verify that model configuration parameter “Shared code placement” is set to `Shared location`. If the parameter is set to `Auto`, the code generator ignores setting `GenerateSharedConstants`. If you try to set the parameter value, an error message appears. By default, value `GenerateSharedConstants` is selected.

In the Command Window, set `GenerateSharedConstants` to `on`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','on')
```

You see the shared constant definitions in the folder `slprj/grt/_sharedutils`, in the file `const_params.c`:

```
extern const real_T rtCP_pooled_H4eTKtECwveN[9];
const real_T rtCP_pooled_H4eTKtECwveN[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6,
  0.75, 1.0 } ;

extern const real_T rtCP_pooled_SghuHxKVKGHD[9];
const real_T rtCP_pooled_SghuHxKVKGHD[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05,
  0.25, 0.5, 1.0 } ;

extern const real_T rtCP_pooled_WqWb2t17NA2R[7];
const real_T rtCP_pooled_WqWb2t17NA2R[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25,
  1.0 } ;

extern const real_T rtCP_pooled_Ygnal0wM3c14[7];
const real_T rtCP_pooled_Ygnal0wM3c14[7] = { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0
} ;
```

In the Command Window, set `GenerateSharedConstants` to `off`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','off')
```

You can see the unshared constants in the folder`rtwdemo_throttlecntrl_grt_rtw`, in the file `rtwdemo_throttlecntrl_data.c`:

```
/* Constant parameters (auto storage) */
const ConstP_rtwdemo_throttlecntrl_T rtwdemo_throttlecntrl_ConstP = {
  /* Pooled Parameter (Expression: P_OutMap)
   * Referenced by:
   *   '<S2>/Proportional  Gain Shape'
   *   '<S3>/Proportional  Gain Shape'
   */
  { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0 },

  /* Pooled Parameter (Expression: P_InErrMap)
   * Referenced by:
   *   '<S2>/Proportional  Gain Shape'
   *   '<S3>/Proportional  Gain Shape'
   */
  { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 },

  /* Pooled Parameter (Expression: I_OutMap)
   * Referenced by:
   *   '<S2>/Integral  Gain Shape'
   *   '<S3>/Integral  Gain Shape'
   */
  { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 },

  /* Pooled Parameter (Expression: I_InErrMap)
   * Referenced by:
   *   '<S2>/Integral  Gain Shape'
   *   '<S3>/Integral  Gain Shape'
   */
  { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5, 1.0 }
};
```

Shared constant Parameters Limitations

The code generator does not produce shared constants for a model when:

- The model has a code replacement library (CRL) that is specified for data alignment.
- The **Memory Section** for constants is `MemVolatile` or `MemConstVolatile`.
- The parameter `GenerateSharedConstants` is set to `off`.

Individual constants are not shared, if:

- A constant is referenced by a non-inlined S-function.
- A constant has a user-defined type where **Data Scope** is not set to `Exported`.

See Also

More About

- “Generate Reentrant Code from Subsystems” on page 8-28
- “Generate Reusable Code from Library Subsystems Shared Across Models”

Determine Why Subsystem Code Is Not Reused

Due to the limitations described in “Generate Reentrant Code from Subsystems”, the code generator might not reuse generated code as you expect. To determine why code generated for a subsystem is not reused:

1. Review the Subsystems section of the code generation report.
2. Compare subsystem checksum data.

Review Subsystems Section of HTML Code Generation Report

If the code generator does not generate code for a subsystem as reusable code, and you configured the subsystem as reusable, examine the Subsystems section of the code generation report (see “Generate Code Generation Report”). The Subsystems section contains:

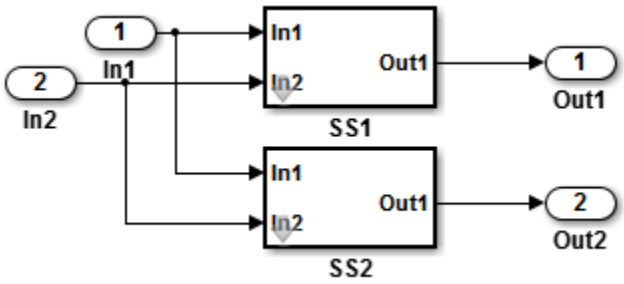
- A table that summarizes how nonvirtual subsystems were converted to generated code.
- Diagnostic information that describes why subsystems were not generated as reusable code.

The Subsystems section also maps noninlined subsystems in the model to functions or reused functions in the generated code. For an example, open and build the `rtwdemo_atomic` model.

Compare Subsystem Checksum Data

You can determine why subsystem code is not reused by comparing subsystem checksum data. The code generator determines whether subsystems are identical by comparing subsystem checksums, as noted in “Limitations”. For subsystem reuse across referenced models, this procedure might not flag every difference.

Consider the model, `rtwdemo_ssreuse`. `SS1` and `SS2` are instances of the same subsystem. In both instances the subsystem block parameter **Function packaging** is set to `Reusable function`.



Use the method `Simulink.SubSystem.getChecksum` to get the checksum for a subsystem. Review the results to determine why code is not reused.

1. Open the model `rtwdemo_ssreuse`. Save a copy of the model in a folder where you have write access.
2. Associate the subsystems `SS1` and `SS2` with `gcb`. For each of the subsystems , in the model window, select the subsystem. While the subsystem is selected, in the Command Window, enter:

   ```
   SS1 = gcb;
   ```

   ```
   SS2 = gcb;
   ```

3. Use the method `Simulink.SubSystem.getChecksum` to get the checksum for each subsystem. This method returns two output values: the checksum value and details on the input used to compute the checksum.

```
[chksum1, chksum1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
[chksum2, chksum2_details] = ...
Simulink.SubSystem.getChecksum(SS2);
```

4. Compare the two checksum values. The values should be equal based on the subsystem configurations.

```
isequal(chksum1, chksum2)
ans =
     1
```

5. To use `Simulink.SubSystem.getChecksum` to determine why the checksums of two subsystems differ, change the data type mode of the output port of SS1 so that it differs from that of SS2.

   a. Look under the mask of `SS1`. Right-click the subsystem. In the context menu, select **Mask > Look Under Mask**.

   b. In the block diagram of the subsystem, double-click the Lookup Table block to open the Subsystem Parameters dialog box.

   c. Click **Data Types**.

   d. Select block parameter **Saturate on integer overflow** and click **OK**.

6. Get the checksum for `SS1`. Compare the checksums for the two subsystems. This time, the checksums are not equal.

```
[chksum1, chksum1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
isequal(chksum1, chksum2)
ans =
     0
```

7. After you determine that the checksums are different, find out why. The Simulink engine uses information, such as signal data types, some block parameter values, and block connectivity information, to compute the checksums. To determine why checksums are different, compare the data that computes the checksum values. You can get this information from the second value returned by `Simulink.SubSystem.getChecksum`, which is a structure array with four fields.

   Look at the structure `chksum1_details`.

```
chksum1_details

chksum1_details =
          ContentsChecksum: [1x1 struct]
         InterfaceChecksum: [1x1 struct]
     ContentsChecksumItems: [287x1 struct]
    InterfaceChecksumItems: [53x1 struct]
```

   `ContentsChecksum` and `InterfaceChecksum` are component checksums of the subsystem checksum. The remaining two fields, `ContentsChecksumItems` and `InterfaceChecksumItems`, contain the checksum details.

8. Determine whether a difference exists in the subsystem contents, interface, or both. For example:

```
isequal(chksum1_details.ContentsChecksum.Value,...
        chksum2_details.ContentsChecksum.Value)
ans =
     0
isequal(chksum1_details.InterfaceChecksum.Value,...
        chksum2_details.InterfaceChecksum.Value)
ans =
     1
```

In this case, differences exist in the content.

9 Write a script like this script to find the differences.

```
idxForCDiffs=[];
for idx = 1:length(chksum1_details.ContentsChecksumItems)
    if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Identifier, ...
                chksum2_details.ContentsChecksumItems(idx).Identifier))
            disp(['Identifiers different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
    end
    if (ischar(chksum1_details.ContentsChecksumItems(idx).Value))
            if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Value, ...
                        chksum2_details.ContentsChecksumItems(idx).Value))
            disp(['Character vector values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
            end
    end
    if (isnumeric(chksum1_details.ContentsChecksumItems(idx).Value))
            if (chksum1_details.ContentsChecksumItems(idx).Value ~= ...
                        chksum2_details.ContentsChecksumItems(idx).Value)
            disp(['Numeric values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
            end
    end
end
```

10 Run the script. The example assumes that you named the script `check_details`.

```
check_details
Character vector values different for contents item 202
```

The results indicate that differences exist for index item 202 in the subsystem contents.

11 Use the returned index values to get the handle, identifier, and value details for each difference found.

```
chksum1_details.ContentsChecksumItems(202)

ans =

        Handle: 'rtwdemo_ssreuse/SS1/Lookup Table'
    Identifier: 'SaturateOnIntegerOverflow'
         Value: 'on'
```

The details identify the Lookup Table block parameter **Saturate on integer overflow** as the focus for debugging a subsystem reuse issue.

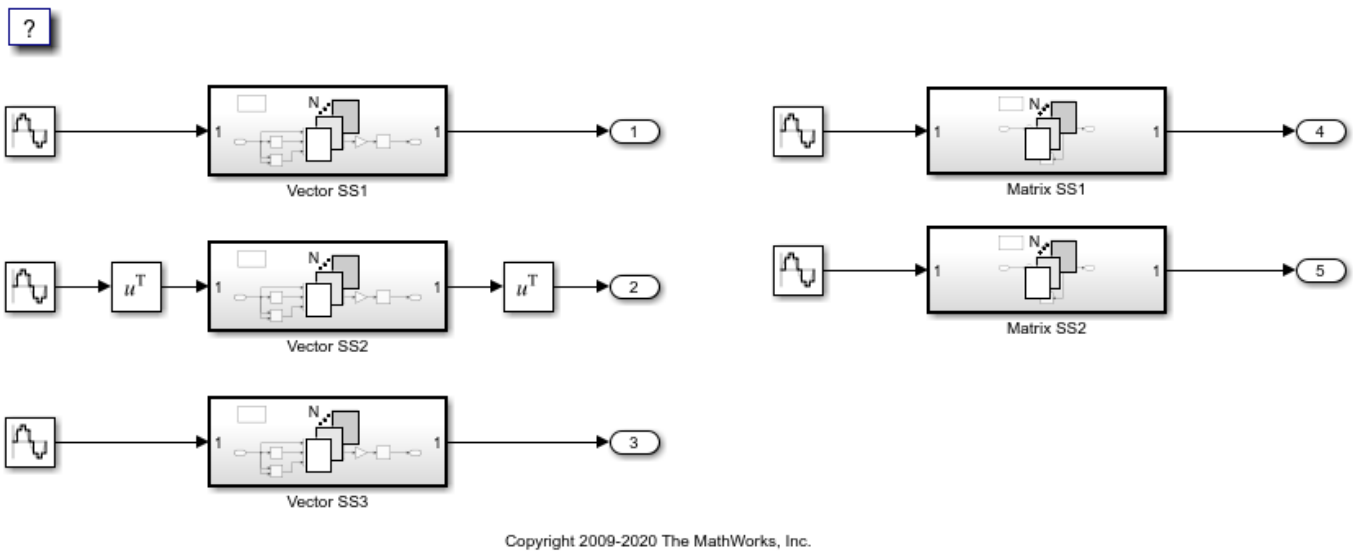
Generate Reusable Code from For Each Subsystems

In the model-based design, to convert inefficient repetitive algorithms into compact forms, use For Each subsystems. For Each subsystems repeat execution during a simulation time step on each element or subarray of an input signal or mask parameter array. In addition, to reduce model complexity, you can also generate reusable code for identical For Each subsystems by configuring them as reusable functions. You must configure mask and block parameters of the identical subsystems the same way. The code generator performs a checksum to determine whether subsystems are identical and the code is reusable. If the code is reusable, the code generator produces a single instance of reusable code which is shared by the identical subsystems.

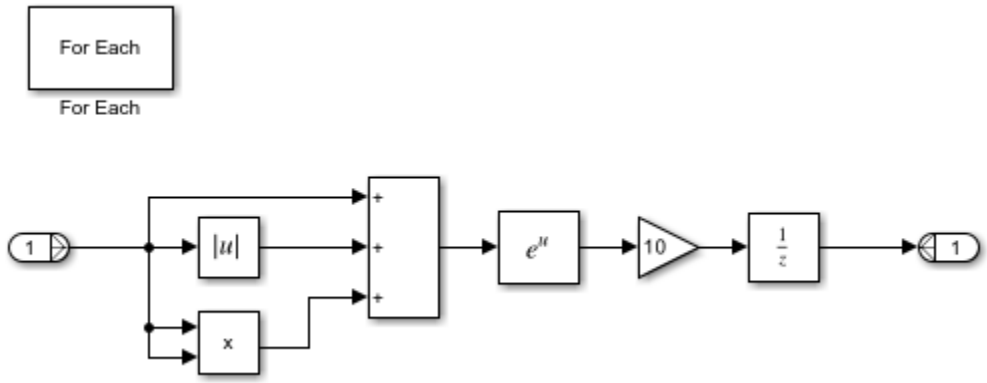
This example shows how you can generate reusable code for For Each subsystems that perform the same processing on their respective inputs.

Example Model

This example generates code from the `ForEachReuse` model provided in “Improved Code Reuse Using For-Each Subsystems”.



The model contains three For Each subsystems `Vector SS1`, `Vector SS2`, and `Vector SS3` that apply the same processing to each scalar element of the vector signal at their respective inputs. For example, the `Vector SS3` subsystem contains these blocks.



To generate a single shared function for the three subsystems, the input signal dimension of the subsystems must be the same. For `Vector SS1` and `Vector SS3`, set the partition dimension and width to `1`. For `Vector SS2` to also set the input signal dimension `1`, insert a Math Function block to transpose the 1-by-8 row vector into an 8-by-1 column vector. To convert the output of the subsystem back to a 1-by-8 row vector, use a second Math Function block.

The remaining For Each subsystem `Matrix SS1` and `Matrix SS2` process matrix signals using the same algorithm and have the same input signal dimension setting.

Generate and Inspect Code

1. To configure the subsystems as reusable functions, open the Block Parameters dialog box. On the **Code Generation** tab, set **Function packaging** to `Reusable function`.
2. To build the model and generate code, press **Ctrl+B**.

```
### Starting build procedure for: ForEachReuse
### Successful completion of build procedure for: ForEachReuse

Build Summary

Top model targets built:

Model         Action                        Rebuild Reason
========================================================================================
ForEachReuse  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.637s
```

The code generator generates a single output function `VectorProcessing` in `VectorProcessing.c`.

```
 * Output and update for iterator system:
 *    '<Root>/Vector SS1'
 *    '<Root>/Vector SS2'
 *    '<Root>/Vector SS3'
 */
void VectorProcessing(int32_T NumIters, const real_T rtu_In1[10], real_T
                      rty_Out1[10], DW_VectorProcessing_T localDW[10])
{
```

```
  /* local scratch DWork variables */
  int32_T ForEach_itr;
  real_T rtb_Out1_CoreSubsysCanOut;
```

The function has an input parameter `NumIters` that indicates the number of independent scalars that each For Each Subsystem block processes. This function is called three times in the `ForEachReuse.c` with the parameter `NumIters` set to `10`, `8`, and `7` respectively.

The code generator also generates a single output function for `Matrix SS1` and `Matrix SS2` subsystem in `MatrixProcessing.c` which is called twice in the `ForEachReuse.c`.

```
 * Output and update for iterator system:
 *    '<Root>/Matrix SS1'
 *    '<Root>/Matrix SS2'
 */
void MatrixProcessing(int32_T NumIters, const real_T rtu_In1[32], real_T
                      rty_Out1[32], DW_MatrixProcessing_T localDW[2])
{
  /* local scratch DWork variables */
  int32_T ForEach_itr;
  real_T rtb_MathFunction_h_0;
  int32_T i;
  int32_T rtb_MathFunction_h_tmp;
  int32_T tmp;
```

See Also

For Each Subsystem

Related Examples

- “Generate Reusable Code from Library Subsystems Shared Across Models”
- “Repeat an Algorithm Using a For-Each Subsystem”

Library-Based Code Generation

Library-Based Code Generation for Reusable Library Subsystems

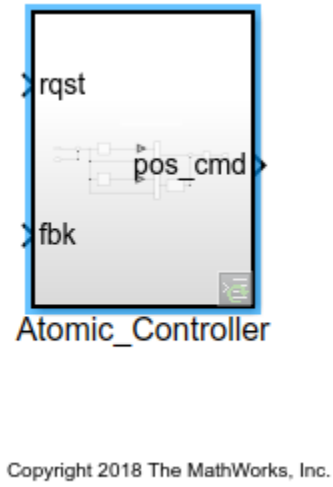
Library-based code generation provides a way of generating code for a set of reusable components that models can share. For each top-level reusable library subsystem, you specify a set of function interfaces that lock down the subsystem interface. A function interface consists of the subsystem input and output block parameter settings and the model configuration parameter settings.

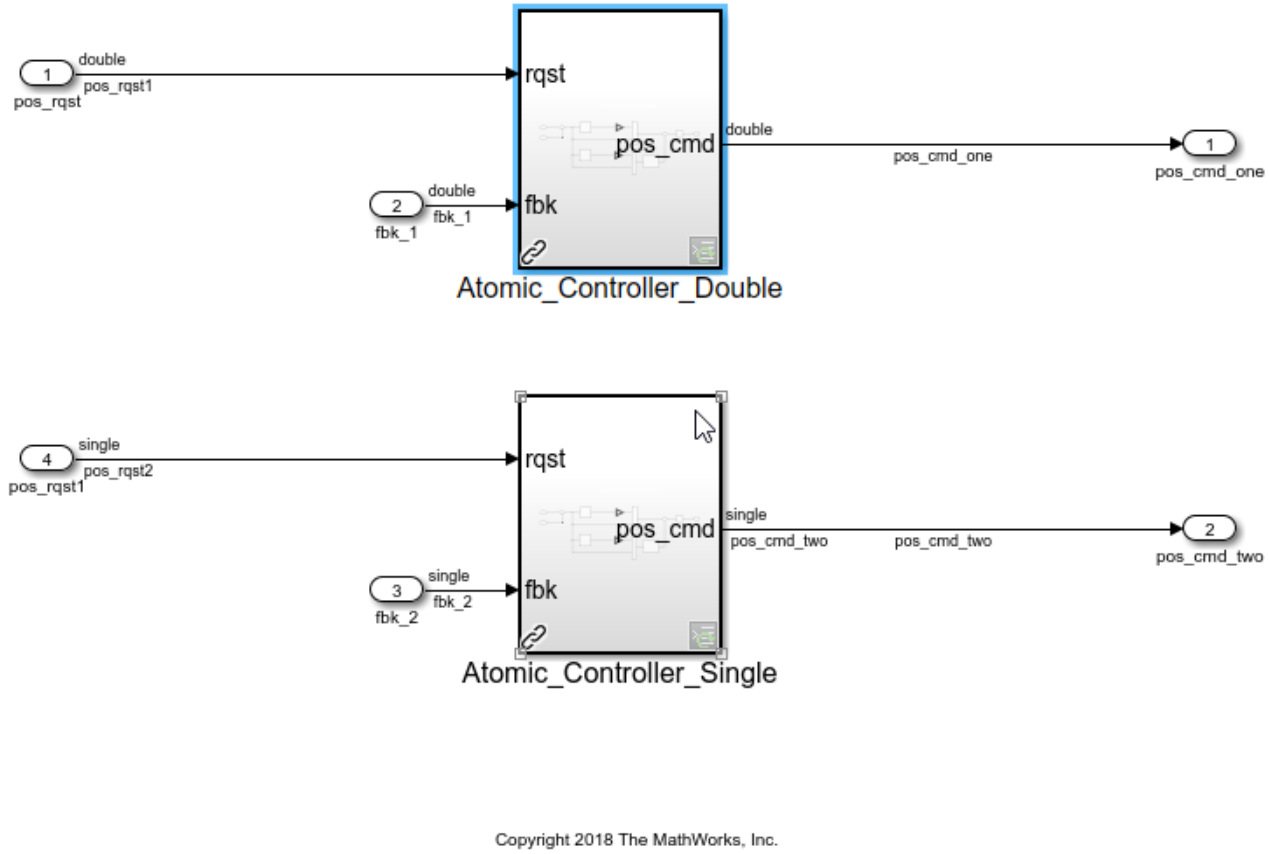
Function interfaces are independent models that you save with an accompanying library. Before generating code for a model containing instances of the reusable library subsystem, you generate code for the library. Library-based code generation makes the library the owner of the code. To make the individual models the owner of the code, you can generate code for reusable library subsystems to the shared utilities folder. For more information, see “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36.

Example Model and Library

To show how to use function interfaces and library-based code generation, this example uses the model `rtwdemo_libcodegen_mdl` and library `rtwdemo_libcodegen_lib`. The model contains two instances of the reusable library subsystem `Atomic_Controller`. For `Atomic_Controller_Single`, the input signal data type is `single`. For `Atomic_Controller_Double`, the input signal data type is `double`. To open the model and library files, at the MATLAB command prompt, enter:

```
rtwdemo_libcodegen_mdl
rtwdemo_libcodegen_lib
```





Configure Reusable Library Subsystems

To perform library-based code generation, you must configure the library subsystem as reusable.

1. In the Subsystem block parameters dialog box, select **Treat as an atomic unit**.
2. On the **Code Generation** tab:
   - Set **Function packaging** to `Reusable function`.
   - Set the **Function name options** and **File name options** parameters based on this table. This table summarizes how the code generator produces code. For the purpose of this table, assume that the Subsystem block is named `mySubsystem` and the `File name` value for `User specified` option is `myFunction`.

Function Name Options	File Name Options	Generated Function Name	Generated File Name
`Auto`	`Auto`	`mySubsystem_`*`checksum`*	`mySubsystem_`*`checksum`*`.c`, `mySubsystem_`*`checksum`*`.h`

Function Name Options	File Name Options	Generated Function Name	Generated File Name
	`Use subsystem name`	Code generation is not supported.	
	`Use function name`	`mySubsystem_`*`checksum`*	`mySubsystem_`*`checksum`*`.c`, `mySubsystem_`*`checksum`*`.h`
	`User specified`	`myFunction_Single`	`myFunction_Single.c`, `myFunction_Single.h`
`Use subsystem name`	`Auto`	Code generation is not supported.	
	`Use subsystem name`	Code generation is not supported.	
	`Use function name`	Code generation is not supported.	
	`User specified`	Code generation is not supported.	
`User specified`	`Auto`	`mySubsystem_Single`	`mySubsystem_Single.c`, `mySubsystem_Single.h`
	`Use subsystem name`	Code generation is not supported.	
	`Use function name`	`mySubsystem_Single`	`mySubsystem_Single.c`, `mySubsystem_Single.h`
	`User specified` (This specification is supported only if the user-specified function name and file name are same.)	`myFunction_Single`	`myFunction_Single.c`, `myFunction_Single.h`

Note When you set **Function name options** or **File name options** to `User specified`, the specification must contain `$R`. The code generator expands `$R` to the function interface name.

For more information about the Subsystem block parameters, see Subsystem.

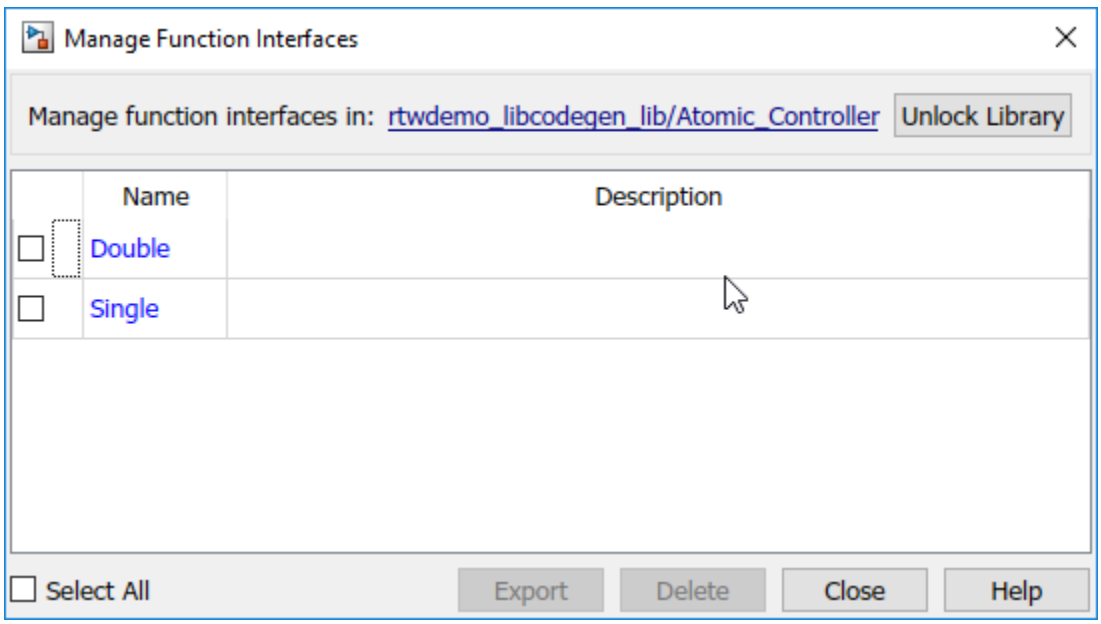
Specify Unique Function Interface Names

Each function interface corresponding to the same reusable library subsystem must have a unique name. To specify a unique name, follow these steps:

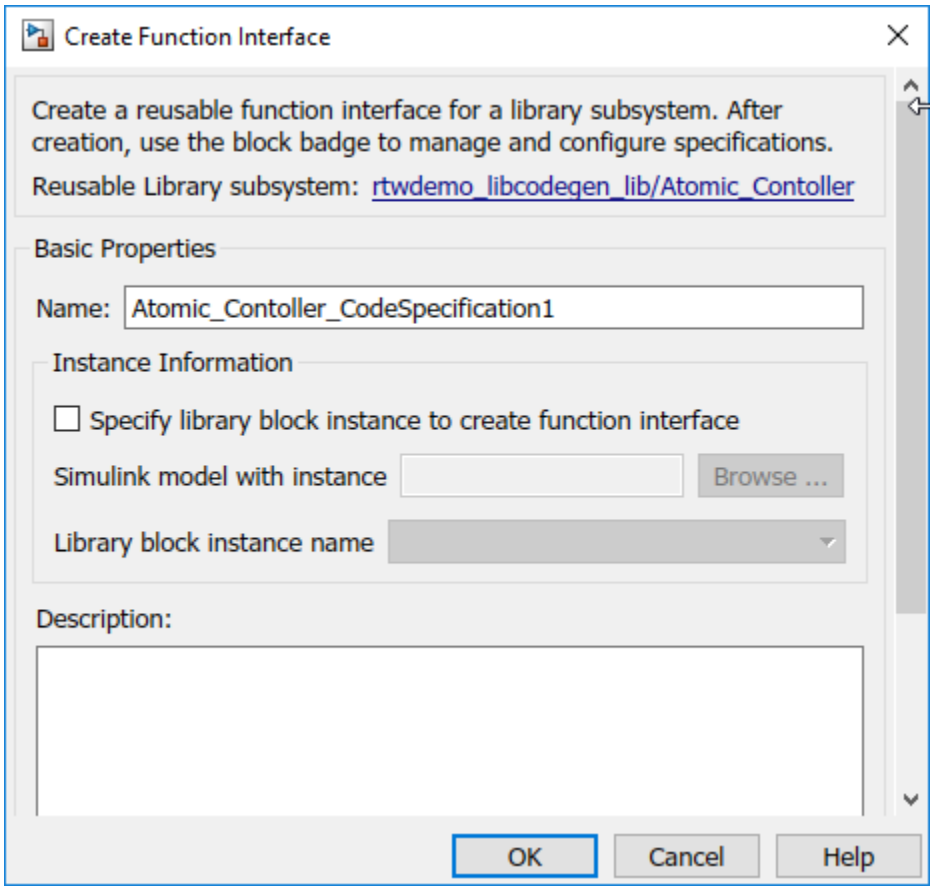
1. In the Subsystem block parameters dialog box, on the **Code Generation** tab, set the **Function name options** parameter to `User specified`.
2. For the **Function name** parameter, specify the `$R` and `$N` tokens. The `$R` token represents the function interface name. The `$N` token represents the subsystem name.
3. Set the **File name option** parameter to `Auto` or `Use function name`.
4. In the library, right-click the Subsystem and select **C/C++ Function Interfaces > Create Function Interface**. In the **Create Function Interface** dialog box, for the **Name** parameter, specify a name that describes the context.

Configure and Manage Function Interfaces

The model `rtwdemo_libcodegen_mdl` contains two instances of the reusable library subsystem `Atomic_Controller`. Each instance represents a function interface. In the library `rtwdemo_libcodegen_lib`, right-click the badge at the lower rightmost side of the reusable library subsystem `Atomic_Controller`. Select **Manage Function Interfaces**. The two function interfaces have the names `Single` and `Double` because `Atomic_Controller` takes `single` and `double` data types.



To create function interfaces, in a library, right-click a subsystem and select **C/C++ Function Interfaces > Create Function Interface**. Specify a function interface **Name**.



Then, to configure the function interfaces, choose one of the following methods.

Specify a Function Interface from Model

To create function interfaces from linked instances of the reusable library subsystem:

1. In the library, right-click the badge at the lower rightmost side of the reusable library subsystem and select **Create Function Interface**. In the dialog box, select the **Specify library block instance to create function interface** parameter.
2. For the **Simulink model with instance** parameter, select the model that contains the subsystem.
3. For the **Library block instance name** parameter, select the subsystem.
4. Click **OK** and close the **Create Function Interface** dialog box.
5. For each function interface, repeat the preceding steps.

Note You can create a function interface from within a model that contains an instance of a linked reusable library subsystem. You must be in the Code perspective. To open the Code perspective, from the **apps** menu, select **Embedded Coder**. If the subsystem has function interfaces, a badge appears at the lower-right corner of the subsystem. Right-click the badge and select **Create Function Interface**. If the subsystem does not have function interfaces, right-click the subsystem, and select **C/C++ Function Interfaces** > **Create Function Interface**.

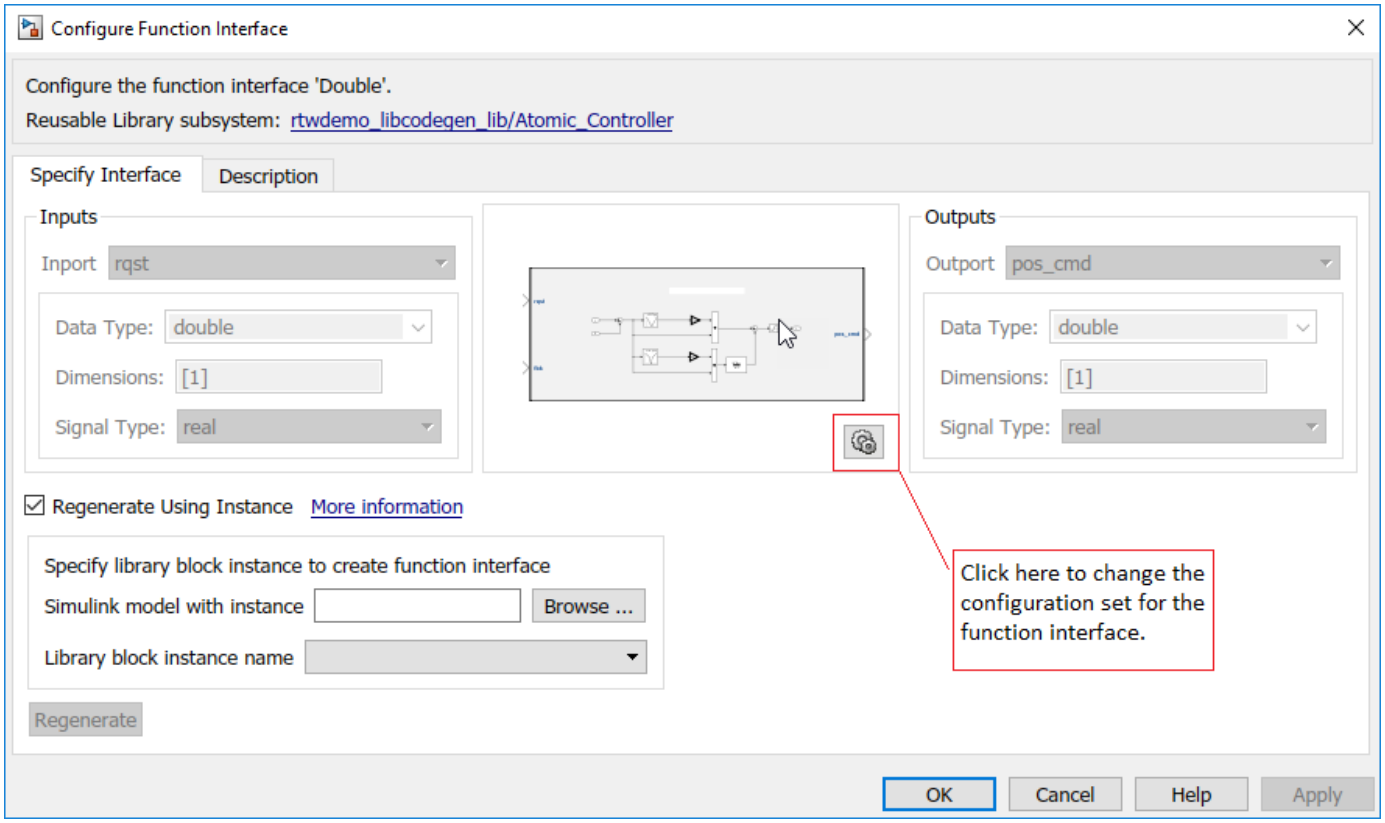
Export and Configure an Existing Function Interface

To export and configure the function interface as an independent model:

1. In the library, right-click the badge at the lower rightmost side of the reusable library subsystem and select **Manage Function Interfaces**.
2. Select the function interfaces that you want to modify.
3. Click **Export**.
4. In the **Save As** window, specify the current working folder. An exported function interface is a .slx file that has the function interface name plus the appendix `_export`.
5. Open the exported model. Make your changes to the subsystem input and output block parameter settings and model configuration parameter settings. Save the model.
6. In the library, right-click the reusable library subsystem and select **C/C++ Function Interfaces > Create Function Interface**. Specify a function interface **Name**.
7. For the **Simulink model with instance** parameter, select the exported model that is in the current working folder.
8. For the **Library block instance name** parameter, select the subsystem.
9. Click **OK** and close the **Create Function Interface** dialog box.

Configure Function Interfaces from Within a Library

1. In the library, right-click the badge at the lower rightmost side of the subsystem and select **Configure Function Interface**.
2. In the **Configure Function Interface** dialog box, for the subsystem inputs and outputs, specify values for the **Data Type**, **Dimensions**, and **Signal Type** parameters. To modify other subsystem input and output parameter settings, follow the export method in the preceding section.



3. To modify the model configuration parameters, click the gear button and make the changes. Click **Apply**. Close the Configuration Parameters dialog box.
4. To replace a function interface with an existing one from an instance model, select **Regenerate Using Instance**.
5. Specify values for the **Simulink model with instance** and **Library block instance name** parameters. Click **Regenerate**.
6. Click **Apply** and close the **Configure Function Interface** dialog box.

Build the Library

After you specify function interfaces for the subsystems in your library, generate code for the library. Before generating code for your model, you must generate code for the library. The code generator packages the library code as a separate C library. The generated code for the library is in a folder corresponding to your hardware settings (for example, IntelWin64). The library code folder has the same name as the library and must be at the same hierarchical level as the library.

To generate code, check that the library is unlocked. Open the **Embedded Coder** app and click **Build**.

When you generate code for the `rtwdemo_libcodegen_lib` library, the `rtwdemo_libcodegen_lib` folder contains these `.c` and `.h` files:

- `Atomic_Controller_Single.h`
- `Atomic_Controller_Single.c`

- `Atomic_Controller_Double.h`
- `Atomic_Controller_Double.c`

These function names are representative of the `$N$R` specification for the **Function name options** and **Function name** parameters on the Subsystem block parameters dialog box.

A `_shared` folder contains code for shared utilities (for example, fixed-point utilities and Lookup Table and MATLAB function blocks), supplementary files, and exported parameters and types.

When you generate code for a model that contains an instance of a reusable library subsystem that can use the pregenerated library code, the model links to the library code. The code generator uses a checksum to determine reusability. The generated code for the model must be in the same folder as the library. At the MATLAB command line, enter:

```
Simulink.fileGenControl('set', 'CodeGenFolderStructure',...
 Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

For more information, see `Simulink.fileGenControl`.

If the model is unable to use the library code, you can specify whether or not Embedded Coder produces a warning or an error during code generation. In the Configuration Parameters dialog box, set the **Behavior when pregenerated library subsystem code is missing** diagnostic parameter setting.

You can generate code for a library but not execute a makefile by entering these commands:

```
library='rtwdemo_libcodegen_lib'
set_param(library, 'GenCodeOnly', 'on')
slbuild(library)
```

Note You can generate code for a library consisting of reusable subsystems that contain S-functions. To avoid uncompilable code, in the TLC function corresponding to the S-function, avoid directing the code generator to interact with model files, such as *`model`*`.c`, *`model`*`.h`, and *`model_types`*`.h`.

Generate Code from Model Containing a Reusable Library Subsystem Instance

To generate code from a model that contains an instance of a reusable library subsystem for which you want to use the library code:

1. Set the model configuration parameter **Shared code placement** to `Shared location`.
2. Specify a setting for the “Behavior when pregenerated library subsystem code is missing” parameter or leave the default setting, which is warning.
3. In the Configuration Parameters dialog box, the parameter settings on the **Code Generation** panes must all be identical to each other. If the settings are different, you might get a warning, error, or neither depending on the setting of the **Behavior when pregenerated library subsystem code is missing** parameter.
4. If a reusable library subsystem uses a shared local data store and you configure default mapping for model data elements, leave the default storage class mapping for category **Shared local data stores** set to **Default**.

Here is the generated C code for `rtwdemo_libcodegen_mdl`.

```
/* Model step function */
void rtwdemo_libcodegen_mdl_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/Atomic_Controller_Double' */

  /* Inport: '<Root>/pos_rqst' incorporates:
   *  Inport: '<Root>/fbk_1'
   *  Outport: '<Root>/pos_cmd_one'
   */
  Atomic_Controller_Double(pos_rqst1, rtU.fbk_1, &rtY.pos_cmd_one,
    &rtDW.Atomic_Controller_Double);

  /* End of Outputs for SubSystem: '<Root>/Atomic_Controller_Double' */

  /* Outputs for Atomic SubSystem: '<Root>/Atomic_Controller_Single' */

  /* Inport: '<Root>/pos_rqst1' incorporates:
   *  Inport: '<Root>/fbk_2'
   *  Outport: '<Root>/pos_cmd_two'
   */
  Atomic_Controller_Single(pos_rqst2, rtU.fbk_2, &rtY.pos_cmd_two,
    &rtDW.Atomic_Controller_Single);

  /* End of Outputs for SubSystem: '<Root>/Atomic_Controller_Single' */
}
```

The code contains calls to the `Atomic_Controller_Single` and `Atomic_Controller_Double` functions. The generated code pulls the function definitions from the pregenerated library code.

Verify Code Generated from Reusable Library Subsystems

Verification Workflow

If you have Simulink Test™ software, you can verify code that you generate from reusable library subsystems. Use this workflow:

1. Create a test harness in a library for a unique subsystem and function interface pair.
2. With the SIL/PIL Manager:
   - **a** Run normal mode and software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the subsystem.
   - **b** Compare numerical results in the Simulation Data Inspector.
   - **c** View the Simulink Coverage™ analysis report.

For more information, see “Test Library Blocks” (Simulink Test).

Verification Workflow Limitations

The verification workflow does not support:

- Function-Call Subsystem blocks.
- Triggered Subsystem blocks or subsystems that use zero-crossing events.
- Function interfaces where the number of Inports/Outports does not match the number of Inports/Outports in the graphical interface.

- Parameters that use the `ExportToFile` storage class.
- Subsystems with Outport blocks that are fed by Mux or Demux blocks.
- Virtual buses that are passed as Inport blocks.
- Data Store Memory blocks that use the `ExportedGlobal` storage class.
- Blocks where data is initialized outside the subsystem code, for example, the Width block.
- Code coverage highlighting and annotation.
- Signal and state data logging.

For such cases, you can create a model that contains an instance of the reusable library subsystem and use the SIL/PIL Manager to run SIL or PIL simulations.

Limitations

Because the code generator uses a checksum to determine reusability, the same limitations that apply to generating code for models that share reusable library subsystems apply to library-based code generation. See “Limitations” on page 8-33. These limitations also apply:

- You cannot specify a function interface on a reusable library subsystem that is within another reusable library subsystem.
- Only ERT and ERT-derived system target files support library-based code generation.
- Each function interface that corresponds to the same reusable library subsystem must be unique.
- On Windows platform, avoid naming the library `lib.slx` as doing so may interfere with the static library compilation process.
- When you generate code from a library, the code generator only generates code for top-level reusable subsystems in the library. The top-level library subsystem can be instantiated at any level in a client model and the instance in the client model reuses library code if it finds any.
- A library subsystem cannot reuse code if it is inside an Enabled Subsystem block with the Enable block parameter, **States when enabling**, set to `reset`.

See Also

More About

- “Generate Reentrant Code from Subsystems” on page 8-28
- “Generate Reusable Code from Library Subsystems Shared Across Models” on page 8-36
- “Determine Why Subsystem Code Is Not Reused” on page 8-67

Configure Model Parameters for Simulink Coder

- “Configure Run-Time Environment Options” on page 10-2
- “Using the Target Framework” on page 10-11
- “Register New Hardware Devices” on page 10-13

Configure Run-Time Environment Options

When you use Simulink software to create and execute a model and use the code generator to produce C or C++ code, consider your configuration for up to three run-time environments:

- The MATLAB development computer run-time environment that runs MathWorks software during application development.
- The production hardware run-time environment in which you deploy an application when it is put into production.
- The test hardware run-time environment in which you test an application under development before deployment.

One run-time environment can serve in multiple capacities, but the run-time environments remain conceptually distinct. Often, the MATLAB development computer is the test hardware. Typically, the production hardware is different from, and less powerful than, the MATLAB development or the test hardware. Many types of production hardware can do little more than run a downloaded executable file.

Provide information about the production hardware board and the compiler that you use with it when:

- You use Simulink software to simulate a model for which you later generate code
- You use the code generator to produce code for deployment on *production* hardware

The software uses the board and compiler information to get bit-true agreement for the results of integer and fixed-point operations performed in simulation and in code generated for the production hardware. The code generator uses the information to create code that executes with maximum efficiency.

When you generate code for testing on *test* hardware, provide information about the test hardware board and the compiler that you use. The code generator uses this information to create code that provides bit-true agreement between results from:

- Integer and fixed-point operations performed in simulation
- Generated code run on the production hardware
- Generated code run on the test hardware

You can achieve bit-true agreement for results even if the production and test hardware are different. Where the C standard does not completely define behavior, the compilers for the two types of hardware can use different defaults.

Configure Production and Test Hardware

You can specify model simulation or code generation for a specific hardware board and its device type. For example, you can set the data size, byte ordering, and compiler behavior, such as integer rounding. You can configure:

- The production hardware and the compiler that you use with it. This information affects simulation and code generation. See “Example Production Hardware Setting That Affects Normal Mode Simulation” on page 10-9.

- The test hardware and the compiler that you use with it. This information affects only code generation.

Configure production hardware by selecting **Configuration Parameters > Hardware Implementation**. By default, the Hardware Implementation pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only. Unless you have installed hardware support packages, **Hardware board** lists values `None` or `Determine by Code Generation system target file`, and `Get Hardware Support Packages`. After installing a hardware support package, the list also includes the corresponding hardware board name. If you select a hardware board name, parameters for that board appear. To set device details, such as data size and byte ordering, click **Device details**.

Configure test hardware on the **Configuration Parameters > Hardware Implementation > Advanced parameters** pane. To enable parameters for configuring test hardware details, disable the **Configuration Parameters > Hardware Implementation > Advanced parameters > Test hardware is the same as production hardware** parameter. Code generated for test hardware executes in the environment specified by the test hardware parameters. The code behaves as if it were executing in the environment specified for the production hardware. For more information, see “Test Hardware Considerations” on page 10-8.

Default values and properties appear as initial values in the **Hardware Implementation** pane when:

- You specify a **System target file** in the **Code Generation** pane.
- The system target file specifies a default microprocessor and its hardware properties.

You cannot change parameters that have only one possible value. Parameters that have more than one possible value provide a list of valid values. If you specify hardware properties manually in **Hardware Implementation** pane, verify that these values are consistent with the system target file. Otherwise, the generated code can fail to compile or execute, or can execute but produce incorrect results.

Hardware implementation parameters describe hardware and compiler properties to MATLAB software. The code generator uses the information to produce code for the run-time environment that runs as efficiently as possible. The generated code gives bit-true agreement for the results of integer and fixed-point operations in simulation, production code, and test code.

For details about specific parameters, see “Hardware Implementation Pane”. To see an example of **Hardware Implementation** pane capabilities, see the `rtwdemo_targetsettings` example model. For details related to configuring a hardware implementation, see:

- “Specify Hardware Board” on page 10-4
- “Specify Device Vendor” on page 10-4
- “Specify the Device Type” on page 10-4
- “Set Bit Lengths for Device Data Types” on page 10-5
- “Set Byte Ordering for Device” on page 10-6
- “Set Quotient Rounding Behavior for Signed Integer Division” on page 10-6
- “Set Arithmetic Right Shift Behavior for Signed Integers” on page 10-7
- “Update Release 14 Hardware Configuration” on page 10-7

Specify Hardware Board

Specify the hardware board that runs the code generated from your model. Select a value for **Configuration Parameters** > **Hardware Implementation** > **Hardware board**.

The Hardware Implementation pane identifies the system target file selected on **Configuration Parameters** > **Code Generation**.

To configure test hardware, use the **Configuration Parameters** > **Hardware Implementation** > **Advanced parameters** pane.

To enable parameters for configuring test hardware details, set `ProdEqTarget` to `off`.

Ways to Specify the Hardware Board

If	Select
The menu *includes* the name of the hardware board that you want to use.	The name of that hardware board. If you select a hardware board name, parameters for that board appear.
The menu *does not include* the name of the hardware board that you want to use.	`Get Hardware Support Packages`. That value opens the Support Package Installer. Install the support package that you want. After you install the support package, the menu includes relevant hardware board names.
The model configuration *uses* system target file `ert.tlc`, `realtime.tlc`, or `autosar.tlc`.	`None`. No hardware board is specified for the hardware implementation.
The model configuration *does not use* system target file `ert.tlc`, `realtime.tlc`, or `autosar.tlc`.	`Determine by Code Generation system target file`. The code generator uses the specified system target file to determine the hardware implementation.

Specify Device Vendor

To specify the vendor of the microprocessor of the hardware device, use the **Device vendor** parameter. Your selection determines the available microprocessors in the **Device type** menu. If the vendor name does not appear, select `Custom Processor`. Then, use the **Device type** parameter to specify the microprocessor.

- For complete lists of **Device vendor** and **Device type** values, see “Device vendor” and “Device type”.
- To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, see “Register New Hardware Devices” on page 10-13.

Specify the Device Type

To specify the microprocessor name from the supported devices listed for your **Device vendor** selection, use the **Device type** parameter. If the microprocessor does not appear in the menu, change **Device vendor** to `Custom Processor`. Then, specify device details for your custom device.

If you select a device type for which the system target file specifies default hardware properties, the properties appear as initial values. You cannot change the value of parameters with only one possible selection. Parameters that have more than one possible value provide a menu. Select values for your hardware.

Set Bit Lengths for Device Data Types

The **Number of bits** parameters describe the **native word size** of the microprocessor and the bit lengths of **char**, **short**, **int**, and **long** data. For code generation to succeed:

- The bit lengths must be such that **char <= short <= int <= long**.
- Bit lengths must be multiples of 8, with a maximum of 32.
- The bit length for **long** data must not be less than 32.

The `rtwtypes.h` file defines integer type names. The values that you provide must be consistent with the word sizes as defined in the compiler `limits.h` header file. The code generator maps its integer type names to the corresponding Simulink integer type names.

If no ANSI® C type with a matching word size is available, but a larger ANSI C type is available, the code generator uses the larger type for `int8_T`, `uint8_T`, `int16_T`, `uint16_T`, `int32_T`, and `uint32_T`. When the code generator uses a larger type, the resulting logged values (for example, MAT-file logging) can have different data types than logged values for simulation.

An application can use an integer data of length from 1 (unsigned) or 2 (signed) bits up 32 bits. If the integer length matches the length of an available type, the code generator uses that type. If a matching type is not available, the code generator uses the smallest available type that can hold the data, generating code that does not use unnecessary higher-order bits. For example, on hardware that supports 8-bit, 16-bit, and 32-bit integers, for a signal specified as 24 bits, the code generator implements the data as an `int32_T` or `uint32_T`.

Code that uses emulated integer data is not maximally efficient. This code can be useful during application development for emulating integer lengths that are available only on production hardware. Emulation does not affect the results of execution.

During code generation, the software checks the compatibility of model data types with the data types that you specify for production hardware.

- If none of the lengths that you specify for production hardware integers is 32 bits, the software generates an error.
- If the lengths of data types that the model uses are smaller than the available production hardware integer lengths, the software generates a warning.

Mapping of Integer Types from Code Generator to Simulink

Code Generator Integer Type	Simulink Integer Type
`boolean_T`	`boolean`
`int8_T`	`int8`
`uint8_T`	`uint8`
`int16_T`	`int16`
`uint16_T`	`uint16`
`int32_T`	`int32`
`uint32_T`	`uint32`

Set Byte Ordering for Device

The **Byte ordering** parameter specifies whether the hardware uses `Big Endian` (most significant byte first) or `Little Endian` (least significant byte first) byte ordering. If left as `Unspecified`, the code generator produces code that determines the endianness of the hardware. This setting is the least efficient.

Set Quotient Rounding Behavior for Signed Integer Division

ANSI C does not completely define the quotient rounding technique for compilers to use when dividing one signed integer by another. So, the behavior is implementation-dependent. If both integers are positive, or both are negative, the quotient must round down. If either integer is positive and the other is negative, the quotient can round up or down.

The **Signed integer division rounds to** parameter instructs the code generator about how the compiler rounds the result of signed integer division. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize code generated for signed integer division. The parameter values are:

- `Zero` — If the quotient is between two integers, the compiler chooses the integer that is closer to zero as the result.
- `Floor` — If the quotient is between two integers, the compiler chooses the integer that is closer to negative infinity.
- `Undefined` — If `Zero` or `Floor` do not describe the compiler behavior or if that behavior is unknown, choose this value.

Avoid selecting `Undefined`. When the code generator does not know the signed integer division rounding behavior of the compiler, the model build generates extra code.

The compiler quotient rounding behavior varies according to these values.

You can obtain the compiler implementation for signed integer division rounding from the compiler documentation. If documentation is not available, you can determine this behavior by experiment.

Example Quotient Rounding for Zero, Floor, and Undefined

N	D	Ideal N/D	Zero	Floor	Undefined
33	4	8.25	8	8	8
-33	4	-8.25	-8	-9	-8 or -9
33	-4	-8.25	-8	-9	-8 or -9
-33	-4	8.25	8	8	8 or 9

Set Arithmetic Right Shift Behavior for Signed Integers

ANSI C does not define the behavior of right shifts on negative integers for compilers. So, the behavior is implementation-dependent. The **Shift right on a signed integer as arithmetic shift** option instructs the code generator about how the compiler implements right shifts on negative integers. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize the code generated for arithmetic right shifts.

If the C compiler implements a signed integer right shift as an arithmetic right shift, select the option. Otherwise, clear the option. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in two's-complement notation. The option is selected by default. If your compiler handles right shifts as arithmetic shifts, this setting is preferred.

- When you select the option, the code generator produces efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is cleared, the code generator produces fully portable but less efficient code to implement right arithmetic shifts.

You can obtain the compiler implementation for arithmetic right shifts from the compiler documentation. If documentation is not available, you can determine this behavior by experiment.

Update Release 14 Hardware Configuration

If your model was created before Release 14 and you have not updated the model, the **Configure current execution hardware device** parameter (`TargetUnknown`) value is `'on'` by default.

To update your model, clear the box for **Configuration Parameters** > **Hardware Implementation** > **Advanced parameters** > **Test hardware** > **Configure test hardware**.Or in the Command Window, type:

```
cs = getActiveConfigSet('your_model_name');
set_param(cs,'TargetUnknown','off');
```

This update to your model:

- Enables the **Test Hardware is the same as production hardware** parameter (`ProdEqTarget`), setting the parameter to `'on'`.
- Copies the **Production device vendor and type** parameter (`ProdHWDeviceType`) value to the **Test device vendor and type** parameter (`TargetHWDeviceType`).

To complete the update:

1. Clear the box for **Configuration Parameters > Hardware Implementation > Advanced parameters > Test hardware > Test Hardware is the same as production hardware**. Apply this step only if your production and test hardware are different.
2. Set the parameters in **Configuration Parameters > Hardware implementation > Advanced parameters** to match your production and test systems.
3. Save the model.

Production Hardware Considerations

When you configure production hardware, consider these points:

- Production hardware can have word sizes and other hardware characteristics that differ from the MATLAB development computer. You can prototype code on hardware that is different from the production hardware or the MATLAB development computer. When producing code, the code generator accounts for these differences.
- The Simulink product uses some of the information in the production hardware configuration. That information enables simulations without code generation to give the same results as executing generated code. For example, the results can detect error conditions that arise on the production hardware, such as hardware overflow.
- The code generator produces code that provides bit-true agreement with Simulink results for integer and fixed-point operations. Generated code that emulates unavailable data lengths runs less efficiently than without emulation. The emulation does not affect bit-true agreement with Simulink for integer and fixed-point results.
- If you change run-time environments during application development, before generating or regenerating code, reconfigure the hardware implementation parameters for the new run-time environment. When code executes on hardware for which it was not generated, bit-true agreement is not always achieved for results of integer and fixed-point operations in simulation, production code, and test code.
- To compile code generated from the model, use the **Integer rounding mode** parameter on model blocks to simulate the rounding behavior of the C compiler that you intend. This setting appears on the **Signal Attributes** pane in the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product, Matrix Multiply and n-D Lookup Table blocks.
- For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see “Precision” (Fixed-Point Designer).
- When models contain Model blocks, configure models that they reference to use identical hardware settings.

Test Hardware Considerations

By default, the test hardware configuration is the same as the configuration for the production hardware. You can use the generated code for testing in an environment that is identical to the production environment.

If the test and production environments differ, you can generate code that runs on test hardware as if it were running on production hardware:

1. To enable test hardware parameters, clear the box for **Configuration Parameters** > **Hardware Implementation** > **Advanced parameters** > **Test hardware** > **Test hardware is the same as production hardware**. Or, in the Command Window type:

   ```
   cs = getActiveConfigSet('your_model_name');
   set_param(cs,'ProdEqTarget','off');
   ```

2. Specify device type details through the test hardware (`Target*`) parameters.

If you select a system target file that specifies a default microprocessor and its hardware properties, these default values and properties appear as initial values.

Parameters with only one possible value cannot be changed. If you modify hardware properties, check that their values are consistent with the system target file. Otherwise, the generated code can fail to compile or execute, or can execute but produce incorrect results.

Example Production Hardware Setting That Affects Normal Mode Simulation

Changing some production hardware settings, for example, `ProdLongLongMode` and `ProdIntDivRoundTo`, can affect normal mode simulation results. The following example simulates an adder with four inputs. In the first simulation, `ProdLongLongMode` is disabled. In the second simulation, `ProdLongLongMode` is enabled. In the plot of simulation outputs, you observe small differences between output values in the time step range 125–175.

```
model = 'hwSettingEffect';
new_system(model)
open_system(model)

% Create adder
pos = [140   140   200   340];
add_block('simulink/Math Operations/Add', ...
          [model '/sum_int32'], ...
          'Inputs','++++', ...
          'SaturateOnIntegerOverflow', ...
          'on', ...
          'Position', ...
          pos)

pos = [75   155   105   175];
add_block('built-in/Inport',[model '/In1'],'Position',pos)
set_param([model '/In1'], 'OutDataTypeStr', ...
          'int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In1/1','sum_int32/1')

pos = [75   205   105   225];
add_block('built-in/Inport',[model '/In2'],'Position',pos)
set_param([model '/In2'], 'OutDataTypeStr', ...
           'int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In2/1','sum_int32/2')

pos = [75   255   105   275];
add_block('built-in/Inport',[model '/In3'],'Position',pos)
set_param([model '/In3'], 'OutDataTypeStr', ...
          'int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In3/1','sum_int32/3')

pos = [75   305   105   325];
add_block('built-in/Inport',[model '/In4'],'Position',pos)
set_param([model '/In4'], 'OutDataTypeStr', ...
           'int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In4/1','sum_int32/4')

pos = [275   230   305   250];
add_block('built-in/Outport',[model '/Out1'],'Position',pos)
```

```
add_line(model, 'sum_int32/1','Out1/1')

% Specify input data
t = 0:200;
peakValue = 1.5e9;
in1 = peakValue * sin(t*2*pi/100);
in2 = peakValue * cos(t*2*pi/70);
in3 = -peakValue * sin(t*2*pi/40);
in4 = -peakValue * cos(t*2*pi/30);
set = Simulink.SimulationData.Dataset;
set = set.addElement(1, timeseries(int32(in1),t,'Name','sig1'));
set = set.addElement(2, timeseries(int32(in2),t,'Name','sig2'));
set = set.addElement(3, timeseries(int32(in3),t,'Name','sig3'));
set = set.addElement(4, timeseries(int32(in4),t,'Name','sig4'));

set_param(model, 'LoadExternalInput', 'on');
set_param(model, 'ExternalInput', 'set');

set_param(model, 'StopTime', '50');

% Disable production hardware setting and run first simulation
set_param(model, 'ProdLongLongMode', 'off');
[~, ~, y1] = sim(model, 200);

% Enable production hardware setting and run second simulation
set_param(model, 'ProdLongLongMode', 'on');
[~, ~, y2] = sim(model, 200);

plot([y1 y2]);
figure(gcf);
```

The difference in behavior is due to the accumulator data type in the Sum block. The **Accumulator data type** block parameter is set to `Inherit: Inherit via internal rule`. For this example, the resulting accumulator data type is 64 bits wide if the use of the C `long long` data type is enabled. Otherwise, it is 32 bits wide. Depending on the input values for the sum block, the 32-bit accumulator can saturate when the 64-bit accumulator does not. Therefore, normal mode behavior can depend on the `ProdLongLongMode` setting. In both cases, the normal mode behavior and production hardware behavior matches bitwise.

See Also

More About

- “Hardware Implementation Pane”
- “Device vendor”
- “Device type”
- “Precision” (Fixed-Point Designer)

Using the Target Framework

The Target Framework enables you to define target hardware characteristics and integrate tools with MATLAB and Simulink. Using classes from the `target` package, capture target hardware or software platform information for MathWorks software. Perform tasks to optimize, prototype, verify, and deploy design components.

Target Framework Overview

Using target objects, you can capture information for MathWorks software. For example, to register a new hardware device for code generation, use this workflow:

1. Create a `target.Processor` object for the new hardware device.
2. Create a `target.LanguageImplementation` object for language implementation details.
3. Specify language implementation details.
4. Associate the language implementation with the hardware device.
5. Add the `target.Processor` object to an internal database.
6. On the **Hardware Implementation** pane, use the **Device vendor** and **Device type** settings to specify the hardware device as the target device for code generation.

For more information, see “Specify Hardware Implementation for New Device” on page 10-13.

To create and manage the objects, use these functions:

- `target.create` –– Create an object that captures component information for MathWorks software, for example, target processor and compiler properties.
- `target.add` –– Add the object to an internal database.
- `target.update` –– Update the target object in the internal database.
- `target.get` –– Retrieve the object from the internal database.
- `target.export` –– Export the object data for sharing.
- `target.upgrade` –– Upgrade existing definitions of hardware devices.
- `target.remove` –– Remove the object from the internal database.
- `target.clear` –– Clear all target objects from internal database.

Target Framework Workflows

This table lists tasks that you can perform by using `target` classes and functions.

Task	See
Register new hardware devices and hardware boards for **Hardware Implementation** settings.	“Register New Hardware Devices” on page 10-13
Build a target application using a CMake-based toolchain.	“Create Custom CMake Toolchain Definition” on page 59-17
Deploy, download, and run an application on target hardware.	“Set Up PIL Connectivity by Using target Package” on page 88-48

Task	See
Tune parameters and monitor target application signals during an external mode simulation.	“Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35 “External Mode Simulation by Using XCP Communication” on page 60-8 “Customize XCP Server Software” on page 60-45
Test numerical equivalence between your MATLAB code or Simulink model and the generated code by running a processor-in-the-loop (PIL) simulation.	“Set Up PIL Connectivity by Using target Package” on page 88-48 “Configure and Run PIL Simulation” on page 88-24
Perform code execution profiling through a PIL simulation.	“Set Up PIL Connectivity by Using target Package” on page 88-48 “Create Execution-Time Profile for Generated Code” on page 81-2

See Also

`target`

Related Examples

- “Custom Targets” on page 94-2
- “Register New Hardware Devices” on page 10-13
- “Create Custom CMake Toolchain Definition” on page 59-17
- “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35
- “Set Up PIL Connectivity by Using target Package” on page 88-48

Register New Hardware Devices

On the **Hardware Implementation** pane, you can specify parameters that describe target hardware and compiler properties for MATLAB software, which enables you to:

- Observe target hardware during model simulations.
- Generate optimized code for production or test hardware.
- Directly test or deploy generated code on target hardware.

The **Hardware Implementation** pane supports a range of target hardware. To extend the range, register new hardware devices by using the `target.Processor` and `target.LanguageImplementation` classes.

Specify Hardware Implementation for New Device

To register a new hardware device:

1. Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');
```

2. Create a `target.LanguageImplementation` object for language implementation details.

```
myLanguageImplementation = target.create('LanguageImplementation', ...
                                         'Name', 'MyProcessorImplementation');
```

3. Specify language implementation details.

```
myLanguageImplementation.Endianess = target.Endianess.Little;

myLanguageImplementation.AtomicIntegerSize = 64;
myLanguageImplementation.AtomicFloatSize = 64;
myLanguageImplementation.WordSize = 64;

myLanguageImplementation.DataTypes.Char.Size = 8;
myLanguageImplementation.DataTypes.Short.Size = 16;
myLanguageImplementation.DataTypes.Int.Size = 32;
myLanguageImplementation.DataTypes.Long.Size = 64;
myLanguageImplementation.DataTypes.LongLong.IsSupported = true;
myLanguageImplementation.DataTypes.LongLong.Size = 64;
myLanguageImplementation.DataTypes.Float.Size = 32;
myLanguageImplementation.DataTypes.Double.Size = 64;

myLanguageImplementation.DataTypes.Pointer.Size = 32;

myLanguageImplementation.DataTypes.SizeT.Size = 64;
myLanguageImplementation.DataTypes.PtrDiffT.Size = 64;
```

4. Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myLanguageImplementation;
```

5. Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

On the **Hardware Implementation** pane, you can now set **Device vendor** and **Device type** to `MyManufacturer` and `MyProcessor` respectively.

Specify Hardware Implementation That Persists Over MATLAB Sessions

By default, when you add the target object to the internal database, the target data is available only for the current MATLAB session. You can specify target data persistence over MATLAB sessions.

1. Create a `target.Processor` object for a new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');

existingImplementation = target.get('LanguageImplementation', ...
                                    'ARM Compatible-ARM Cortex');
myProc.LanguageImplementations = existingImplementation;
```

2. Add the `target.Processor` object to an internal database and specify persistence of target data over MATLAB sessions.

```
objectsAdded = target.add(myProc, 'UserInstall', true);
```

   If you subsequently modify the object in the MATLAB workspace and then want to update the object in the internal database, you can use the `target.update` function.

3. You can remove the object from the internal database.

```
target.remove(objectsAdded);
```

   To remove multiple persistent objects from the internal database, use the `target.clear` function.

Create Hardware Implementation by Modifying Existing Implementation

If an existing hardware implementation contains most of the values that you want in a new hardware implementation, you can quickly create the new implementation by creating and modifying a copy of the existing implementation.

1. Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                      'Name', 'MyProcessor', ...
                      'Manufacturer', 'MyManufacturer');
```

2. Create a `target.LanguageImplementation` object that copies an existing language implementation.

```
myCopiedImplementation = target.create('LanguageImplementation', ...
                                       'Name', 'MyCopiedImplementation', ...
                                       'Copy', 'Atmel-AVR');
```

3. Specify the required language implementation details. For example, byte ordering.

```
myCopiedImplementation.Endianess = target.Endianess.Big;
```

4. Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myCopiedImplementation;
```

5. Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

Create Hardware Implementation by Reusing Existing Implementation

If your hardware device requires the same hardware implementation as an existing implementation, you can reuse the existing implementation.

1. Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');
```

2. Retrieve the existing implementation by using the identifier for the device vendor and type, for example, `'ARM Compatible-ARM Cortex'`.

```
existingImplementation = target.get('LanguageImplementation', ...
                                    'ARM Compatible-ARM Cortex');
```

3. Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = existingImplementation;
```

4. Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

Validate Hardware Device Data

To validate the data integrity of target objects, use the `IsValid` property or `validate` method of the `target.Object` base class.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create('Processor');
myProcessor.LanguageImplementations = target.get('LanguageImplementation', ...
                                                 'ARM Compatible-ARM Cortex');
```

To validate the created object, run `myProcessor.IsValid` or `myProcessor.validate()`.

```
myProcessor.IsValid
```

```
ans =
  logical
  0
```

```
myProcessor.validate()
```

```
Error using target.Processor/validate
Target data validation failed.
* Undefined property "Name" in "Processor" object.
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- `Name` — Processor name
- `Id` — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = 'MyProcessor';
```

Check the validity of `myProcessor`.

```
myProcessor.IsValid
```

```
ans =
  logical
  1
```

The validity of the object is established.

Note When you use the `target.add` function to register a target object, the software also checks the validity of the object.

Export Hardware Device Data

You can share previously created hardware device data across computers and users.

For this example, specify a hardware device and add it to an internal database.

```
myProc = target.create('Processor', ...
                    'Name', 'MyProcessor', ...
                    'Manufacturer', 'MyManufacturer');
existingImplementation = target.get('LanguageImplementation', ...
                                 'ARM Compatible-ARM Cortex');
myProc.LanguageImplementations = existingImplementation;

objectsAdded = target.add(myProc);
```

To create a function for sharing the hardware device data, run:

```
target.export(myProc, 'FileName', 'exportMyProcFunction')
```

The `target.export` function creates `exportMyProcFunction.m` in the current working folder.

```
function registeredObjects = exportMyProcFunction(varargin)
% This function was generated using target data export.

    % Create target.Processor "MyManufacturer-MyProcessor"
    processor = target.create("Processor");
    processor.LanguageImplementations(1) = ...
      target.get("LanguageImplementation", "ARM Compatible-ARM Cortex");
    processor.Manufacturer = "MyManufacturer";
    processor.Name = "MyProcessor";

    % Add the target objects to MATLAB memory
    registeredObjects = target.add(processor, varargin{:});
```

Now, you can use the generated function to share the hardware device data in your database across computers and users. For example, on another computer, run this command.

```
objectsAdded = exportMyProcFunction;
```

The generated function recreates the `target.Processor` object, `MyManufacturer-MyProcessor`, and adds it to an internal database.

Create Alternative Identifier for Target Object

To create alternative identifiers for target objects, use the `target.Alias` class.

For example, if a `target.Processor` object has a long class identifier, you can create a `target.Alias` object that provides a short identifier for the `target.Processor` object.

1. Retrieve the `target.Processor` object.

   ```
   proccesorObj = target.get('Processor', ...
                              'Analog Devices-ADSP-CM40x (ARM Cortex-M)');
   ```

2. Use the `target.create` function to create a `target.Alias` object.

   ```
   aliasProcessorObj = target.create('Alias');
   ```

3. Use `target.Alias` object properties to specify the alternative identifier and original target object.

   ```
   aliasProcessorObj.Name = 'myShortName';
   aliasProcessorObj.For = proccesorObj;
   ```

4. Add the `target.Alias` object to an internal database.

   ```
   target.add(aliasProcessorObj);
   ```

5. To retrieve the original `target.Processor` object, run:

   ```
   target.get('Processor', 'myShortName');
   ```

Upgrade Data Definitions for Hardware Devices

To upgrade existing hardware device definitions that are specified through `rtwTargetInfo.m` or `sl_customization.m` files, use the `target.upgrade` function.

rtwTargetInfo.m File

Suppose you have the hardware device definition in an `rtwTargetInfo.m` file:

```
function rtwTargetInfo(tr)

    % Add registration function handle to the Target Registry
    tr.registerTargetInfo(@loc_register_hardware);
end

function hw = loc_register_hardware
    hw = RTW.HWDeviceRegistry;
    hw.Vendor = 'MyManufacturer';
    hw.Type = 'MyDevice';
    hw.Alias = {};
    hw.Platform = {'Prod', 'Target'};
    hw.setWordSizes([8 16 32 64 64 64 64 64 64 64 64]);
    hw.Endianess = 'Little';
    hw.IntDivRoundTo = 'Zero';
    hw.ShiftRightIntArith = true;
    hw.LargestAtomicInteger = 'Long';
    hw.LargestAtomicFloat = 'Double';
end
```

To upgrade the data definitions contained in the file, run:

```
target.upgrade('rtwTargetInfo', 'myPathTo/rtwTargetInfo.m');
```

In the current folder, the function creates this `registerUpgradedTargets.m` file:

```
function processor = registerUpgradedTargets(varargin)
% This function was generated using target data export.

    % Create target.LanguageImplementation 'MyManufacturer-MyDevice'
    languageimplementation = target.create('LanguageImplementation');
    languageimplementation.AtomicFloatSize = 64;
    languageimplementation.AtomicIntegerSize = 64;
    languageimplementation.DataTypes.Char.Size = 8;
    languageimplementation.DataTypes.Double.Size = 64;
    languageimplementation.DataTypes.Float.Size = 64;
    languageimplementation.DataTypes.Half.IsSupported = false;
    languageimplementation.DataTypes.Half.Size = 16;
    languageimplementation.DataTypes.Int.Size = 32;
    languageimplementation.DataTypes.Long.Size = 64;
```

```
    languageimplementation.DataTypes.LongLong.IsSupported = false;
    languageimplementation.DataTypes.LongLong.Size = 64;
    languageimplementation.DataTypes.Pointer.Size = 64;
    languageimplementation.DataTypes.PtrDiffT.Size = 64;
    languageimplementation.DataTypes.Short.Size = 16;
    languageimplementation.DataTypes.SizeT.Size = 64;
    languageimplementation.Name = 'MyManufacturer-MyDevice';
    languageimplementation.WordSize = 64;

    % Create target.Processor 'MyManufacturer-MyDevice'
    processor = target.create('Processor');
    processor.LanguageImplementations(1) = languageimplementation;
    processor.Manufacturer = 'MyManufacturer';
    processor.Name = 'MyDevice';

    % Add the target objects to MATLAB memory
    target.add(processor, varargin{:});
end
```

To register the hardware device with MATLAB, run:

```
registerUpgradedTargets()
```

If you want the registration to persist across MATLAB sessions, run:

```
registerUpgradedTargets('UserInstall', true)
```

sl_customization.m File

Suppose you have multiple hardware device definitions in an `sl_customization.m` file:

```
function sl_customization(cm)
  % sl_customization function to register a device
  % vendor and type with Simulink.
  % Copy or rename this file to sl_customization.m.
  cm.registerTargetInfo(@loc_register_device);
  cm.registerTargetInfo(@loc_register_device2);
  cm.registerTargetInfo(@loc_createConfig);

  cm.registerTargetInfo(@locRegisterTfl);
  cm.CodeCoverageTools.add('DummyCoverageToolForTesting',...
                           'HDummyCovTool',...
                           'A Coverage Tool Vendor');
end

function thisDev = loc_register_device
  thisDev = RTW.HWDeviceRegistry;
  thisDev.Vendor = 'MyDevVendor';
  thisDev.Type = 'MyDevType';
  thisDev.Alias = {};
  thisDev.Platform = {'Prod', 'Target'};
  thisDev.setWordSizes([8 16 32 32 32]);
  thisDev.LargestAtomicInteger = 'Char';
  thisDev.LargestAtomicFloat = 'None';
  thisDev.Endianess = 'Unspecified';
  thisDev.IntDivRoundTo = 'Undefined';
  thisDev.ShiftRightIntArith = true;
  thisDev.setEnabled({'IntDivRoundTo'});
end

function thisDev = loc_register_device2
  thisDev = RTW.HWDeviceRegistry;
  thisDev.Vendor = 'MyDevVendor';
  thisDev.Type = 'MyDevType2';
  thisDev.Alias = {};
  thisDev.Platform = {'Prod', 'Target'};
  thisDev.setWordSizes([8 16 32 32 32]);
  thisDev.LargestAtomicInteger = 'Char';
  thisDev.LargestAtomicFloat = 'None';
  thisDev.Endianess = 'Unspecified';
  thisDev.IntDivRoundTo = 'Undefined';
```

```
  thisDev.ShiftRightIntArith = true;
  thisDev.setEnabled({'IntDivRoundTo'});
end

% local function
function config = loc_createConfig
  config = rtw.connectivity.ConfigRegistry;
  config.ConfigName = 'Infineon->C16x, XC16x';
  config.ConfigClass = 'pil_slcust.HostDemoConfig1';
  config.SystemTargetFile = {'custom_target.tlc'};
  config.TemplateMakefile = {'custom_target.tmf'};
  config.TargetHWDeviceType = {'Infineon->C16x, XC16x'};
end

function thisTfl = locRegisterTfl
  thisTfl(1) = RTW.TflRegistry;
  thisTfl(1).Name = 'myTFL1';
  thisTfl(1).Description = 'Test';
  thisTfl(1).TableList = {'tfl_table_Sum',...
                          'tfl_table_Product',...
                         }; % Sum includes Add and Subtract
  thisTfl(1).BaseTfl = 'ANSI_C';
  thisTfl(1).TargetHWDeviceType = {'*'};
end
```

To upgrade the `RTW.HWDeviceRegistry` data definitions in the file, run:

```
target.upgrade('sl_customization', 'myPathTo/sl_customization.m')
```

In the current folder, the function creates this `registerUpgradedTargets.m` file:

```
function targetObjects = registerUpgradedTargets(varargin)
% This function was generated using target data export.

    % Create target.LanguageImplementation 'MyDevVendor-MyDevType'
    languageimplementation = target.create('LanguageImplementation');
    languageimplementation.AtomicIntegerSize = 8;
    languageimplementation.DataTypes.Char.Size = 8;
    languageimplementation.DataTypes.Double.Size = 64;
    languageimplementation.DataTypes.Float.Size = 32;
    languageimplementation.DataTypes.Half.IsSupported = false;
    languageimplementation.DataTypes.Half.Size = 16;
    languageimplementation.DataTypes.Int.Size = 32;
    languageimplementation.DataTypes.Long.Size = 32;
    languageimplementation.DataTypes.LongLong.IsSupported = false;
    languageimplementation.DataTypes.LongLong.Size = 64;
    languageimplementation.DataTypes.Pointer.Size = 32;
    languageimplementation.DataTypes.PtrDiffT.Size = 32;
    languageimplementation.DataTypes.Short.Size = 16;
    languageimplementation.DataTypes.SizeT.Size = 32;
    languageimplementation.Endianess = target.Endianess.Unspecified;
    languageimplementation.Name = 'MyDevVendor-MyDevType';
    languageimplementation.WordSize = 32;

    % Create target.Processor 'MyDevVendor-MyDevType'
    processor = target.create('Processor');
    processor.LanguageImplementations(1) = languageimplementation;
    processor.Manufacturer = 'MyDevVendor';
    processor.Name = 'MyDevType';

    % Create target.LanguageImplementation 'MyDevVendor-MyDevType2'
    languageimplementation2 = target.create('LanguageImplementation');
    languageimplementation2.AtomicIntegerSize = 8;
    languageimplementation2.DataTypes.Char.Size = 8;
    languageimplementation2.DataTypes.Double.Size = 64;
    languageimplementation2.DataTypes.Float.Size = 32;
    languageimplementation2.DataTypes.Half.IsSupported = false;
    languageimplementation2.DataTypes.Half.Size = 16;
    languageimplementation2.DataTypes.Int.Size = 32;
    languageimplementation2.DataTypes.Long.Size = 32;
    languageimplementation2.DataTypes.LongLong.IsSupported = false;
    languageimplementation2.DataTypes.LongLong.Size = 64;
```

```
    languageimplementation2.DataTypes.Pointer.Size = 32;
    languageimplementation2.DataTypes.PtrDiffT.Size = 32;
    languageimplementation2.DataTypes.Short.Size = 16;
    languageimplementation2.DataTypes.SizeT.Size = 32;
    languageimplementation2.Endianess = target.Endianess.Unspecified;
    languageimplementation2.Name = 'MyDevVendor-MyDevType2';
    languageimplementation2.WordSize = 32;

    % Create target.Processor 'MyDevVendor-MyDevType2'
    processor2 = target.create('Processor');
    processor2.LanguageImplementations(1) = languageimplementation2;
    processor2.Manufacturer = 'MyDevVendor';
    processor2.Name = 'MyDevType2';

    targetObjects = [processor, processor2];

    % Add the target objects to MATLAB memory
    target.add(targetObjects, varargin{:});
end
```

To register the hardware device definitions with MATLAB, run:

```
registerUpgradedTargets()
```

If you want the registration to persist across MATLAB sessions, run:

```
registerUpgradedTargets('UserInstall', true)
```

See Also

`target.LanguageImplementation` | `target.Processor`

More About

- “Configure Run-Time Environment Options” on page 10-2

Model Protection in Simulink Coder

- “Protect Models to Conceal Contents” on page 11-2
- “Create Protected Models with Multiple Targets” on page 11-9
- “Test Protected Models” on page 11-10
- “Package and Share Protected Models” on page 11-12
- “Specify Custom Obfuscators for Protected Models” on page 11-16
- “Define Callbacks for Protected Models” on page 11-17
- “Sign a Protected Model” on page 11-19
- “Best Practices for Creating Protected Models” on page 11-21

Protect Models to Conceal Contents

When you want to share a model with a third party without revealing intellectual property, protect the model. Protecting a model conceals the implementation details of the original model by compiling it into a referenced model. The protected model includes derived files to support the optional functionalities that you specify.

When you protect a model, you can allow the user of the protected model to:

- Simulate a model that includes the protected model in normal, accelerator, or rapid accelerator mode.
- Open a read-only web view of the model, including model contents and block parameters. Creating a read-only web view of the model requires Simulink Report Generator™.
- Generate code for a model that includes the protected model.
- Generate HDL code for a model that includes the protected model.
- Generate code for the protected model through the standalone interface, if you have Embedded Coder and specify an ERT-based system target file for the model.

You can optionally password-protect each option. If you choose-password protection for one of these options, the software protects the supporting files by using AES-256 encryption.

When you create a protected model:

- Simulink creates and stores a protected version of the model in a file that has the same name as the source model, with an `.slxp` extension.
- The original model file, with the `.slx` extension, does not change. If you protect the model through a Model block, that Model block does not change.
- Optionally, Simulink creates a project archive (`.mlproj`) that contains the protected model, a harness model for the protected model, and additional supporting files.

This example shows how to create a protected model from a referenced model for read-only viewing, simulation, or code generation.

Prepare the Parent Model

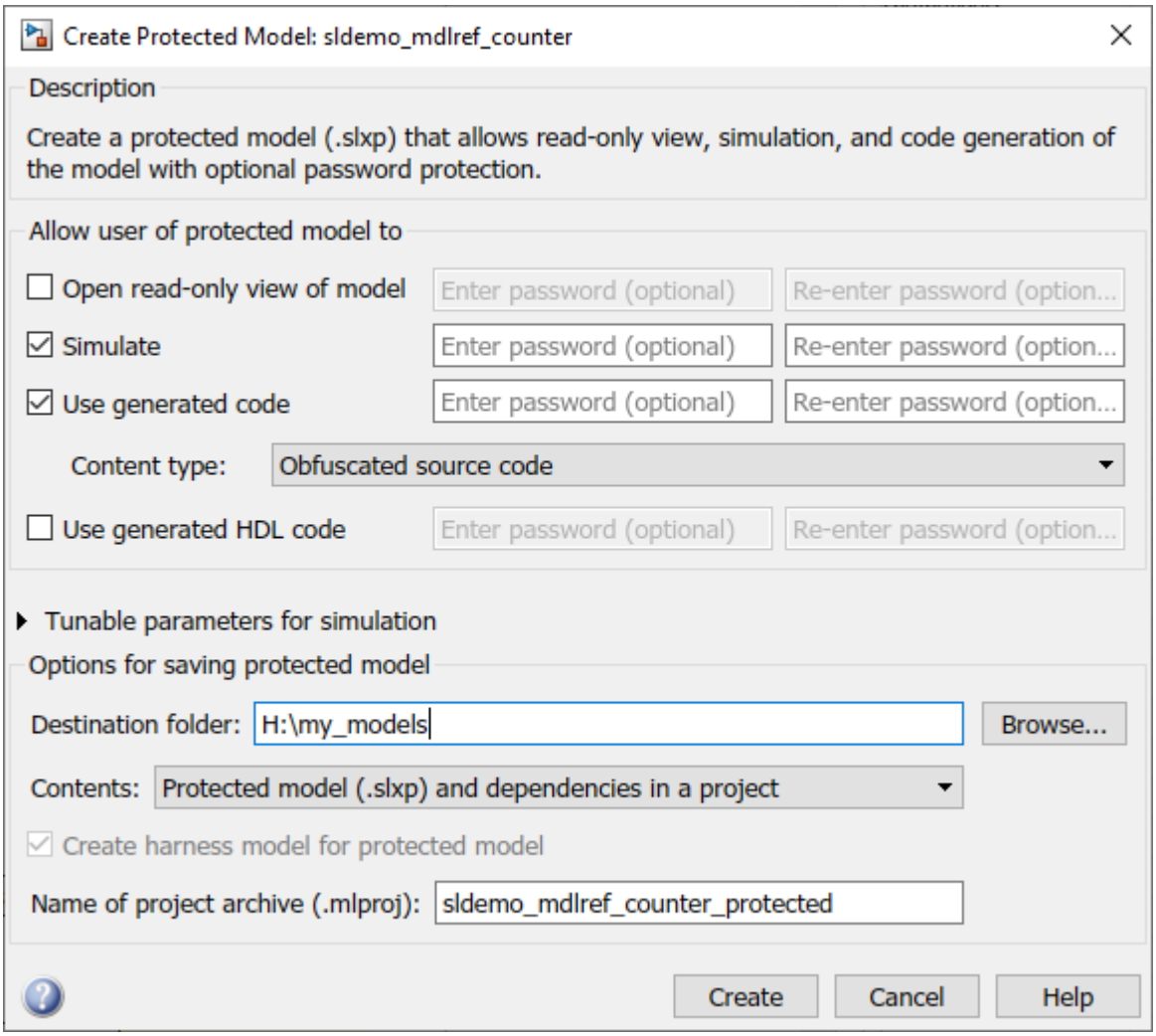
Configure the Model blocks in the parent model to refer to the original referenced model. This step prevents the Model blocks from becoming protected references when you create the protected model.

1. Open the parent model that references the model you want to protect. For this example, open the model `sldemo_mdlref_basic`.
2. To run the workflow, create a local copy of the model `sldemo_mdlref_counter` that you want to protect. You can then create a local copy of the parent model `sldemo_mdlref_basic`. You must save the parent model in the same folder as the referenced model.
3. Open the `sldemo_mdlref_basic` model that you saved locally. Make sure that the Model blocks `CounterA`, `CounterB`, and `CounterC` reference the `sldemo_mdlref_counter` model that you saved locally.
4. For each Model block, open the Block Parameters dialog box and specify the extension `.slx` in the **Model name** field. When both the model and the protected model exist in the same

folder, `.slxp` takes precedence over `.slx`. If you do not specify an extension, then the original Model block in the model refers to the protected model instead of the original model. Click **OK**.

Protect the Referenced Model

1 In the `sldemo_mdlref_basic` model, click any of the three Model blocks. On the Simulink Toolstrip **Model Block** tab, click **Protect**.



2 In the Create Protected Model dialog box, select the **Simulate** and **Use generated code** check boxes. These options allow the protected model user to simulate and generate code for a model that references the protected model. If you want to password-protect the functionality of the protected model, enter a password with a minimum of eight characters. Each option can have a unique password.

3 If you have Embedded Coder and specify an ERT-based system target file (for example, `ert.tlc`) for the model, the **Code interface** field is visible.

In this example, `sldemo_mdlref_basic` does not specify an ERT-based system target file, therefore the **Code interface** options are not available in the Create Protected Model dialog box.

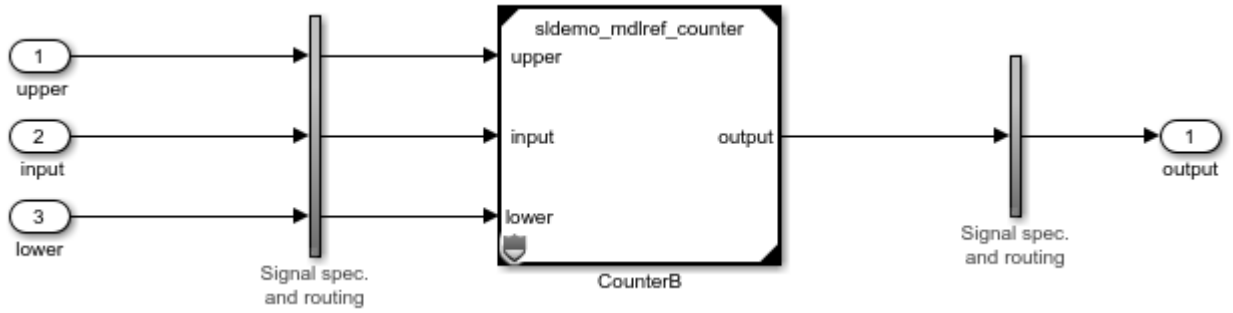
From the **Code interface** drop-down list, select one of these options:

- `Model reference` — Specifies code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. Users can run Model block SIL/PIL simulations with the protected model.
- `Top model` — Specifies code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

4. From the **Content type** list, select `Obfuscated source code` to conceal the source code purpose and logic of the protected model. For more information on the model protection options, see “Content type”.
5. Expand the **Tunable parameters for simulation** section and select the parameters that you want the protected model user to be able to tune during simulation.

   For code generation, parameters with non-auto storage classes are tunable. For more information, see “Storage Classes for Parameters and Signals Used in Model Blocks”.
6. In the **Destination folder** box, specify the folder path for the protected model. The default value is the current working folder.
7. To automatically collect, create, and package supporting files with the protected model, set **Contents** to `Protected Model (.slxp) and dependencies in a project`.

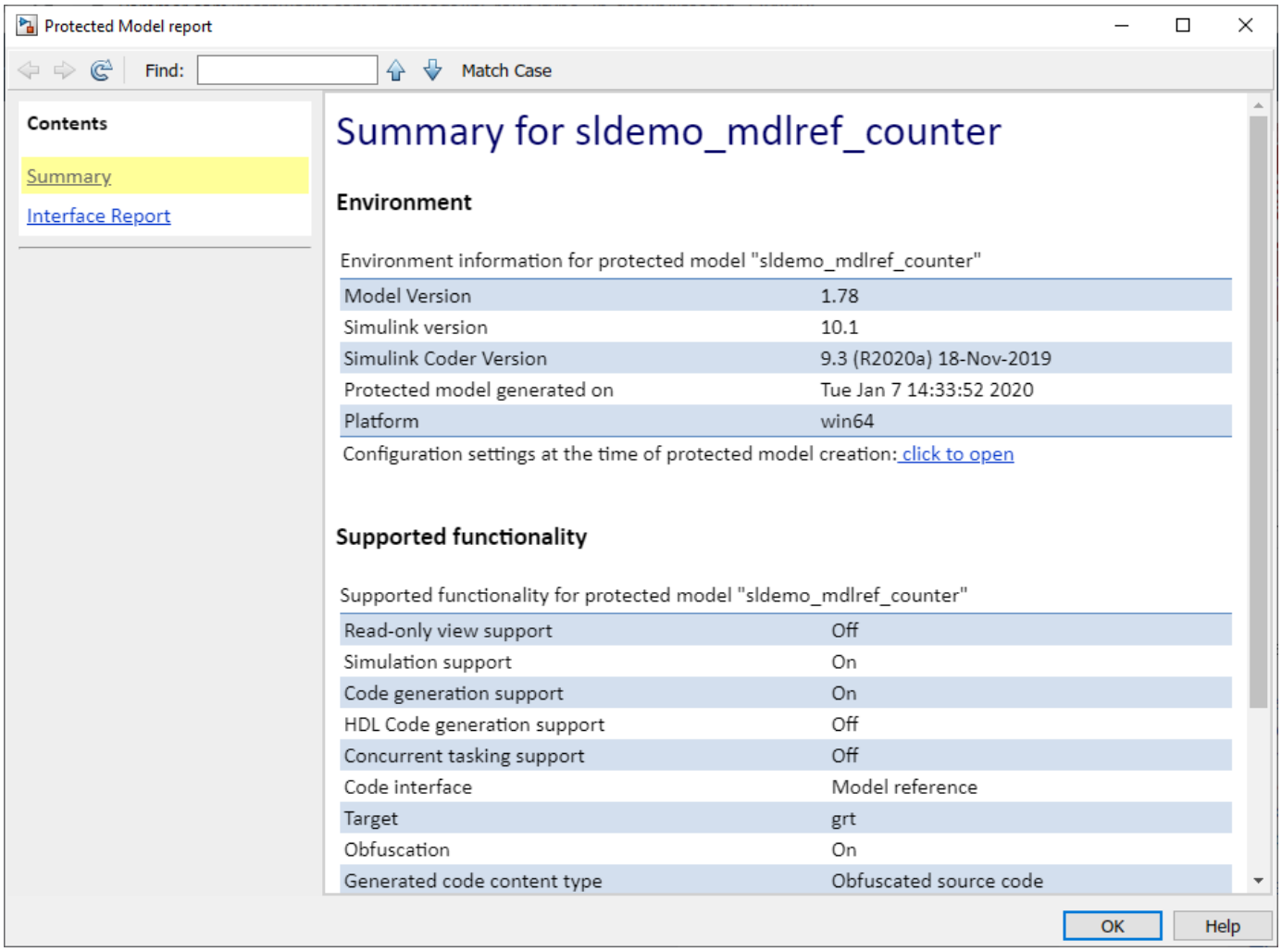
   Note Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.
8. Setting **Contents** to `Protected Model (.slxp) and dependencies in a project` selects the **Create harness model for protected model** check box. The harness model is included in the project and provides an isolated environment for the Model block that references the protected model.



9. To further customize your protected model, you can:
   - Specify a custom obfuscator for code obfuscation. See “Specify Custom Obfuscators for Protected Models” on page 11-16.
   - Specify multiple code generation targets. See “Create Protected Models with Multiple Targets” on page 11-9.
   - Define callbacks for the protected model. See “Define Callbacks for Protected Models” on page 11-17.
10. Click **Create**.

A project archive (`.mlproj`) that contains the protected model, harness model, and supporting files appears in the specified destination folder. To test the protected model, extract the contents of the project archive by double-clicking it in the Current Folder browser and specifying the destination folder. Alternatively, right-click on the project archive and click **Extract Here**. The project opens, and the extracted files are in a new folder named after the project archive.

11 When you create the protected model from the Simulink Editor, a protected model report is generated and included as part of the protected model. For this example, to view the protected model report, double-click the protected model or right-click the protected-model badge icon on the block in the harness model and select **Display Report**.



The report contains:

- A **Summary**, including the following tables:
  - **Environment**, providing the Simulink version and other product versions and the platform used to create the protected model.
  - **Supported functionality**, reporting `On`, `Off`, or `On with password protection` for each possible functionality that the protected model supports. If you configure your protected model for multiple targets, this table includes a list of supported targets.

- An **Interface Report**, including model interface information such as input and output specifications, exported function information, interface parameters, and data stores.

The report does not contain other sections that appear in a code generation report for non-protected models, such as a code metrics report and traceability report, because these reports include more information about the model's design.

12 You can test the protected model to compare it to the original model. For more information, see “Test Protected Models” on page 11-10.

13 To use the protected model in a model hierarchy, reference it through a Model block such as the one included in the harness model. The **Simulation mode** for Model blocks that reference a protected model is set to `Accelerator`. You cannot change the mode. For more information, see “Reference Protected Models from Third Parties”.

Other Methods of Protecting Models

To create a protected model from the current model, use one of these options:

- On the Simulink Toolstrip **Simulation** tab, click the **Save** button arrow, then select **Protected Model**.
- On the Simulink Toolstrip **C Code** tab, click the **Share** button arrow, then select **Generate Protected Model**.

To programmatically create a protected model, use the `Simulink.ModelReference.protect` function.

Protected Model Requirements and Limitations

When you create a protected model, consider these following requirements:

- You must have a Simulink Coder or HDL Coder™ license to create a protected model.
- The model must be available on the MATLAB path.
- The model cannot have unsaved changes.
- The model uses the configuration that is active during protection. You cannot change the configuration of a protected model.
- If the model contains variants, the protected model includes only the variant that is active during protection.
- The model protection process does not preserve callbacks. For more information on creating callbacks for use with a protected model, see “Define Callbacks for Protected Models” on page 11-17
- Do not rename the protected model or change its suffix. If you do so, the model is unusable until you restore its original name and suffix.
- Use a unique name for the model and for models that it references. If a protected model references a model that shares a name with either a different protected model or a different model within the hierarchy of another protected model, there are limitations for using the protected models. If a top model references two protected models that have such a naming conflict, you cannot protect the top model, generate code for the top model, or simulate the top model in software-in-the-loop (SIL), processor-in-the-loop (PIL), or rapid accelerator modes.
- If your model includes a non-inlined S-function, use a unique name for the .MEX file. If there is a name conflict with a different .MEX file on the MATLAB path, you cannot simulate the protected model.

The model must also meet the requirements listed in “Model Reference Requirements and Limitations”.

Code Generation Requirements and Limitations

To create a protected model that supports code generation, the model being protected must meet these requirements:

- The model must use normal, accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) modes and a single target.
- The model must not include noninlined S-functions.
- The model must not have the **Code Generation > Verification > Measure function execution time** configuration parameter selected. If this option is selected when you protect a model, the software turns off the parameter and displays a warning.
- Protected referenced models must support code generation without password protection.
- The model must be compatible with the **Content type** of each protected referenced model. This table provides compatibility information.

  Code Generation Content Type Compatibility

Protected Parent Model Content Type	Compatible Protected Referenced Model Content Types
`Binaries`	• `Binaries` • `Obfuscated source code`
`Binaries` with `'ObfuscateCode'` set to `'false'`	• `Binaries` • `Binaries` with `'ObfuscateCode'` set to `'false'` • `Obfuscated source code` • `Readable source code`
`Obfuscated source code`	• `Obfuscated source code`
`Readable source code`	• `Obfuscated source code` • `Readable source code`

  When the **Content type** of the protected parent model and protected referenced models do not match, code generation applies the **Content type** that provides the higher level of protection. For example, a protected parent model set for `Binaries` generates `Binaries` for protected referenced models that are set to `Obfuscated source code`. A protected parent model set for `Readable source code` generates `Obfuscated source code` for protected referenced models that are set to `Obfuscated source code`.

To avoid an error during code generation of a model that includes a protected model:

- The protected model name must be unique from other model names in the same model reference hierarchy.
- The interfaces must match.
- The parameters must be compatible.

Nested Protected Model Requirements and Limitations

To enable a recipient of your protected model to protect a model that references it, your protected model must:

- Support accelerator mode. You must select **Simulate** in the Create Protected Model dialog box or set `'Mode'` to `'Accelerator'` or `'CodeGeneration'` using the `Simulink.ModelReference.protect` function.
- Use the default `'CodeInterface'` setting: `'Model reference'`.
- Not use password protection for simulation.
- Not have callbacks.
- Support without password protection the operations that the protected parent model is intended to support.
- Use a **Content type** that is compatible with the intended **Content type** of the protected parent model if the protected parent model is intended to support code generation. For a list of compatible **Content type** options, see the table in the preceding section.
- Use the same compiler as the protected parent model is intended to use.
- Not reference a model that shares a name with either a different protected model or a different model within the hierarchy of another protected model.

If the model that you want to protect references a protected model, the referenced protected model must meet the preceding requirements.

When you protect a model that references a protected model, you must specify the tunable parameters of the referenced protected model as tunable for the model you are protecting.

Note A protected model that you create with the **Use generated HDL code** option selected allows encryption and support for simulation and HDL code generation from a model that references the protected model. You cannot obfuscate the HDL source code, have callbacks, or use nested protected models with this option. To learn more about HDL code generation limitations, see “Protected Model Restrictions for HDL Code Generation” (HDL Coder).

See Also

More About

- “Package and Share Protected Models” on page 11-12
- “Specify Custom Obfuscators for Protected Models” on page 11-16
- “Create Protected Models with Multiple Targets” on page 11-9
- “Define Callbacks for Protected Models” on page 11-17

Create Protected Models with Multiple Targets

You can create a protected model that supports multiple code generation targets. This example shows how to use command-line functions to create a protected model that supports code generation for GRT and ERT targets.

1. Load a model and save a local copy. This model is configured for a GRT target.

```
openExample('sldemo_mdlref_counter');
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

2. Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

3. Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

4. Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

```
st =

    'grt'    'sim'
```

5. Configure the unprotected model to support an ERT target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

6. Add support to the protected model for the ERT target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

7. Verify that the list of supported targets now includes the ERT target.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

```
st =

    'ert'    'grt'    'sim'
```

See Also

More About

- “Protect Models to Conceal Contents” on page 11-2

Test Protected Models

To test a protected model that you created, compare the output of the protected model to the output of the original model. Because you supply the protected model from the original model, both the original and the protected model might exist on the MATLAB path. In the parent model, if the Model block **Model name** parameter names the model without providing a suffix, the protected model takes precedence over the unprotected model. To override this default when testing the output, in the Model block **Model name** parameter, specify the file name with the extension of the unprotected model, `.slx`.

To compare the unprotected and protected versions of a Model block, use the Simulation Data Inspector. This example uses `sldemo_mdlref_basic` and the protected model, `sldemo_mdlref_counter.slxp`, which is created in “Protect Models to Conceal Contents” on page 11-2.

1. If it is not already open, open `sldemo_mdlref_basic`.

   ```
   openExample('sldemo_mdlref_basic');
   ```

2. Enable logging for the output signal of the Model block, `CounterA`. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select the **Signal logging** parameter. Click **Apply** and **OK**.
3. Right-click the output signal. From the context menu, select `Properties`. In the Signal Properties dialog box, select **Log signal data**. Click **Apply** and **OK**. For more information, see “Save Signal Data Using Signal Logging”.
4. Right-click the `CounterA` block. From the context menu, select `Block Parameters (ModelReference)`. In the Block Parameters dialog box, specify the **Model name** parameter with the name of the unprotected model and the extension, `sldemo_mdlref_counter.slx`. Click **Apply** and **OK**. Repeat this step for `CounterB` block and `CounterC` block.
5. Simulate the model. When the simulation is complete, click the **Data Inspector** button to open the Simulation Data Inspector.
6. In the Simulation Data Inspector, rename the run to indicate that it is for the unprotected model.
7. In the Simulink Editor, right-click the `CounterA` block. From the context menu, select `Block Parameters (ModelReference)`. In the Block Parameters dialog box, specify the **Model name** parameter with the name of the protected model, `sldemo_mdlref_counter.slxp`. A badge icon appears on the Model block. Repeat this step for `CounterB` block and `CounterC` block.
8. Simulate the model, which now refers to the protected model. When the simulation is complete, a new run appears in the Simulation Data Inspector.
9. In the Simulation Data Inspector, rename the new run to indicate that it is for the protected model.
10. In the Simulation Data Inspector, click the **Compare** tab. From the **Baseline** and **Compare To** lists, select the runs from the unprotected and protected model, respectively. To compare the runs, click **Compare Runs**. For more information about comparing runs, see “Compare Simulation Data”.

See Also

More About

- “Protect Models to Conceal Contents” on page 11-2
- “Package and Share Protected Models” on page 11-12

Package and Share Protected Models

When you protect a model, you can automatically create and package the following contents in a project archive (`.mlproj`) for easy sharing:

- Protected model file (`.slxp`)
- Harness model file
- MAT-file with base workspace definitions
- Data dictionary pruned to relevant definitions
- Other supporting files

In the Create Protected Model dialog box, set **Contents** to `Protected model (.slxp) and dependencies in a project`.

Note Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.

Alternatively, you can use one of these options to deliver the protected model package:

- Create a project archive to share a project that contains the protected model file and supporting files. For more information, see “Create a Project from a Model” and “Share Projects”.
- Provide the protected model file and supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Provide the files in some other standard or proprietary format specified by the receiver.

Whichever approach you use to deliver a protected model, include information on how to retrieve the original files.

Harness Model

You can create a harness model when you create your protected model. The harness model contains a Model block that references the protected model. A third party can use the Model block to reference your protected model. The harness model is set up for simulation of the protected model.

MAT-File with Base Workspace Definitions

Referenced models can use object definitions or tunable parameters that are defined in the MATLAB base workspace. These variables are not saved with the model. When you protect a model, you must obtain the definitions of required base workspace entities and ship them with the model.

For example, if the model uses the following base workspace variables, they must be saved to a MAT-file:

- Global tunable parameter
- Global data store
- The following objects used by a signal that connects to a root-level model Inport or Outport:

- `Simulink.Signal`
- `Simulink.Bus`
- `Simulink.Alias`
- `Simulink.NumericType` that is an alias

To determine the required base workspace definitions and save them to a MAT-file, see “Explore Protected Model Capabilities”. Before executing the protected model as a part of a third-party model, the receiver of the protected model must load the MAT-file.

Simulink Data Dictionary

Referenced models can use data definitions from a data dictionary, which are not saved with the model. When you protect a model that uses a data dictionary, package and ship the data dictionary with the protected model.

Protected Model File Contents

A protected model file (`.slxp`) contains derived files that support the options that you select when you create the protected model. The derived files are unpacked when you or a third party use the protected model in simulation or code generation.

The derived files that are unpacked depend on the support that you enable when creating the protected model. The `slprj/sim/`*`model`*`/*` files are deleted after they are used.

This table provides the files that are unpacked depending on the options that you specify.

Supported Functionality	Derived Files
Simulation only and the referencing model simulates in normal mode	The *`model`*`_msp.`*`mexext`* file is placed in the build folder. If the model uses a noninlined S-function directly or indirectly, the S-function MEX-files are placed in the build folder. The `slprj/sim/`*`model`*`/tmwinternal/*` files are unpacked in the `slprj/sim/` folder.
Simulation only and the referencing model simulates in accelerator or rapid accelerator mode	In addition to the preceding files in the build folder, these files are unpacked in the `slprj/sim/` folder: • `slprj/sim/`*`model`*`/*.h` • `slprj/sim/`*`model`*`/`*`model`*`lib.a` (or *`model`*`lib.lib`) • `slprj/sim/`*`model`*`/tmwinternal/*` • `slprj/sim/_sharedutils/*` For the protected model report, these additional files are unpacked (but not in the build folder): • `slprj/sim/`*`model`*`/html/*` • `slprj/sim/`*`model`*`/buildinfo.mat`

Supported Functionality	Derived Files
Code generation	These files are unpacked in the `slprj` folder after building your model (in addition to the preceding files): • `slprj/sim/model/*.h` • `slprj/sim/model/modellib.a` (or `modellib.lib`) • `slprj/sim/model/tmwinternal/*` • `slprj/sim/_sharedutils/*` • `slprj/target/model/*.h` • `slprj/target/model/model_rtwlib.a` (or `model_rtwlib.lib`) • `slprj/target/model/buildinfo.mat` • `slprj/target/model/codeinfo.mat` • `slprj/target/_sharedutils/*` • `slprj/target/model/tmwinternal/*`
Code generation with `Top model` code interface (requires Embedded Coder license)	These files are unpacked in the `slprj` folder after building your model (in addition to the preceding files): • `slprj/sim/model/*.h` • `slprj/sim/model/modellib.a` (or `modellib.lib`) • `slprj/sim/model/tmwinternal/*` • `slprj/sim/_sharedutils/*` • `model_target_rtw/*.h` • `model_target_rtw/*.objExt` • `model_target_rtw/buildinfo.mat` • `model_target_rtw/codeinfo.mat` • `slprj/target/_sharedutils/*` • `slprj/target/model/tmwinternal/*` For the protected model report, after building your model these files are unpacked (in addition to the preceding files): • `slprj/target/model/html/*` • `slprj/target/model/buildinfo.mat` • `slprj/target/_sharedutils/html/*`

Supported Functionality	Derived Files
HDL code generation (requires HDL Coder license)	The files are unpacked in the `hdlsrc` folder: • `hdlsrc/model/model.vhd` (`model.v` if you specified `Verilog` as the **Target language**.) • `hdlsrc/model/Subsystem.vhd` (`Subsystem.v` if you specified `Verilog` as the **Target language** of the model that you protected. The additional HDL files depend on how hierarchically the referenced model was designed.) • `hdlsrc/model/model_pkg.vhd` (This file is not generated if you specified `Verilog` as the **Target language** of the model that you protected.) • `hdlsrc/model/model_report.html` • `hdlsrc/model/gm_model.slxp` (This file is a generated protected model.)

See Also

More About

- “Protect Models to Conceal Contents” on page 11-2

Specify Custom Obfuscators for Protected Models

When creating a protected model, you can specify your own postprocessing function for files that the protected model creation process generates. Before packaging the protected model files, this function is called by the `Simulink.ModelReference.protect` function. You can use this functionality to run your own custom obfuscator on the generated files by following these steps:

1. Create your postprocessing function. Use this function to call your custom obfuscator. The function must be on the MATLAB path and accept a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable.
2. In your function, get the files and exported symbol information that your custom obfuscator requires to process the protected model files. To get the files and information, access the properties of your function input variable. The variable is a `Simulink.ModelReference.ProtectedModel.HookInfo` object with the following properties:
   - `SourceFiles`
   - `NonSourceFiles`
   - `ExportedSymbols`
3. Pass the protected model file information to your custom obfuscator. The following is an example of a postprocessing function for custom obfuscation:

   ```
   function myHook(protectedModelInfo)

       % Get source file list information.
       srcFileList = protectedModelInfo.SourceFiles;
       disp('### Obfuscating...');
       for i=1:length(srcFileList)
           disp(['### Obfuscator: Processing ' srcFileList{i} '...']);
           % call to custom obfuscator
           customObfuscator(srcFileList{i});
       end
   end
   ```
4. Specify your postprocessing function when creating the protected model:

   ```
   Simulink.ModelReference.protect('myModel, ...
       'Mode', ...
       'CodeGeneration', ...
       'CustomPostProcessingHook', ...
       @(protectedModelInfo)myHook(protectedModelInfo))
   ```

The protected model creator can also enable obfuscation of simulation target code and generated code through the ‘`ObfuscateCode`’ option of the `Simulink.ModelReference.protect` function. Your custom obfuscator runs only on the generated code and not on the simulation target code or the generated HDL code. If both obfuscators are in use, the custom obfuscator is the last to run on the generated code before the files are packaged.

See Also

More About

- “Protect Models to Conceal Contents” on page 11-2

Define Callbacks for Protected Models

When you create a protected model, you can customize its behavior by defining callbacks. Callbacks specify code that executes when you view, simulate, or generate code for the protected model. You cannot have protected model callbacks with HDL code generation support enabled for a protected model. To learn more about HDL code generation limitations, see “Protected Model Restrictions for HDL Code Generation” (HDL Coder).

A protected model user cannot view or modify a callback. If a model references a protected model with callbacks, you cannot protect the model.

To create a protected model with callbacks:

1. Define `Simulink.ProtectedModel.Callback` objects for each callback.
2. To create your protected model, call the `Simulink.ModelReference.protect` function. Use the `'Callbacks'` option to specify a cell array of callbacks to include in the protected model.

Creating Callbacks

To create and define a protected model callback, create a `Simulink.ProtectedModel.Callback` object. Callback objects specify:

- The code to execute for the callback. The code can be a character vector of MATLAB commands or a script on the MATLAB path.
- The event that triggers the callback. The event can be `'PreAccess'` or `'Build'`.
- The protected model functionality that the event applies to. The functionality can be `'CODEGEN'`, `'SIM'`, `'VIEW'`, or `'AUTO'`. If you select `'AUTO'`, and the event is `'PreAccess'`, the callback applies to each functionality. If you select `'AUTO'`, and the event is `'Build'`, the callback applies only to `'CODEGEN'` functionality. If you do not select a functionality, the default behavior is `'AUTO'`.
- The option to override the protected model build process. This option applies only to `'CODEGEN'` functionality.

You can create only one callback per event and per functionality.

Defining Callback Code

You can define the code for a callback by using either a character vector of MATLAB commands or a script on the MATLAB path. When you write callback code, follow these guidelines:

- Callbacks must use MATLAB code (`.m` or `.p`).
- The code can include protected model functions or a MATLAB command that does not require loading the model.
- Callback code must not call out to external utilities unless those utilities are available in the environment where the protected model is used.
- Callback code cannot reference the source protected model unless you are using protected model functions.

You can use the `Simulink.ProtectedModel.getCallbackInfo` function in callback code to get information on the protected model. The function returns a

`Simulink.ProtectedModel.CallbackInfo` object that provides the protected model name and the names of submodels. If the callback is specified for `'CODEGEN'` functionality and `'Build'` event, the object provides the target identifier and model code interface type (`'Top model'` or `'Model reference'`).

Create a Protected Model with Callbacks

This example creates a protected model with a callback for code generation.

1. On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
s1 = 'Code interface is: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
disp([s1 cbinfobj.CodeInterface]);
```

2. Create a callback that uses the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m');
```

3. Create the protected model and specify the code generation callback.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

4. Build the protected model. Before the build, the callback displays the code interface.

```
slbuild('sldemo_mdlref_basic')
```

See Also

`Simulink.ProtectedModel.Callback` | `Simulink.ProtectedModel.getCallbackInfo` | `Simulink.ModelReference.protect`

More About

- “Protect Models to Conceal Contents” on page 11-2

Sign a Protected Model

A *digital signature* is a method of verifying information about the origin of a digital message or file. You can digitally sign a protected model to allow the user of the model to verify:

- That the model was created by you, the publisher that signed the model.
- That the model was not changed after you signed it.

By using digital signature validation, the recipient can detect issues, such as an unfamiliar publisher or a tampered model, before opening or executing a protected model.

These parties interact in the digital signing workflow:

- The creator of the protected model
- The recipient who uses the protected model
- An external certificate authority (CA) that the creator and the recipient trust

To exchange a protected model with certificate validation, they use this workflow:

1. The certificate authority provides a digital certificate, which proves that a public key belongs to the creator of the model. The certificate authority also provides the corresponding public certificate to the model recipient.
2. The creator signs the protected model by using the private key that corresponds to the public key in the certificate. The creator sends the signed protected model to the recipient.
3. The recipient verifies the signature on the protected model. Verification compares the signature on the model to the certificate attached to the model and to the certificate of the CA. Verification fails if:
   - The protected model was changed after it was signed.
   - The protected model was not signed.
   - The protected model was signed with an expired certificate.
   - The protected model was self-signed with a certificate issued by the author.
   - The protected model was signed with a certificate for which the key usage does not include digital signing.
   - The protected model was signed with a missing or invalid certificate.
   - The certificate of the CA is missing in your system or is invalid.

As the model creator, you generate a public-private key pair and obtain a certificate for the public key from the CA. Then, you create a PKCS #12 file (`.pfx` file or `.p12` file) that includes the private key and the certificate. Because the PKCS #12 file contains the private key, it is typically password-protected. Follow this workflow to sign your protected model before you send it to the recipient.

Sign Your Protected Model

To sign a protected model, use a certificate that is issued to you by a certificate authority (CA) that you and the user of the model both trust. If you sign a protected model using a self-signed certificate or a certificate issued by an internal CA, the recipient's system must trust the certificate for the signature validation to succeed. Using a self-signed certificate to sign protected models is not recommended.

1. Locate the PKCS #12 file that you or your organization created from the private key and the certificate. The file must have a `.pfx` or `.p12` extension.
2. Protect the model that you want to sign. The protected version of the model has the extension `.slxp`.
3. Provide the password for the PKCS #12 file.

   ```
   Simulink.ModelReference.ProtectedModel.setPasswordForCertificate('certificate_file.pfx',...
   'password');
   ```

4. Sign the model by using the PKCS #12 file.

   ```
   Simulink.ProtectedModel.sign('protected_model.slxp','certificate_file.pfx');
   ```

Your protected model is now signed with the PKCS #12 file. Do not change the protected model after you sign it. If you or another party changes the signed protected model, verification of the signature fails and the protected model cannot be used.

The recipient can verify the protected model by using the function `Simulink.ProtectedModel.verifySignature` or by selecting the Simulink preference Verify digital signature of protected model before opening. For more information, see “Reference Protected Models from Third Parties”.

See Also

`Simulink.ProtectedModel.sign`

More About

- “Protect Models to Conceal Contents” on page 11-2
- “Reference Protected Models from Third Parties”

Best Practices for Creating Protected Models

When you want to conceal the implementation details of a model that you share with a third party, create a protected model. Instead of including your original model files, the protected model includes derived files to support the functionalities that you specify. To create a protected model that supports functionalities that your user requires and that includes minimal information about your model, follow these best practices. The settings that you choose for the model before and during protection specify what the user of the protected model can do with it. Consider the trade-offs of different design options. To support additional functionalities, the protected model must include more content and therefore, more elements of your design might be visible. Because the protected model uses derived artifacts from a compiled model reference, you cannot change the model after you protect it.

Coordinate Settings with the User of Your Protected Model

When you develop a model that you will protect, coordinate with the user of the protected model to choose settings that allow their system to reference your protected model. After you protect your model, you cannot modify options such as configuration parameter settings to make your protected model compatible with a parent model. Before you protect your model, configure it to use compatible options.

Choose a Unique Model Name

You cannot change the name of your protected model after you protect it. To avoid name conflicts, choose a model name that is unique from other models in the system. Choose a model name that does not conflict with other models within your user's model hierarchy. Use the name to save your source (unprotected) model before you protect it.

Configure Model Parameters

Before you protect your model, choose configuration parameter settings by coordinating with the user of your protected model. When the parent model references your protected model, the configuration parameters of the models must be compatible. For example, if you set the parameter **Solver** to `Variable-step` for your protected model, the model that references your protected model must have **Solver** set to `Variable-step`. If your model contains multiple configuration sets, the protected model uses the configuration set that is active when you protect the model. To minimize parameter incompatibilities, send your model configuration set to the user of your protected model so that they can use the configuration for their parent model. For more information, see “Share a Configuration with Multiple Models”.

Minimize the Model External Interface

Except for input and output ports, minimize the number of design elements that provide an interface into your model. To allow interfacing with external components, the protected model does not protect those design elements. For the following items, consider these alternative options that allow protection.

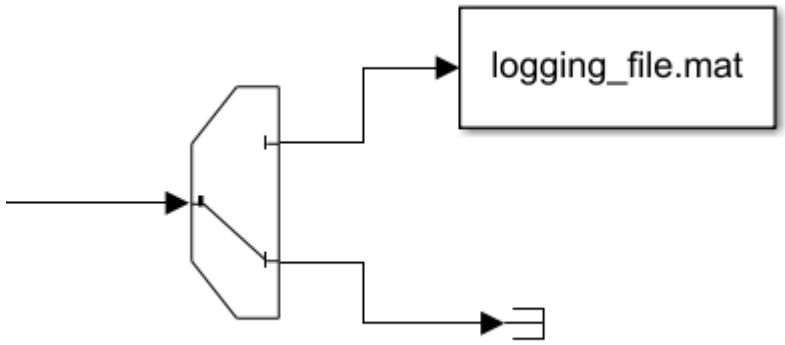
- Tunable Parameters — Minimize the number of tunable parameters in the code that use a storage class other than `Auto` or `Localizable`. Parameters that use a global storage class in your model's generated code are visible in the code for the protected model. If the user of your protected model requires access to tunable parameters, agree on a list of the parameters that the protected model provides. For more information, see “C Code Generation Configuration for Model Interface Elements”.

- Data Objects — Data objects, including parameter and signal data objects, data type objects, and bus objects, that you store externally are not protected. Include only data objects that the user of the protected model requires access to. Use the data objects to monitor signals and calibrate parameters while you develop your model. After calibration, specify values for the parameters, and then inline them. For more information, see “Data Objects”.
- Data Stores — Do not use global data stores on signal objects. Global data stores can make data visible across referenced models. Instead, use inports and outports to communicate data across model references. You can use Data Store Memory blocks as data stores that are local to your model. For more information, see “Local and Global Data Stores”.
- Model Arguments — Minimize the number of arguments that your model uses. When your protected model uses arguments, the user of your protected model must specify values for the arguments in the parent model. These data items are visible to the user of your protected model. For each model argument that your model uses, communicate the argument specifications to the user of your protected model.
- Simulink Functions — Consider the scope of Simulink functions that your model uses. When a Simulink function has a global scope or is scoped at the root level of your model, it provides a visible interface with your protected model. If you do not want a Simulink function to interface into your protected model, consider specifying a scope within a subsystem of your model. For more information, see “Scoped Simulink Function Blocks in Subsystems”.
- Logging Blocks — Do not use block-based logging in your protected model. To File blocks and To Workspace blocks that use the `Timeseries` format function in protected models, and therefore make the logged data visible to the protected model user. Signal logging, and other options that use the signal selector, do not work for signals inside a protected model.
- Noninlined S-functions — Consider the content of noninlined S-functions that your model uses. When you protect a model that contains a noninlined S-function, the MEX files that you write for the S-function are placed in the protected model build folder and are visible to the user of your protected model.

Design Options in Your Protected Model

If you want to use one of the preceding design elements while you develop your model, but the element is not required by the user of the model, you can use variants to exclude the design element from your protected model. An example is using a To File block during testing to log data that the user of the protected model does not require access to. To exclude the block by using variants:

1. Insert a variant block, such as a Manual Variant Sink block, on the signal that leads to the To File block.
2. Connect the first output of the variant block to the To File block.
3. Connect the second output of the variant block to a Terminator block to terminate the signal.
4. To log the signal while testing the model, activate the first variant output.
5. To protect the model without using the logging block, activate the second variant output. The protected model uses only the active variant, so the signal is terminated without being logged.



Contents to Include in Your Protected Model

Model Protection Options

When you specify functionalities for your protected model, the protected model includes artifacts to support those functionalities. Limiting the included artifacts enables you to control what contents are visible to the user of your model or to an external attacker that wants to view or change your model without permission. Consider the trade-offs of different design options. To support additional functionalities, the protected model must include more content and therefore, more elements of your design might be visible. Coordinate with the user of your model to agree on what functionalities they require. Specify support for only those functionalities.

- Obfuscated source code — Use this option when the user of your protected model must compile the generated source code. The protected model includes an obfuscated version of the generated source code. It behaves the same as your source code but is more difficult to understand for the model user and potential attackers.
- Compiled code — Use this option when the user of your protected model can use the compiled library from your protected model. The protected model includes the binary files and header files compiled from your generated source code. It does not include the source code. This option prevents the user of your model and potential attackers from accessing the model's generated source code.
- Password protection — Use this option to support some functionality for your protected model's user and protect the supporting files from external attackers. For each option that you specify a password, the derived files that support that option are protected by using AES-256 encryption. Before executing each functionality, the protected model prompts the user for a password. Coordinate with the user of your protected model to choose the password for each option. If your protected model is referenced by a parent model that will also be protected, do not password-protect functionalities that the parent protected model will support.
- Digital signature — Sign your protected model by using a certificate to protect the recipient from using the model if it has been changed by an external attacker. When you sign your model, the user of the model can verify that the protected model is published by you and did not change after you signed it. After you sign the model, if it is changed, a user cannot use it. Make sure that the user's computer storage of trusted certificates has the certificate from the certificate authority that certified your certificate.

Interface Report

When you create a protected model, the generated protected model report includes an interface report. The interface report includes information about input and output specifications, exported functions, interface parameters, and data stores. Use the interface report as a reference to communicate with the user of your protected model.

Create a Harness Model

When you create a protected model, you have the option to create a harness model that references your protected model. Use the harness model to define information that the user requires to interface your protected model with their system. The harness contains the same configuration set that your protected model uses. In the harness model, you can include data definitions that your protected model uses.

See Also

More About

- “Protect Models to Conceal Contents” on page 11-2

Stateflow Blocks in Simulink Coder

- “Code Generation of Stateflow Blocks” on page 12-2
- “Generate Code from Atomic Subcharts” on page 12-5
- “Inline State Functions in Generated Code” on page 12-7
- “Air-Fuel Ratio Control System with Stateflow Charts” on page 12-10

Code Generation of Stateflow Blocks

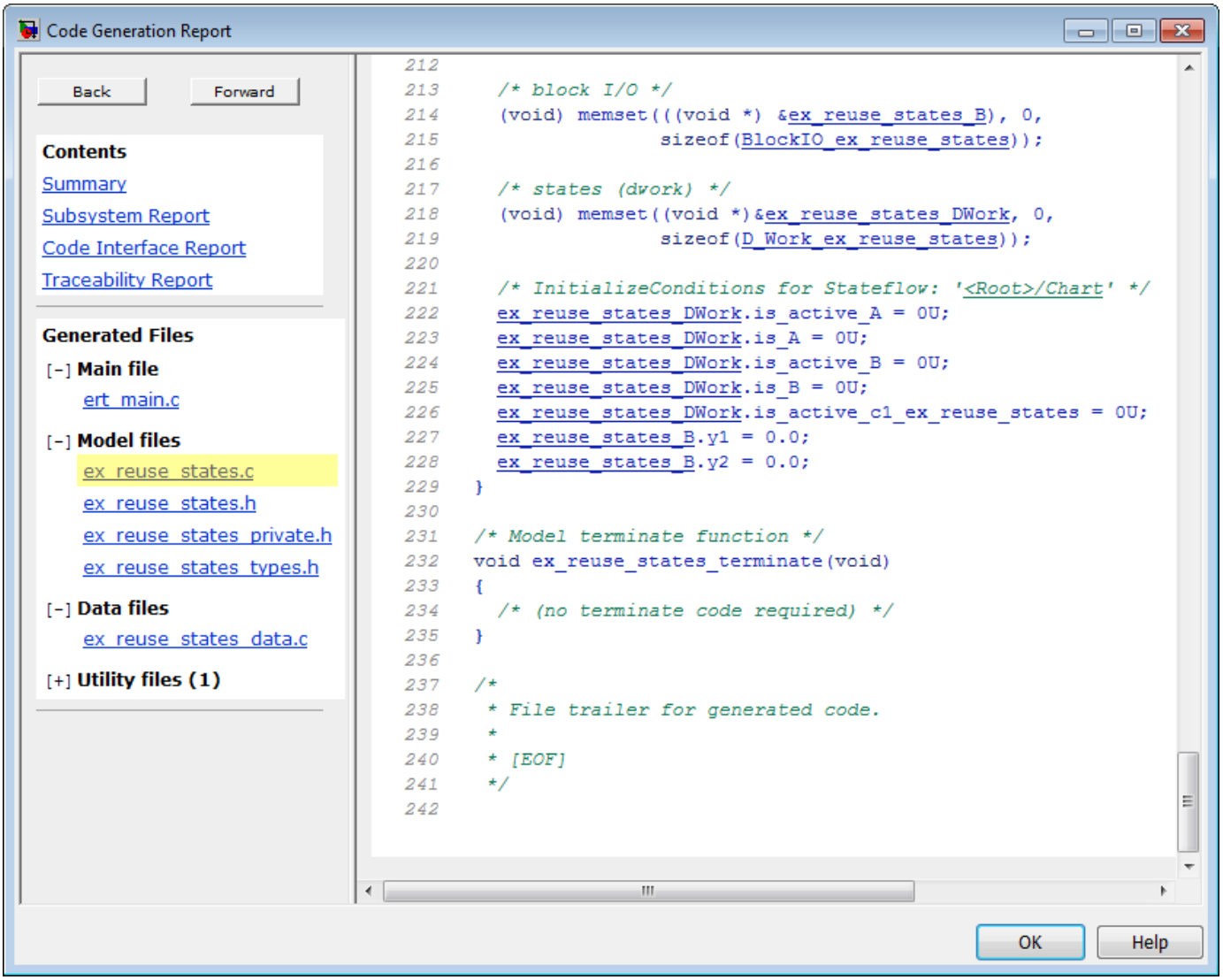
The code generator produces code for Stateflow blocks for rapid prototyping. If you have an Embedded Coder license, you can generate production code for Stateflow blocks.

Comparison of Code Generation Methods

The following sections compare two ways of generating code.

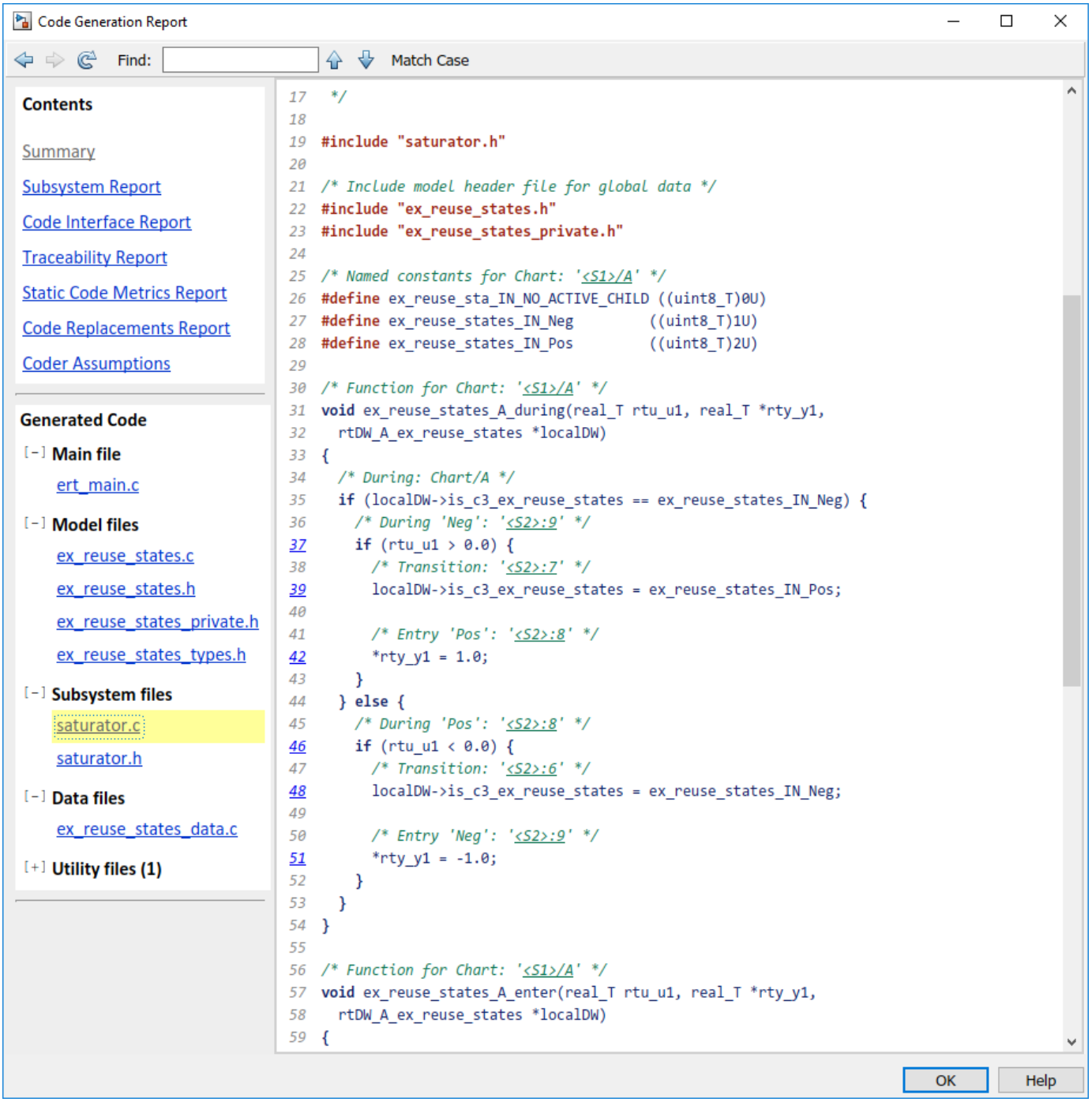
Code Generation Without Atomic Subcharts

You generate code for the entire model in one file and look through that entire file to find code for a specific part of the chart.



Code Generation With Atomic Subcharts

You specify code generation parameters so that code for an atomic subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You can avoid searching through unrelated code and focus only on the part that interests you.



Note Unreachable Stateflow states are optimized out and are not included in the generated code.

See Also

Related Examples

- “Generate Code from Atomic Subcharts” on page 12-5
- “Generate C or C++ Code from Stateflow Blocks” (Stateflow)

Generate Code from Atomic Subcharts

To unit test a Stateflow chart in a Simulink model, first break the chart into smaller, independent components by using atomic subcharts. When you generate code for your chart, a separate file stores the code for the atomic subchart. Generating reusable code from atomic subcharts is useful for testing individual parts of your Stateflow chart. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” (Stateflow).

Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

1. In your chart, right-click the atomic subchart and select **Properties**.
2. In the dialog box, specify these parameters:
   a. Set **Code generation function packaging** to `Reusable function`.
   b. Set **Code generation file name options** to `User specified`.
   c. For **Code generation file name**, enter the name of the file with no extension.
3. Open the Configuration Parameters dialog box.
4. In the **Code Generation** pane, set the **System target file** parameter to `ert.tlc`.
5. (OPTIONAL) Customize the generated function names for atomic subcharts. In the **Code Generation** > **Identifiers** pane, set the **Subsystem methods** parameter. Specify the format of the function names by using a combination of these tokens:
   - `$R` — root model name
   - `$F` — type of interface function for the atomic subchart
   - `$N` — block name
   - `$H` — subsystem index
   - `$M` — name-mangling text

For more information, see “Generate Separate Code for an Atomic Subchart” (Stateflow).

Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1. Open the library model that contains your atomic subchart.
2. Unlock the library.
3. Right-click the library chart and select **Block Parameters**.
4. In the dialog box, specify these parameters:
   a. On the **Main** tab, select **Treat as atomic unit**.
   b. On the **Code Generation** tab, set **Function packaging** to `Reusable function`.
   c. Set **File name options** to `User specified`.
   d. For **File name**, enter the name of the file with no extension.
5. In the **Code Generation** tab, set the **System target file** parameter to `ert.tlc`.

6 (OPTIONAL) Customize the generated function names for atomic subcharts. In the **Code Generation > Identifiers** tab, set the **Subsystem methods** parameter. Specify the format of the function names by using a combination of these tokens:

- `$R` — root model name
- `$F` — type of interface function for the atomic subchart
- `$N` — block name
- `$H` — subsystem index
- `$M` — name-mangling text

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

See Also

More About

- "Create Reusable Subcomponents by Using Atomic Subcharts" (Stateflow)
- "Generate C or C++ Code from Stateflow Blocks" (Stateflow)
- "Generate Separate Code for an Atomic Subchart" (Stateflow)

Inline State Functions in Generated Code

In this section...
“Inlined Generated Code for State Functions” on page 12-7 “How to Set the State Function Inline Option” on page 12-8 “Best Practices for Controlling State Function Inlining” on page 12-8

Inlined Generated Code for State Functions

By default, the code generator uses an internal heuristic to determine whether to inline generated code for state functions. The heuristic takes into consideration an inlining threshold. As code grows and shrinks in size, generated code for state functions can be unpredictable.

If your model includes Stateflow objects and you have rigorous requirements for traceability between generated code and the corresponding state functions, you can override the default behavior. Use the state property `Function Inline Option` to explicitly force or prevent inlining of state functions.

What Happens When You Force Inlining

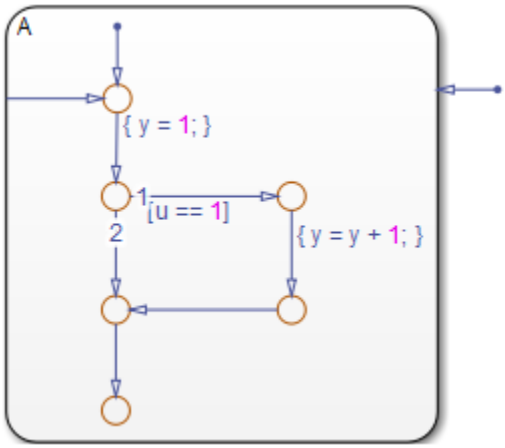
If you force inlining for a state, the code generator inlines code for state actions into the parent function. The parent function contains code for executing the state actions, outer transitions, and flow charts. It does not include code for empty state actions.

What Happens When You Prevent Inlining

If you prevent inlining for a state, the code generator produces these static functions for state *foo*.

Function	Description
`enter_atomic_`*`foo`*	Marks *`foo`* active and performs entry actions.
`enter_internal_`*`foo`*	Calls default paths.
`inner_default_`*`foo`*	Executes flow charts that originate when an inner transition and default transition reach the same junction inside a state. The code generator produces this function only when the flow chart is complex enough to exceed the inlining threshold. In generated code, Stateflow software calls this function from both the `enter_internal_`*`foo`* and *`foo`* functions.
`foo`	Checks for valid outer transitions and if none, performs during actions.
`exit_atomic_`*`foo`*	Performs exit actions and marks *`foo`* inactive.
`exit_internal_`*`foo`*	Performs exit actions of the child substates and then exits *`foo`*.

Suppose the following chart is in model `M`.



If you prevent inlining for state A, the code generator produces this code.

```
static void M_inner_default_A(void);
static void M_exit_atomic_A(void);
static void M_A(void);
static void M_enter_atomic_A(void);
static void M_enter_internal_A(void);
```

How to Set the State Function Inline Option

To set the function inlining property for a state:

1. Right-click inside the state and, from the context menu, select **Properties**.

   The State properties dialog box opens.

2. For property **Function Inline Option**, select one of these values.

Value	Behavior
`Inline`	Forces inlining of state functions into the parent function, as long as the function is not part of a recursion. See “What Happens When You Force Inlining” on page 12-7.
`Function`	Prevents inlining of state functions. Generates up to six static functions for the state. See “What Happens When You Prevent Inlining” on page 12-7.
`Auto`	Uses internal heuristics to determine whether or not to inline the state functions.

3. Click **Apply**.

Best Practices for Controlling State Function Inlining

To	Set Function Inline Option Property To
Generate a separate function for each action of a state and a separate function for each action of its substates	`Function` for the state and each substate

To	Set Function Inline Option Property To
Generate a separate function for each action of a state, but include code for the associated action of its substates	`Function` for the state and `Inline` for each substate

Air-Fuel Ratio Control System with Stateflow Charts

Generate code for an air-fuel ratio control system designed with Simulink® and Stateflow®.

Figures 1, 2, and 3 show relevant portions of the sldemo_fuelsys model, a closed-loop system containing a plant and controller. The plant validates the controller in simulation early in the design cycle. In this example, you generate code for the relevant controller subsystem, "fuel_rate_control". Figure 1 shows the top-level simulation model.

Open and configure the `sldemo_fuelsys` model. Then, compile the model to see the signal data types.

```
model = 'sldemo_fuelsys';
open_system(model)
rtwconfiguredemo(model,'ERT','float');
set_param(model,'ShowPortDataTypes','on');
set_param(model,'SampleTimeColors','on');
sldemo_fuelsys([],[],[],'compile');
sldemo_fuelsys([],[],[],'term');
```

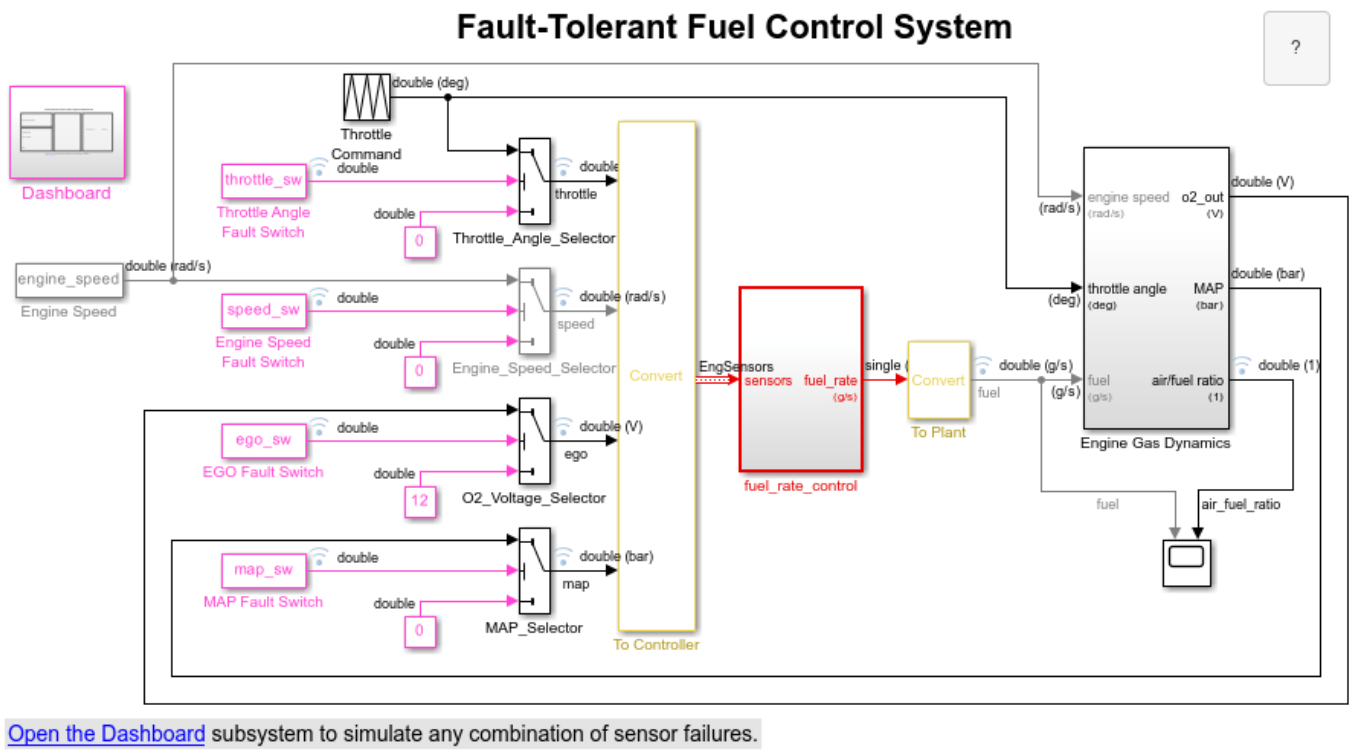


Figure 1: Top-level model of the plant and controller

The air-fuel ratio control system is comprised of Simulink and Stateflow. The control system is the portion of the model for which you generate code.

```
open_system('sldemo_fuelsys/fuel_rate_control');
```

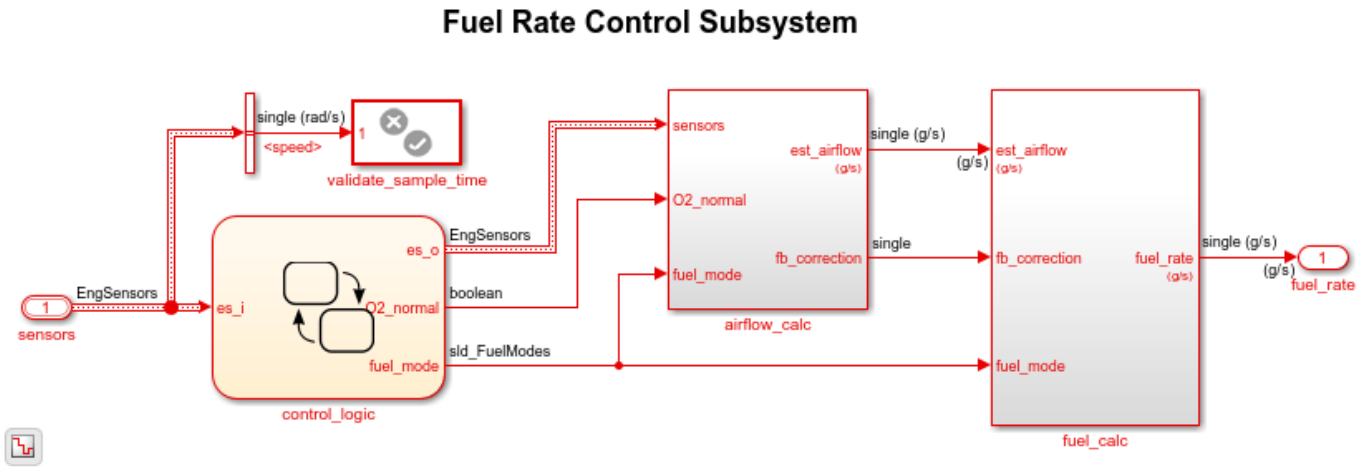


Figure 2: The air-fuel ratio controller subsystem

The control logic is a Stateflow chart that specifies the different modes of operation.

```
open_system('sldemo_fuelsys/fuel_rate_control/control_logic');
```

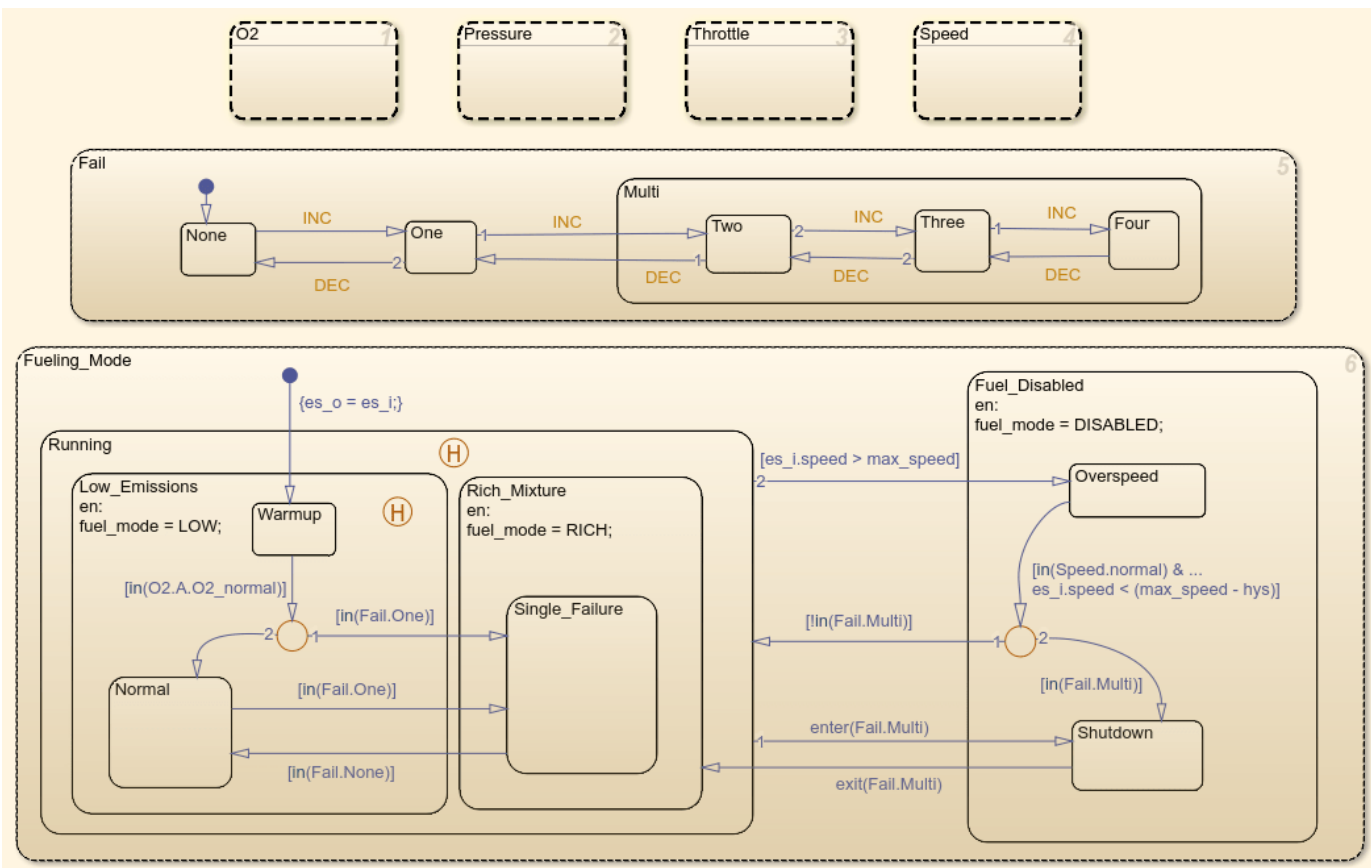
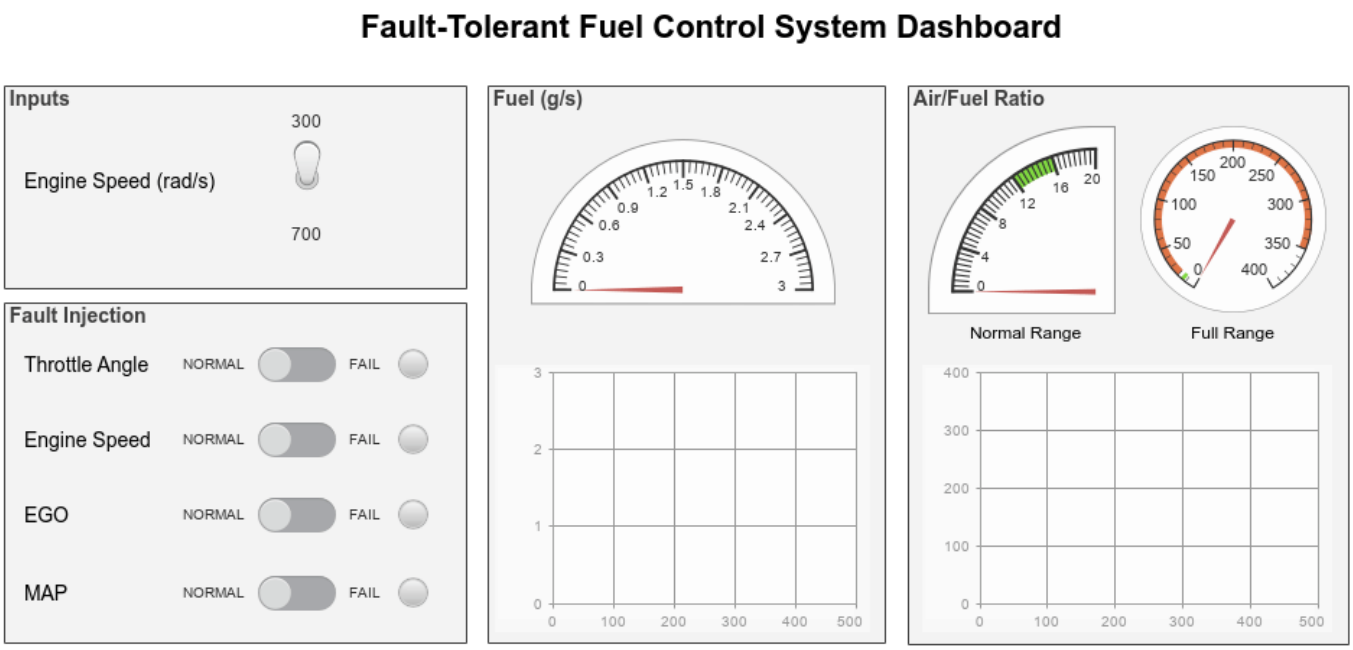


Figure 3: Air-fuel rate controller logic

Close these windows.

```
close_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
close_system('sldemo_fuelsys/fuel_rate_control/fuel_calc');
close_system('sldemo_fuelsys/fuel_rate_control/control_logic');
hDemo.rt=sfroot;hDemo.m=hDemo.rt.find('-isa','Simulink.BlockDiagram');
hDemo.c=hDemo.m.find('-isa','Stateflow.Chart','-and','Name','control_logic');
hDemo.c.visible=false;
close_system('sldemo_fuelsys/fuel_rate_control');
```



Update Diagram (Ctrl+D) to bind switches to model workspace variables.

Configure and Build the Model with Embedded Coder

To configure and build production ANSI® C/C++ code for the model, set model configuration parameter **System target file** to `ert.tlc` (Embedded Real-Time (ERT)). You can set the **System target file** parameter programmatically.

```
rtwconfiguredemo('sldemo_fuelsys','ERT');
```

Generate and inspect the code. You can navigate to the relevant code segments interactively by using **Previous** and **Next** buttons. From the chart context menu (right-click the Stateflow block), select **C/C++ Code > Navigate to C/C++ Code**. Programmatically, use the `rtwtrace` utility.

```
slbuild('sldemo_fuelsys/fuel_rate_control');
rtwtrace('sldemo_fuelsys/fuel_rate_control/control_logic')
```

```
### Starting build procedure for: fuel_rate_control
### Successful completion of build procedure for: fuel_rate_control

Build Summary

Top model targets built:
```

```
Model              Action                        Rebuild Reason
==========================================================================================
fuel_rate_control  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 23.173s

ans =

     []
```

View the air-fuel ratio control logic in the generated code.

```
rtwdemodbtype('fuel_rate_control_ert_rtw/fuel_rate_control.c',...
    '/* Function for Chart:','case IN_Warmup:',1,0);
```

```
/* Function for Chart: '<S1>/control_logic' */
static void Fueling_Mode(const int32_T *sfEvent)
{
  switch (rtDWork.bitsForTID0.is_Fueling_Mode) {
   case IN_Fuel_Disabled:
    rtDWork.fuel_mode = DISABLED;
    switch (rtDWork.bitsForTID0.is_Fuel_Disabled) {
     case IN_Overspeed:
      /* Inport: '<Root>/sensors' */
      if ((rtDWork.bitsForTID0.is_Speed == IN_normal) && (rtU.sensors.speed <
           603.0F)) {
        if (rtDWork.bitsForTID0.is_Fail != IN_Multi) {
          rtDWork.bitsForTID0.is_Fuel_Disabled = IN_NO_ACTIVE_CHILD;
          rtDWork.bitsForTID0.is_Fueling_Mode = IN_Running;

          /* The fuel is actively controlled while in this state. */
          switch (rtDWork.bitsForTID0.was_Running) {
           case IN_Low_Emissions:
            rtDWork.bitsForTID0.is_Running = IN_Low_Emissions;
            rtDWork.bitsForTID0.was_Running = IN_Low_Emissions;
            rtDWork.fuel_mode = LOW;
            switch (rtDWork.bitsForTID0.was_Low_Emissions) {
             case IN_Normal:
              rtDWork.bitsForTID0.is_Low_Emissions = IN_Normal;
              rtDWork.bitsForTID0.was_Low_Emissions = IN_Normal;

              /* All sensors are in correct operating modes, so effective closed-loop mixture co
              break;
```

Close the model and code generation report.

```
clear hDemo;
rtwdemoclean;
close_system('sldemo_fuelsys',0);
```

Block Authoring and Code Generation for Simulink Coder

- “S-Functions and Code Generation” on page 13-2
- “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6
- “Legacy Code Tool Examples” on page 13-11
- “Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type” on page 13-47
- “Integrate External C Functions That Implement N-Dimensional Table Lookups” on page 13-50
- “Integrate External C++ Object Methods” on page 13-53
- “External Code Integration Examples” on page 13-56
- “S-Functions That Support Expression Folding” on page 13-64
- “S-Functions That Specify Port Scope and Reusability” on page 13-73
- “S-Functions That Specify Sample Time Inheritance Rules” on page 13-77
- “S-Functions for Code Reuse” on page 13-81
- “S-Functions for Multirate Multitasking Environments” on page 13-86
- “Write Noninlined S-Function” on page 13-91
- “Write Wrapper S-Function and TLC Files” on page 13-93
- “Write Fully Inlined S-Functions” on page 13-100
- “Write Fully Inlined S-Functions with mdlRTW Routine” on page 13-102

S-Functions and Code Generation

You use S-functions to extend Simulink support for simulation and code generation. For example, you can use them to:

- Represent custom algorithms
- Interface existing external code with Simulink model and the generated code
- Represent device drivers for interfacing with hardware
- Generate highly optimized code for embedded systems
- Verify code generated for a subsystem as part of a Simulink simulation

The application program interface (API) for writing S-functions allows you to implement generic algorithms in the Simulink environment with a great deal of flexibility. If you intend to use S-functions in a model for code generation, the level of flexibility can vary. For example, it is not possible to access the MATLAB workspace from an S-function that you use with the code generator. This topic explains conditions to be aware of for using S-functions. However, using the techniques presented in this topic, you can create S-functions for most applications that work with the generated code.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in a model, the underlying API incurs overhead in terms of memory and computation resources. Most often the additional resources are acceptable for real-time rapid prototyping systems. Often, though, additional resources are unavailable in real-time embedded applications. You can minimize memory and computational requirements by using the Target Language Compiler technology provided with the code generator to inline your S-functions. If you are producing an S-function for existing external code, consider using the Legacy Code Tool to generate your S-function and relevant TLC file.

This topic assumes that you understand the following concepts:

- Level-2 S-functions
- Target Language Compiler (TLC) scripting
- How the code generator produces and builds C/C++ code

Notes This information is for code generator users. Even if you do not currently use the code generator, follow these practices when writing S-functions, especially if you are creating general-purpose S-functions.

Types of S-Functions

Examples for which you might choose to implement an S-function for simulation and code generation include:

1. “I'm not concerned with efficiency. I want to write one version of my algorithm and have it work in the Simulink and code generator products automatically.”
2. “I want to implement a highly optimized algorithm in the Simulink and code generator products that looks like a built-in block and generates efficient code.”
3. “I have a lot of hand-written code that I need to interface. I want to call my function from the Simulink and code generator products efficiently.”

Respectively, the preceding situations map to the following MathWorks terminology:

1. Noninlined S-function
2. Inlined S-function
3. Auto-generated S-function for external code

Noninlined S-Functions

A noninlined S-function is a C or C++ MEX S-function that is treated identically by the Simulink engine and generated code. In general, you implement your algorithm once according to the S-function API. The Simulink engine and generated code call the S-function routines (for example, `mdlOutputs`) during model execution.

Additional memory and computation resources are required for each instance of a noninlined S-Function block. However, this routine of incorporating algorithms into models and code generation applications is typical during the prototyping phase of a project where efficiency is not important. The advantage gained by forgoing efficiency is the ability to change model parameters and structures rapidly.

Writing a noninlined S-function does not involve TLC coding. Noninlined S-functions are the default case for the build process in the sense that once you build a MEX S-function in your model, there is no additional preparation before pressing **Ctrl+B** to build your model.

Some restrictions exist concerning the names and locations of noninlined S-function files when generating makefiles. See “Write Noninlined S-Function” on page 13-91.

Inlined S-Functions

For S-functions to work in the Simulink environment, some overhead code is generated. When the code generator produces code from models that contain S-functions (without *`sfunction`*`.tlc` files), it embeds some of this overhead code in the generated code. If you want to optimize your real-time code and eliminate some of the overhead code, you must *inline* (or embed) your S-functions. This involves writing a TLC (*`sfunction`*`.tlc`) file that eliminates overhead code from the generated code. The Target Language Compiler processes *`sfunction`*`.tlc` files to define how to inline your S-function algorithm in the generated code.

Note Do not confuse the term *inline* with the C++ *inline* keyword. Inline means to specify text in place of the call to the general S-function API routines (for example, `mdlOutputs`). For example, when a TLC file is used to inline an S-function, the generated code contains the C/ C++ code that normally appears within the S-function routines and the S-function itself that has been removed from the build process.

A fully inlined S-function builds your algorithm (block) into generated code in a manner that is indistinguishable from a built-in block. Typically, a fully inlined S-function requires you to implement your algorithm twice: once for the Simulink model (C/C++ MEX S-function) and once for code generation (TLC file). The complexity of the TLC file depends on the complexity of your algorithm and the level of efficiency you try to achieve in the generated code. TLC files vary from simple to complex in structure. See “Inlining S-Functions”.

Auto-generated S-Functions for Legacy or Custom Code

If you need to invoke hand-written C/C++ code in your model, consider using the Simulink Legacy Code Tool. The Legacy Code Tool can automate the generation of a fully inlined S-function and a corresponding TLC file based on information that you register in a Legacy Code Tool data structure.

For more information, see “Integrate C Functions Using Legacy Code Tool” and see “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6.

Files Required for Implementing Noninlined and Inlined S-Functions

This topic briefly describes what files and functions you need to create noninlined and inlined S-functions.

- Noninlined S-functions require the C or C++ MEX S-function source code (*sfunction*.c or *sfunction*.cpp).
- Fully inlined S-functions require an *sfunction*.tlc file. Fully inlined S-functions produce the optimal code for a parameterized S-function. This is an S-function that operates in a specific mode dependent upon fixed S-function parameters that do not change during model execution. For a given operating mode, the *sfunction*.tlc file specifies the exact code that is generated to implement the algorithm for that mode. For example, the direct-index lookup table S-function in “Write Fully Inlined S-Functions with mdlRTW Routine” on page 13-102 contains two operating modes — one for evenly spaced x-data and one for unevenly spaced x-data.

> **Note** Fully inlined S-functions that are generated to invoke legacy or custom C/C++ code also require an *sfunction*.tlc file, which is generated by Legacy Code Tool.

Fully inlined S-functions might require the placement of the mdlRTW routine in your S-function MEX-file *sfunction*.c or *sfunction*.cpp. The mdlRTW routine lets you place information in *model*.rtw. *model*.rtw is the record file that specifies a model, and which the code generator invokes the Target Language Compiler to process before executing *sfunction*.tlc when generating code.

Including a mdlRTW routine is useful when you want to introduce non-tunable parameters into your TLC file. Such parameters are used to determine which operating mode is active in a given instance of the S-function. Based on this information, the TLC file for the S-function can generate highly efficient, optimal code for that operating mode.

Guidelines for Writing S-Functions that Support Code Generation

- You can use C/C++ MEX, MATLAB language, and Fortran MEX S-functions with code generation.
- You can inline S-functions for code generation by providing an inlining TLC file. See S-Function Inlining in “Target Language Compiler”. MATLAB and Fortran MEX S-functions must be inlined. C/C++ MEX S-functions can be inlined for code efficiency, or noninlined.
- To automatically generate a fully inlined C MEX S-function for invoking legacy or custom code, use the Legacy Code Tool. For more information, see “Integrate C Functions Using Legacy Code Tool” and see “Import Calls to External Code into Generated Code with Legacy Code Tool”.
- If code efficiency is not an overriding consideration, for example, if you are rapid prototyping, you can choose not to inline a C/C++ MEX S-function. For more information, see “Write Noninlined S-Function” on page 13-91.

See Also

More About

- “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6

Import Calls to External Code into Generated Code with Legacy Code Tool

In this section...

"Legacy Code Tool and Code Generation" on page 13-6
"Generate Inlined S-Function Files for Code Generation" on page 13-6
"Apply Code Style Settings to Legacy Functions" on page 13-7
"Address Dependencies on Files in Different Locations" on page 13-8
"Deploy S-Functions for Simulation and Code Generation" on page 13-8
"Integrate External C++ Objects" on page 13-8

Legacy Code Tool and Code Generation

You can use the Simulink Legacy Code Tool to generate fully inlined C MEX S-functions for legacy or custom code. The S-functions are optimized for embedded components, such as device drivers and lookup tables, and they call existing C or C++ functions.

Note The Legacy Code Tool can interface with C++ functions, but not C++ objects. To work around this issue so that the tool can interface with C++ objects, see "Legacy Code Tool Limitations".

You can use the tool to:

- Compile and build the generated S-function for simulation.
- Generate a masked S-Function block that is configured to call the existing external code.

If you want to include these types of S-functions in models for which you intend to generate code, use the tool to generate a TLC block file. The TLC block file specifies how the generated code for a model calls the existing C or C++ function.

If the S-function depends on files in folders other than the folder containing the S-function dynamically loadable executable file, use the tool to generate an `sFunction_makecfg.m` or `rtwmakecfg.m` file for the S-function. Generating the file maintains those dependencies when you build a model that includes the S-function. For example, for some applications, such as custom targets, you might want to locate files in a target-specific location. The build process looks for `sFunction_makecfg.m` or `rtwmakecfg.m` in the same folder as the S-function dynamically loadable executable and calls the function in the file.

For more information, see "Integrate C Functions Using Legacy Code Tool".

Generate Inlined S-Function Files for Code Generation

Depending on the code generation requirements of your application, to generate code for a model that uses the S-function, do either of the following:

- Generate one `.cpp` file for the inlined S-function. In the Legacy Code Tool data structure, set the value of the `Options.singleCPPMexFile` field to `true` before generating the S-function source file from your existing C function. For example:

```
def.Options.singleCPPMexFile = true;
legacy_code('sfcn_cmex_generate', def);
```

- Generate a source file and a TLC block file for the inlined S-function. For example:

```
def.Options.singleCPPMexFile = false;
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
  - `Simulink.Bus`
  - `Simulink.AliasType`
  - `Simulink.NumericType`
- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
- `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Apply Code Style Settings to Legacy Functions

To apply the model configuration parameters for code style to a legacy function:

1. Initialize the Legacy Code Tool data structure. For example:

```
def = legacy_code('initialize');
```

2. In the data structure, set the value of the `Options.singleCPPMexFile` field to `true`. For example:

```
def.Options.singleCPPMexFile = true;
```

To check the setting, enter:

```
def.Options.singleCPPMexFile
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
  - `Simulink.Bus`
  - `Simulink.AliasType`
  - `Simulink.NumericType`
- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
- `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Address Dependencies on Files in Different Locations

By default, the Legacy Code Tool assumes that files on which an S-function depends reside in the same folder as the dynamically loadable executable file for the S-function. If your S-function depends on files that reside elsewhere and you are using the template makefile build process, generate an *sFunction*_makecfg.m or rtwmakecfg.m file for the S-function. For example, you might generate this file if your Legacy Code Tool data structure defines compilation resources as path names.

To generate the *sFunction*_makecfg.m or rtwmakecfg.m file, call the legacy_code function with 'sfcn_makecfg_generate' or 'rtwmakecfg_generate' as the first argument, and the name of the Legacy Code Tool data structure as the second argument. For example:

```
legacy_code('sfcn_makecfg_generate', lct_spec);
```

If you use multiple registration files in the same folder and generate an S-function for each file with a single call to legacy_code, the call to legacy_code that specifies 'sfcn_makecfg_generate' or 'rtwmakecfg_generate' must be common to all registration files. For more information, see “Handling Multiple Registration Files”.

For example, if you define defs as an array of Legacy Code Tool structures, you call legacy_code with 'sfcn_makecfg_generate' once.

```
defs = [defs1(:);defs2(:);defs3(:)];
legacy_code('sfcn_makecfg_generate', defs);
```

For more information, see “Build Support for S-Functions”.

Deploy S-Functions for Simulation and Code Generation

You can deploy the S-functions that you generate with the Legacy Code Tool so that other people can use them. To deploy an S-function for simulation and code generation, share the following files:

- Registration file
- Compiled dynamically loadable executable
- TLC block file
- *sFunction*_makecfg.m or rtwmakecfg.m file
- Header, source, and include files on which the generated S-function depends

When you use these deployed files:

- Before using the deployed files in a Simulink model, add the folder that contains the S-function files to the MATLAB path.
- If the Legacy Code Tool data structure registers required files as absolute paths and the location of the files changes, regenerate the *sFunction*_makecfg.m or rtwmakecfg.m file.

Integrate External C++ Objects

The Legacy Code Tool can interface with C++ functions, but not C++ objects. Using the previous example as a starting point, here is an example of how you can work around this limitation.

- Modify the class definition for adder in a new file adder_cpp.hpp. Add three new macros that dynamically allocate a new adder object, invoke the method add_one(), and free the memory

allocated. Each macro takes a pointer to an `adder` object. Because each function called by the Legacy Code Tool must have a C-like signature, the pointer is cached and passed as a `void*`. Then you must explicitly cast to `adder*` in the macro. The new class definition for `adder`:

```
#ifndef _ADDER_CPP_
#define _ADDER_CPP_

class adder {
private:
    int int_state;
public:
    adder(): int_state(0) {};
    int add_one(int increment);
    int get_val() {return int_state;};
};

// Method wrappers implemented as macros
#define createAdder(work1) \
    *(work1) = new adder

#define deleteAdder(work1) \
    delete(static_cast<adder*>(*(work1)))

#define adderOutput(work1, u1) \
    (static_cast<adder*> ((work1)))->add_one(u1)

#endif /* _ADDER_CPP_ */
```

- Update `adder_cpp.cpp`. With the class modification, instead of one global instance, each generated S-function manages its own `adder` object.

```
#include "adder_cpp.hpp"

int adder::add_one(int increment)
{
    int_state += increment;
    return int_state;
}
```

- Update `rtwdemo_sfun_adder_cpp.cpp` with the following changes:
  - `StartFcnSpec` calls the macro that allocates a new `adder` object and caches the pointer.

    ```
    def.StartFcnSpec  = 'createAdder(void **work1)';
    ```

  - `OutputFcnSpec` calls the macro that invokes the method `add_one()` and provides the S-function specific `adder` pointer object.

    ```
    def.OutputFcnSpec = 'int32 y1 = adderOutput(void *work1, int32 u1)';
    ```

  - `TerminateFcnSpec` calls the macro that frees the memory.

    ```
    def.TerminateFcnSpec = 'deleteAdder(void **work1)';
    ```

See Also

`legacy_code`

Related Examples

- “Integrate C Functions Using Legacy Code Tool”
- “Legacy Code Tool Examples”
- “Call External C Code from Model and Generated Code” on page 53-48

Legacy Code Tool Examples

You can use the Simulink Legacy Code Tool to generate fully inlined C MEX S-functions for legacy or custom code. The S-functions are optimized for embedded components, such as device drivers and lookup tables, and they call existing C or C++ functions. These examples show how to use the Legacy Code Tool to integrate a variety of external functions.

Integrate External C Functions That Pass Input Arguments by Value Versus Address

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their input arguments by value versus address.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototypes of the legacy functions being called in this example are:

- FLT filterV1(const FLT signal, const FLT prevSignal, const FLT gain)
- FLT filterV2(const FLT* signal, const FLT prevSignal, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, myfilter.h, filterV1.c, and filterV2.c.

Note the difference in the OutputFcnSpec defined in the two structures; the first case specifies that the first input argument is passed by value, while the second case specifies pass by pointer.

```
defs = [];

% rtwdemo_sfun_filterV1
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_filterV1';
def.OutputFcnSpec = 'single y1 = filterV1(single u1, single u2, single p1)';
def.HeaderFiles   = {'myfilter.h'};
def.SourceFiles   = {'filterV1.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_filterV2
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_filterV2';
```

```
def.OutputFcnSpec = 'single y1 = filterV2(single u1[1], single u2, single p1)';
def.HeaderFiles   = {'myfilter.h'};
def.SourceFiles   = {'filterV2.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions in simulation. The source code for the S-functions is in the files rtwdemo_sfun_filterV1.c and rtwdemo_sfun_filterV2.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_filterV1
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_filterV1.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV1
### Exit

### Start Compiling rtwdemo_sfun_filterV2
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_filterV2.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV2
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_filterV1.tlc and rtwdemo_sfun_filterV2.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the

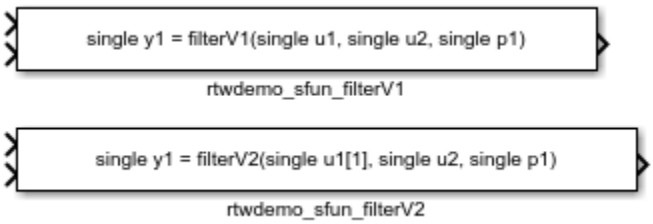
required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

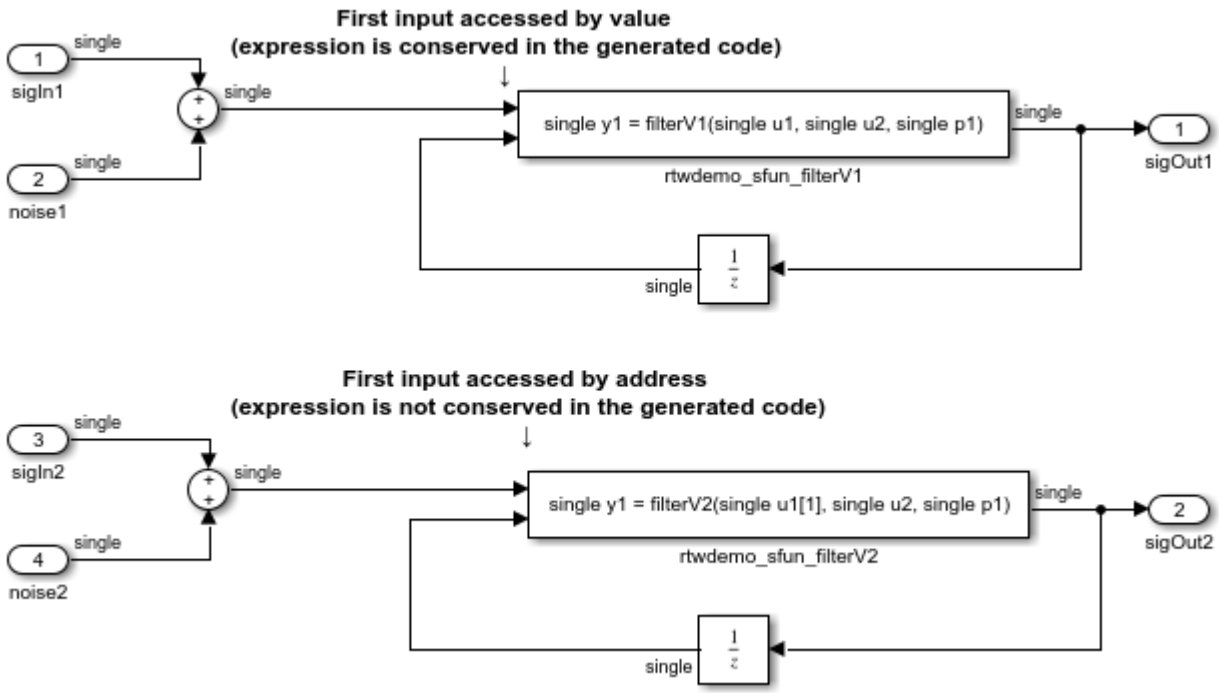
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_filter shows integration of the model with the legacy code. The subsystem TestFilter serves as a harness for the calls to the legacy C functions via the generate S-functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_filter')
open_system('rtwdemo_lct_filter/TestFilter')
sim('rtwdemo_lct_filter')
```



Integrate External C Functions That Pass the Output Argument as a Return Argument

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their output as a return argument.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

FLT gainScalar(const FLT in, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, gain.h, and gainScalar.c.

```
% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
```

```
def.SFunctionName = 'rtwdemo_sfun_gain_scalar';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_gain_scalar.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_gain_scalar
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_gain_scalar.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_gain_scalar
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_gain_scalar.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

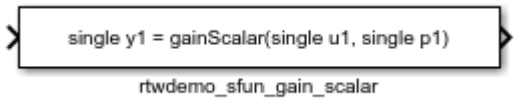
```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-

function. The software places the block in a new model. From there you can copy it to an existing model.

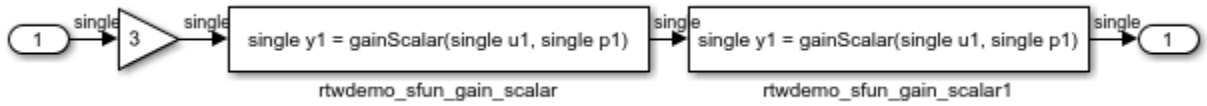
```
legacy_code('slblock_generate', def);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_gain shows integration of the model with the legacy code. The subsystem TestGain serves as a harness for the call to the legacy C function via the generate S-function.

```
open_system('rtwdemo_lct_gain')
open_system('rtwdemo_lct_gain/TestGain')
sim('rtwdemo_lct_gain')
```



Integrate External C++ Object Methods

Integrate legacy C++ object methods by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C++ MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a block TLC file and optional rtwmakecfg.m file that calls the legacy code during code generation.

Provide the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The definition of the legacy C++ class in this example is:

```
class adder {
  private:
      int int_state;
  public:
      adder();
      int add_one(int increment);
      int get_val();
};
```

The legacy source code is in the files adder_cpp.h and adder_cpp.cpp.

```
% rtwdemo_sfun_adder_cpp
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_adder_cpp';
def.StartFcnSpec  = 'createAdder()';
def.OutputFcnSpec = 'int32 y1 = adderOutput(int32 u1)';
def.TerminateFcnSpec = 'deleteAdder()';
def.HeaderFiles   = {'adder_cpp.h'};
def.SourceFiles   = {'adder_cpp.cpp'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.Options.language = 'C++';
def.Options.useTlcWithAccel = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_adder_cpp.cpp.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_adder_cpp
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
    mex('rtwdemo_sfun_adder_cpp.cpp', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\T
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_adder_cpp
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file that supports code generation through Simulink® Coder™. If the TLC block file is not created and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_adder_cpp.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file that supports code generation through Simulink® Coder™. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

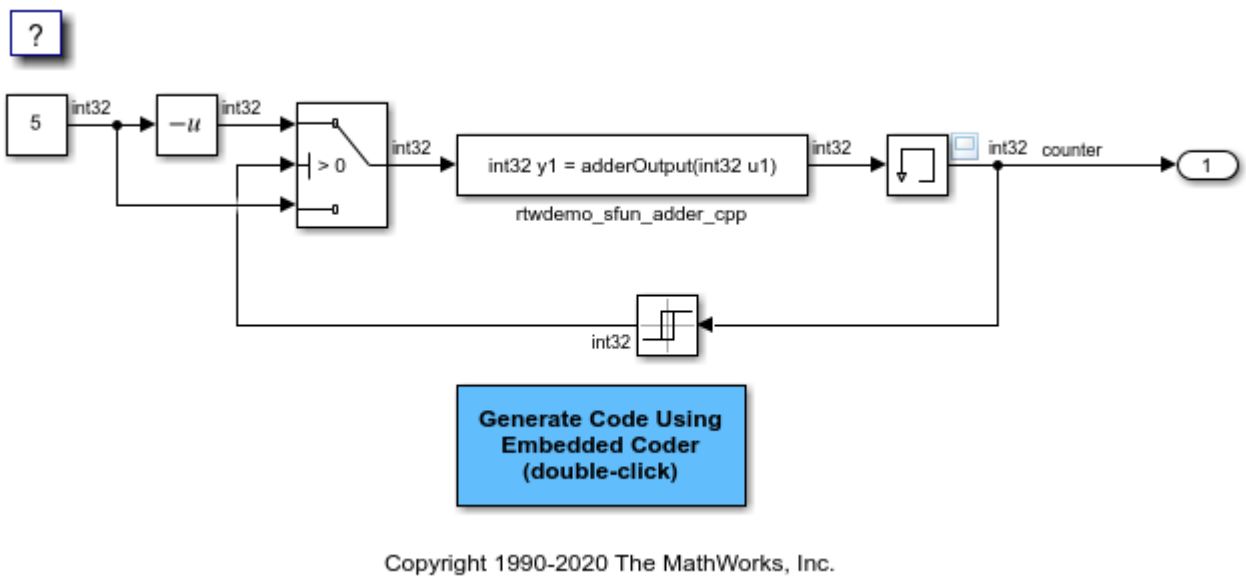
After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate a masked S-function block that is configured to call that S-function. The software places the block in a new model. You can copy the block to an existing model.

```
% legacy_code('slblock_generate', def);
```

Show the Generated Integration with Legacy Code

The model rtwdemo_lct_cpp shows integration with the legacy code.

```
open_system('rtwdemo_lct_cpp')
sim('rtwdemo_lct_cpp')
```



Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type

Integrate legacy C functions that pass their inputs and outputs by using parameters of a fixed-point data type with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)

myFixpt is logically a fixed point data type, which is physically a typedef to a 16-bit integer:

```
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is in the files timesFixpt.h, and timesS16.c.

```
% rtwdemo_sfun_gain_fixpt
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_gain_fixpt';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt p1, uint8 p2)';
def.HeaderFiles   = {'timesFixpt.h'};
def.SourceFiles   = {'timesS16.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_gain_fixpt.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_gain_fixpt
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_gain_fixpt.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TE
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_gain_fixpt
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_gain_fixpt.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

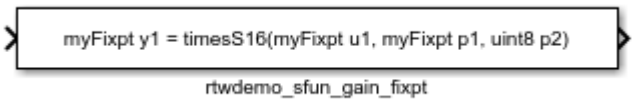
After you create the TLC block file, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file that supports code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. You can copy the block to an existing model.

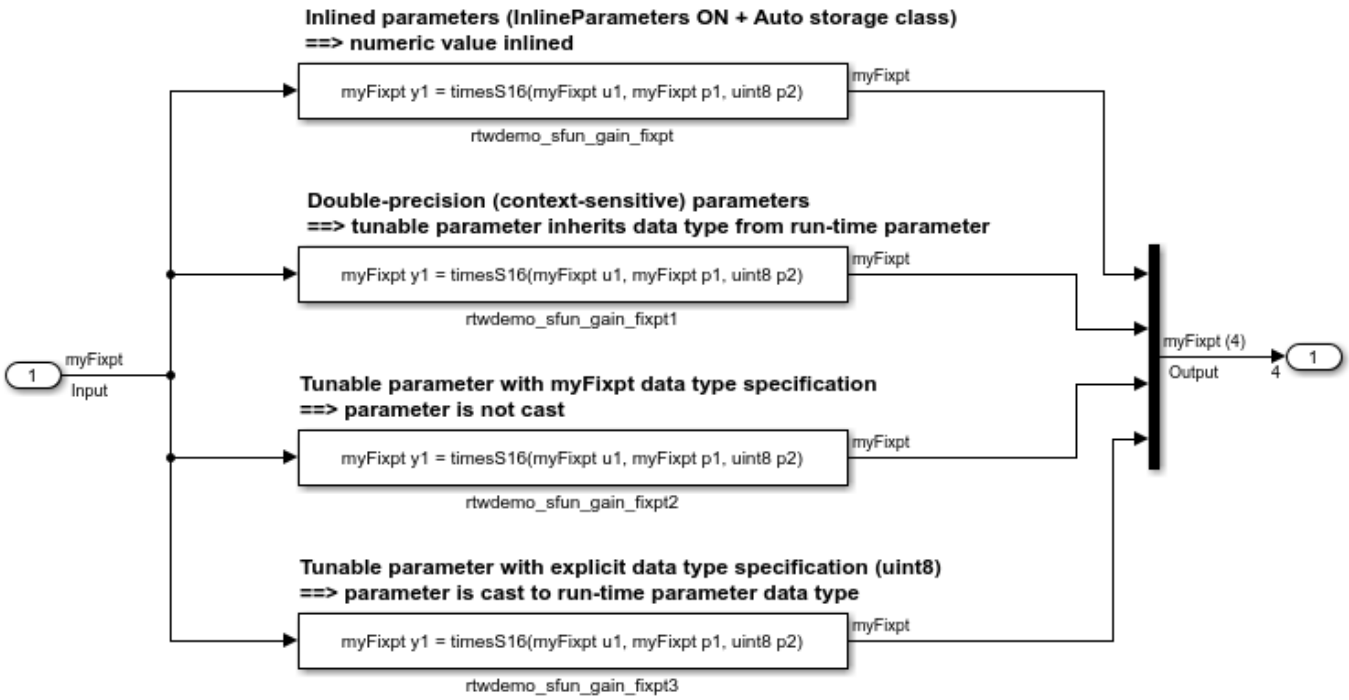
```
legacy_code('slblock_generate', def);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_fixpt_params shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function.

```
open_system('rtwdemo_lct_fixpt_params')
open_system('rtwdemo_lct_fixpt_params/TestFixpt')
sim('rtwdemo_lct_fixpt_params')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their inputs and outputs by using parameters of fixed-point data type.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)

myFixpt is logically a fixed-point data type, which is physically a typedef to a 16-bit integer:

```
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
```

```
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is in the files timesFixpt.h, and timesS16.c.

```
% rtwdemo_sfun_times_s16
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_times_s16';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt u2, uint8 p1)';
def.HeaderFiles   = {'timesFixpt.h'};
def.SourceFiles   = {'timesS16.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_times_s16.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_times_s16
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_times_s16.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TE
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_times_s16
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_times_s16.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the

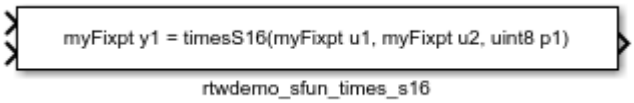
required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

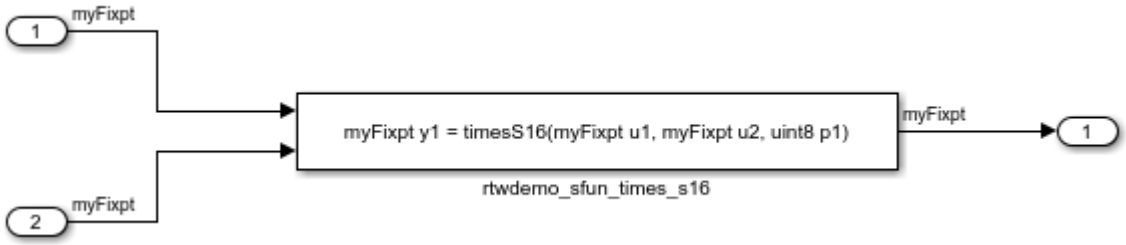
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_fixpt_signals shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function.

```
open_system('rtwdemo_lct_fixpt_signals')
open_system('rtwdemo_lct_fixpt_signals/TestFixpt')
sim('rtwdemo_lct_fixpt_signals')
```



Integrate External C Functions with Instance-Specific Persistent Memory

Integrate legacy C functions that use instance-specific persistent memory by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.

- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototypes of the legacy functions being called in this example are:

void memory_bus_init(COUNTERBUS *mem, int32_T upper_sat, int32_T lower_sat);

void memory_bus_step(COUNTERBUS *input, COUNTERBUS *mem, COUNTERBUS *output);

mem is an instance-specific persistent memory for applying a one integration step delay. COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files memory_bus.h, and memory_bus.c.

```
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_work
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_work';
def.InitializeConditionsFcnSpec = ...
    'void memory_bus_init(COUNTERBUS work1[1], int32 p1, int32 p2)';
def.OutputFcnSpec = ...
    'void memory_bus_step(COUNTERBUS u1[1], COUNTERBUS work1[1], COUNTERBUS y1[1])';
def.HeaderFiles   = {'memory_bus.h'};
def.SourceFiles   = {'memory_bus.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_work.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_work
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_work.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdo
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

```
### Finish Compiling rtwdemo_sfun_work
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_work.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

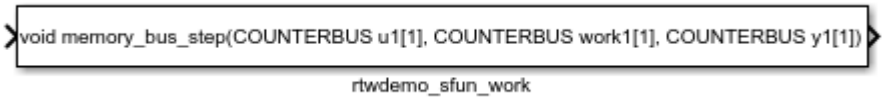
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

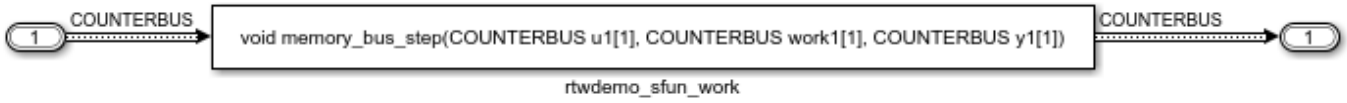
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_work shows integration of the model with the legacy code. The subsystem memory_bus serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_work')
open_system('rtwdemo_lct_work/memory_bus')
sim('rtwdemo_lct_work')
```



Integrate External C Functions That Use Structure Arguments

Integrate legacy C functions with structure arguments that use Simulink® buses with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)

COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files counterbus.h, and counterbus.c.

```
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_counterbus
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_counterbus';
def.OutputFcnSpec = ...
    'void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1])';
def.HeaderFiles   = {'counterbus.h'};
def.SourceFiles   = {'counterbus.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_counterbus.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_counterbus
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

```
    mex('rtwdemo_sfun_counterbus.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TE
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_counterbus
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_counterbus.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

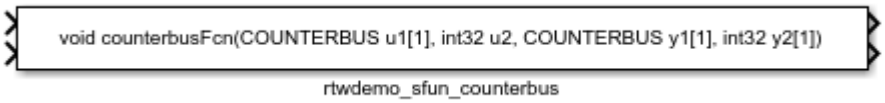
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

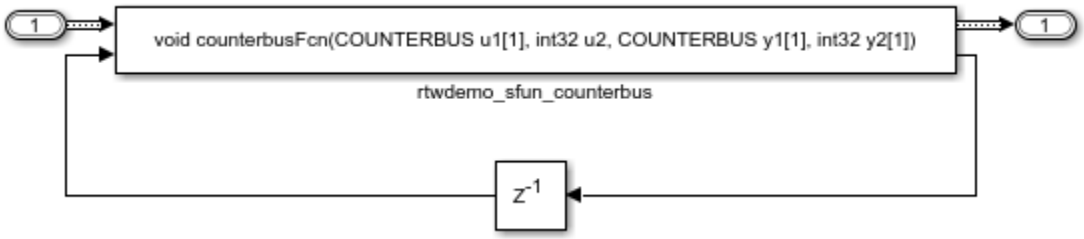
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_bus shows integration of the model with the legacy code. The subsystem TestCounter serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_bus')
open_system('rtwdemo_lct_bus/TestCounter')
sim('rtwdemo_lct_bus')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data

Integrate legacy C functions using complex signals with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

void cplx_gain(creal_T *input, creal_T *gain, creal_T *output);

creal_T is the complex representation of a double. The legacy source code is in the files cplxgain.h, and cplxgain.c.

```
% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_cplx_gain';
def.OutputFcnSpec = ...
    ['void cplx_gain(complex<double> u1[1], '...
    'complex<double> p1[1], complex<double> y1[1])'];
def.HeaderFiles   = {'cplxgain.h'};
def.SourceFiles   = {'cplxgain.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_cplx_gain.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_cplx_gain
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_cplx_gain.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TE
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_cplx_gain
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_cplx_gain.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

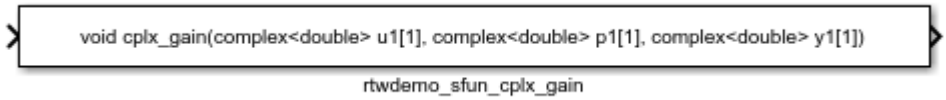
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

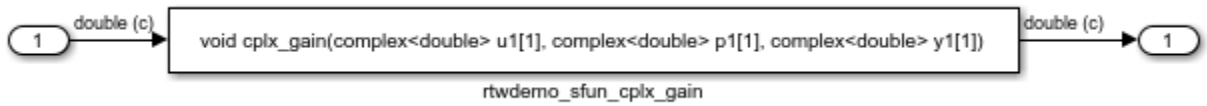
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_cplxgain shows integration of the model with the legacy code. The subsystem complex_gain serves as a harness for the call to the legacy C function via the generate S-function.

```
if isempty(find_system('SearchDepth',0,'Name','rtwdemo_lct_cplxgain'))
    open_system('rtwdemo_lct_cplxgain')
    open_system('rtwdemo_lct_cplxgain/complex_gain')
    sim('rtwdemo_lct_cplxgain')
end
```



Integrate External C Functions That Pass Arguments That Have Inherited Dimensions

This example shows how to use the Legacy Code Tool to integrate legacy C functions whose arguments have inherited dimensions.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototypes of the legacy functions being called in this example are:

- void mat_add(real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, real_T *y1)
- void mat_mult(real_T *u1, real_T *u2, int32_T nbRows1, int32_T nbCols1, int32_T nbCols2, real_T *y1)

real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is in the files mat_ops.h, and mat_ops.c.

```
defs = [];

% rtwdemo_sfun_mat_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_add';
def.OutputFcnSpec = ['void mat_add(double u1[][], double u2[][], ' ...
                     'int32 u3, int32 u4, double y1[size(u1,1)][size(u1,2)])'];
def.HeaderFiles   = {'mat_ops.h'};
```

```
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_mat_mult
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_mult';
def.OutputFcnSpec =  ['void mat_mult(double u1[p1][p2], double u2[p2][p3], '...
                    'int32 p1, int32 p2, int32 p3, double y1[p1][p3])'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-function is in the files rtwdemo_sfun_mat_add.c and rtwdemo_sfun_mat_mult.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_mat_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_mat_add
### Exit

### Start Compiling rtwdemo_sfun_mat_mult
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_mult.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEM
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_mat_mult
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you

try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_mat_add.tlc and rtwdemo_sfun_mat_mult.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

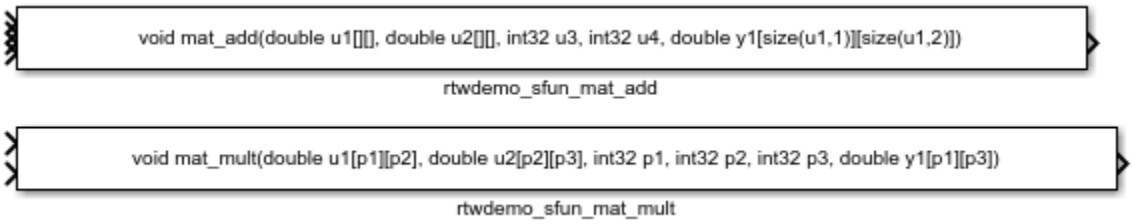
After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model

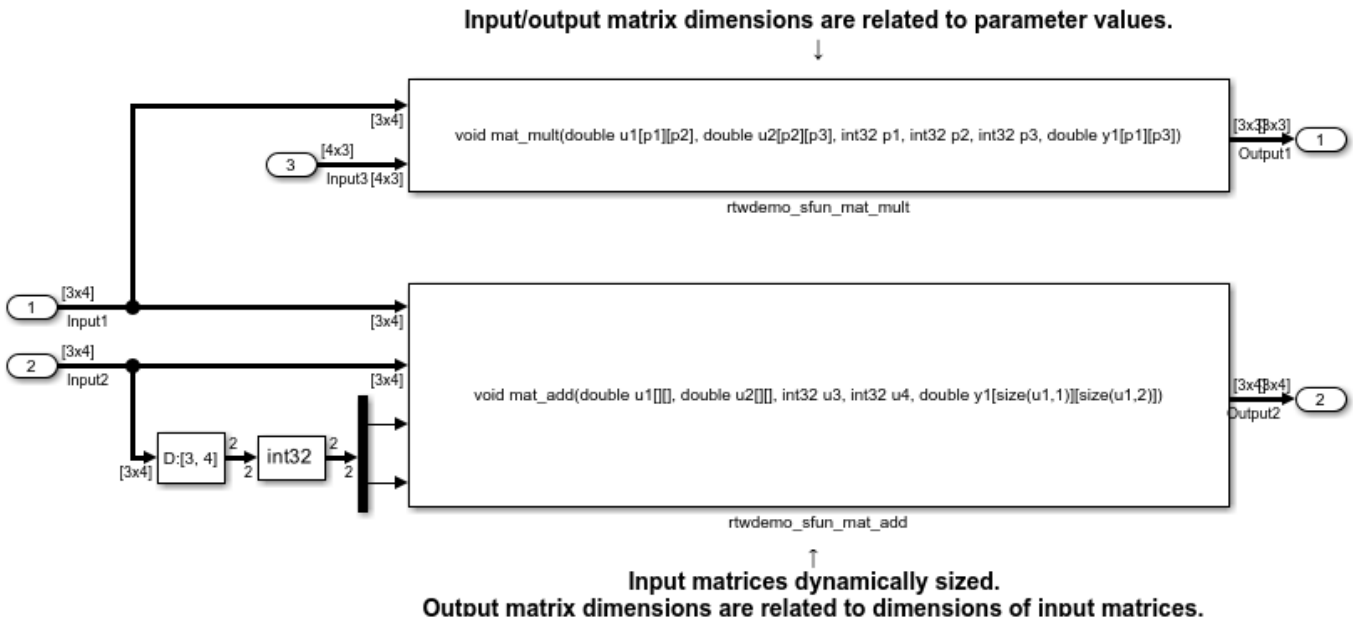
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_inherit_dims shows integration of the model with the legacy code. The subsystem TestMatOps serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_inherit_dims')
open_system('rtwdemo_lct_inherit_dims/TestMatOps')
sim('rtwdemo_lct_inherit_dims')
```



Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals

This example shows how to use the Legacy Code Tool to integrate legacy C functions with multi-dimensional Signals.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

void array3d_add(real_T *y1, real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, int32_T nbPages);

real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is in the files ndarray_ops.h, and ndarray_ops.c.

```
% rtwdemo_sfun_ndarray_add
def = legacy_code('initialize');
```

```
def.SFunctionName = 'rtwdemo_sfun_ndarray_add';
def.OutputFcnSpec =...
    ['void array3d_add(double y1[size(u1,1)][size(u1,2)][size(u1,3)], ',...
    'double u1[][][], double u2[][][], ' ...
    'int32 size(u1,1), int32 size(u1,2), int32 size(u1,3))'];
def.HeaderFiles   = {'ndarray_ops.h'};
def.SourceFiles   = {'ndarray_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

y1 is a 3-D output signal of same dimensions as the 3-D input signal u1. Note that the last 3 arguments passed to the legacy function correspond to the number of element in each dimension of the 3-D input signal u1.

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_ndarray_add.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_ndarray_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_ndarray_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_ndarray_add
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_ndarray_add.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

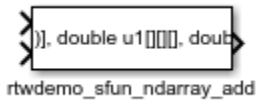
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

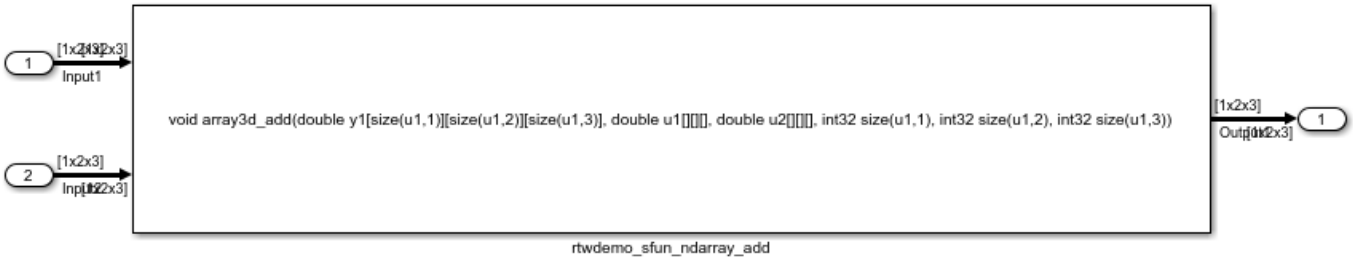
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model rtwdemo_lct_ndarray shows integration of the model with the legacy code. The subsystem ndarray_add serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_ndarray')
open_system('rtwdemo_lct_ndarray/ndarray_add')
sim('rtwdemo_lct_ndarray')
```



Integrate External C Functions That Implement Start and Terminate Actions

Integrate legacy C functions that have start and terminate actions by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first

input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototypes of the legacy functions being called in this example are:

- void initFaultCounter(unsigned int *counter)
- void openLogFile(void **fid)
- void incAndLogFaultCounter(void *fid, unsigned int *counter, double time)
- void closeLogFile(void **fid)

The legacy source code is in the files your_types.h, fault.h, and fault.c.

```
% rtwdemo_sfun_fault
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_fault';
def.InitializeConditionsFcnSpec  = 'initFaultCounter(uint32 work2[1])';
def.StartFcnSpec                 = 'openLogFile(void **work1)';
def.OutputFcnSpec                = ...
    'incAndLogFaultCounter(void *work1, uint32 work2[1], double u1)';
def.TerminateFcnSpec             = 'closeLogFile(void **work1)';
def.HeaderFiles                  = {'fault.h'};
def.SourceFiles                  = {'fault.c'};
def.IncPaths                     = {'rtwdemo_lct_src'};
def.SrcPaths                     = {'rtwdemo_lct_src'};
def.Options.useTlcWithAccel      = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_fault.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_fault
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_fault.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bd
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_fault
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC

block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_fault.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

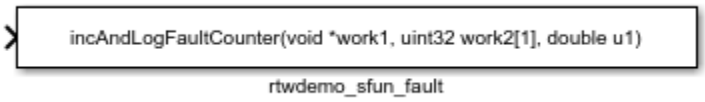
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

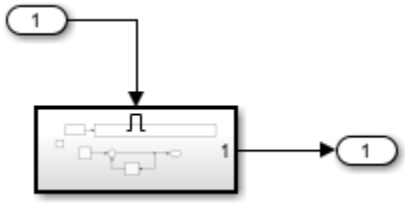
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model rtwdemo_lct_start_term shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the scope compares the output of the function with the output of the built-in Simulink® product block; the results should be identical.

```
open_system('rtwdemo_lct_start_term')
open_system('rtwdemo_lct_start_term/TestFault')
sim('rtwdemo_lct_start_term')
```



Integrate External C Functions That Implement N-Dimensional Table Lookups

Integrate legacy C functions that implement N-dimensional table lookups by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

FLT directLookupTableND(const FLT *tableND, const UINT32 nbDims, const UINT32 *tableDims, const UINT32 *tableIdx)

FLT is a typedef to float, and UINT32 is a typedef to unsigned int32. The legacy source code is in the files your_types.h, lookupTable.h, and directLookupTableND.c.

```
defs = [];
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_dlut3D
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_dlut3D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable3D(single p1[][][], uint32 p2[3], uint32 u1[3]
def.HeaderFiles   = {'lookupTable.h'};
def.SourceFiles   = {'directLookupTableND.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_dlut4D
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_dlut4D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable4D(single p1[][][][], uint32 p2[4], uint32 u1[4
def.HeaderFiles   = {'lookupTable.h'};
def.SourceFiles   = {'directLookupTableND.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again. Set the first input to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-functions is in the files rtwdemo_sfun_dlut3D.c and rtwdemo_sfun_dlut4D.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again. Set the first input to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_dlut3D
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_dlut3D.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_dlut3D
### Exit

### Start Compiling rtwdemo_sfun_dlut4D
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_dlut4D.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_dlut4D
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_dlut3D.tlc and rtwdemo_sfun_dlut4D.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

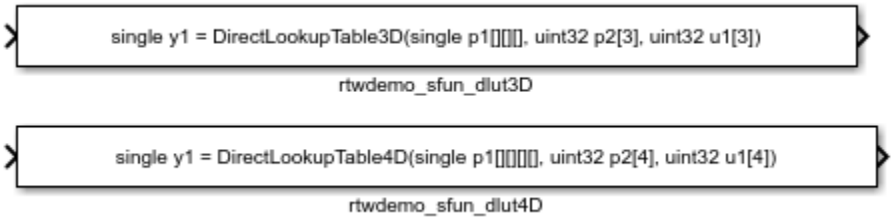
After you create the TLC block files, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. You can copy the blocks to an existing model.

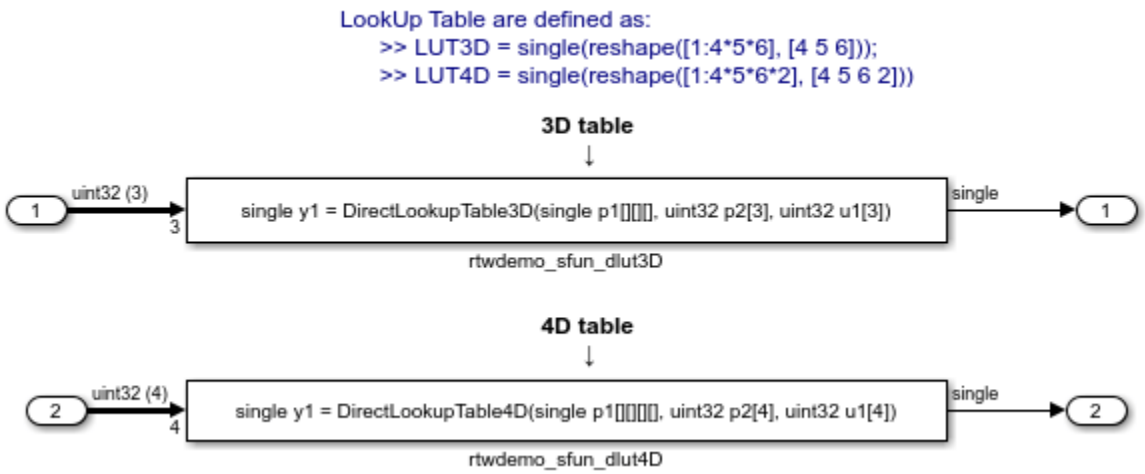
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_lut shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the Display blocks compare the output of the function with the output of the built-in Simulink® lookup blocks. The results are identical.

```
open_system('rtwdemo_lct_lut')
open_system('rtwdemo_lct_lut/TestLut1')
sim('rtwdemo_lct_lut')
```



Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals

This example shows how to use the Legacy Code Tool to integrate legacy C functions with multi-dimensional Signals.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

void array3d_add(real_T *y1, real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, int32_T nbPages);

real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is in the files ndarray_ops.h, and ndarray_ops.c.

```
% rtwdemo_sfun_ndarray_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_ndarray_add';
def.OutputFcnSpec =...
    ['void array3d_add(double y1[size(u1,1)][size(u1,2)][size(u1,3)], ',...
    'double u1[][][], double u2[][][], ' ...
    'int32 size(u1,1), int32 size(u1,2), int32 size(u1,3))'];
def.HeaderFiles   = {'ndarray_ops.h'};
def.SourceFiles   = {'ndarray_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

y1 is a 3-D output signal of same dimensions as the 3-D input signal u1. Note that the last 3 arguments passed to the legacy function correspond to the number of element in each dimension of the 3-D input signal u1.

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_ndarray_add.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_ndarray_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_ndarray_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_ndarray_add
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_ndarray_add.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

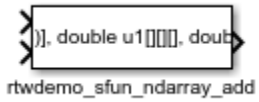
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

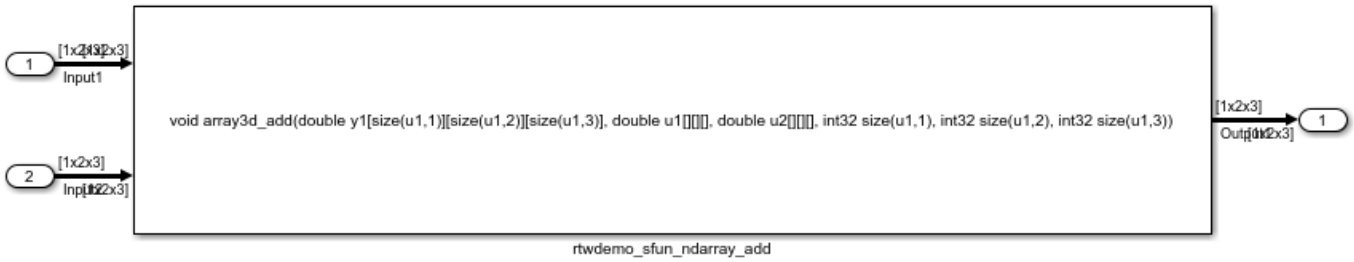
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model rtwdemo_lct_ndarray shows integration of the model with the legacy code. The subsystem ndarray_add serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_ndarray')
open_system('rtwdemo_lct_ndarray/ndarray_add')
sim('rtwdemo_lct_ndarray')
```



Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized

This example shows how to use the Legacy Code Tool to integrate legacy C functions with the block's sample time specified, inherited and parameterized.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

FLT gainScalar(const FLT in, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, gain.h, and gainScalar.c.

```
defs = [];

% rtwdemo_sfun_st_inherited
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_inherited';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_st_fixed
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_fixed';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
```

```
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.SampleTime    = [2 1];
defs = [defs; def];

% rtwdemo_sfun_st_parameterized
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_parameterized';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.SampleTime    = 'parameterized';
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-functions is in the files rtwdemo_sfun_st_inherited.c and rtwdemo_sfun_st_fixed.c. rtwdemo_sfun_st_parameterized.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_st_inherited
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_inherited.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_inherited
### Exit

### Start Compiling rtwdemo_sfun_st_fixed
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_fixed.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_fixed
### Exit

### Start Compiling rtwdemo_sfun_st_parameterized
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

```
    mex('rtwdemo_sfun_st_parameterized.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_parameterized
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_st_inherited.tlc and rtwdemo_sfun_st_fixed.tlc. rtwdemo_sfun_st_parameterized.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

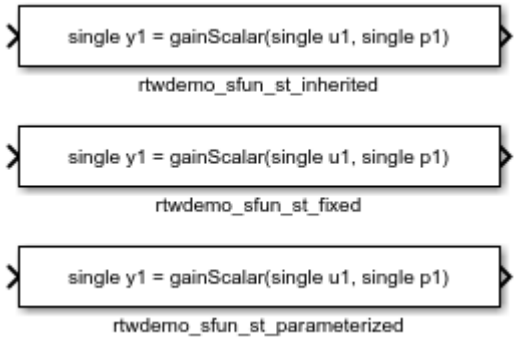
After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

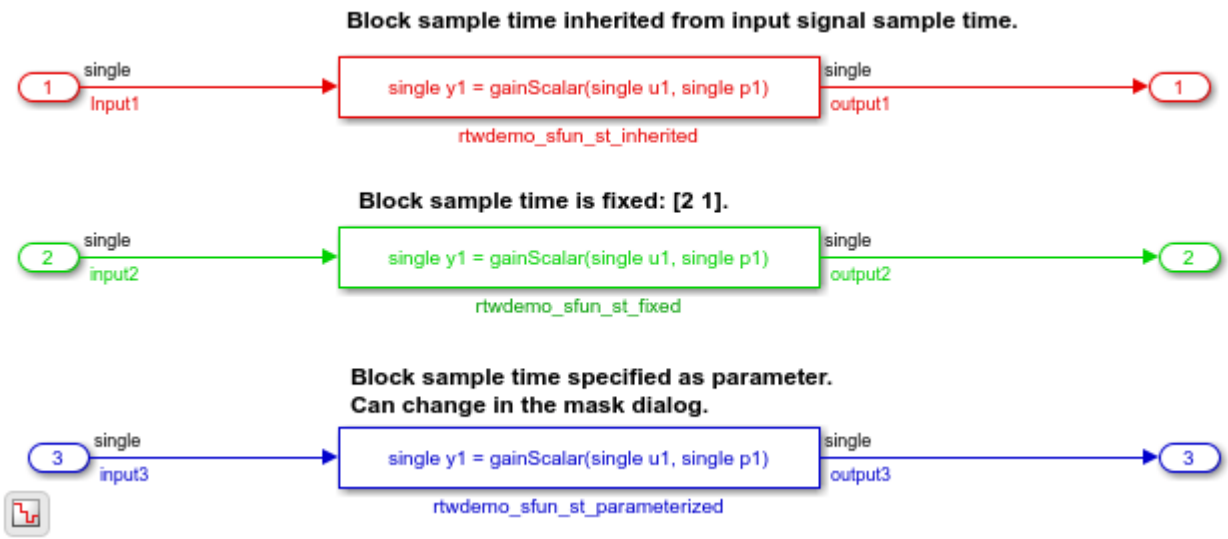
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_sampletime shows integration of the model with the legacy code. The subsystem sample_time serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_sampletime')
open_system('rtwdemo_lct_sampletime/sample_time')
sim('rtwdemo_lct_sampletime')
```



See Also

`legacy_code`

Related Examples

- “Import Calls to External Code into Generated Code with Legacy Code Tool”
- “Integrate C Functions Using Legacy Code Tool”
- “Call External C Code from Model and Generated Code” on page 53-48

Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type

Integrate legacy C functions that pass their inputs and outputs by using parameters of a fixed-point data type with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)

myFixpt is logically a fixed point data type, which is physically a typedef to a 16-bit integer:

```
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is in the files timesFixpt.h, and timesS16.c.

```
% rtwdemo_sfun_gain_fixpt
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_gain_fixpt';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt p1, uint8 p2)';
def.HeaderFiles   = {'timesFixpt.h'};
def.SourceFiles   = {'timesS16.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_gain_fixpt.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_gain_fixpt
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_gain_fixpt.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TE
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_gain_fixpt
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_gain_fixpt.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file that supports code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

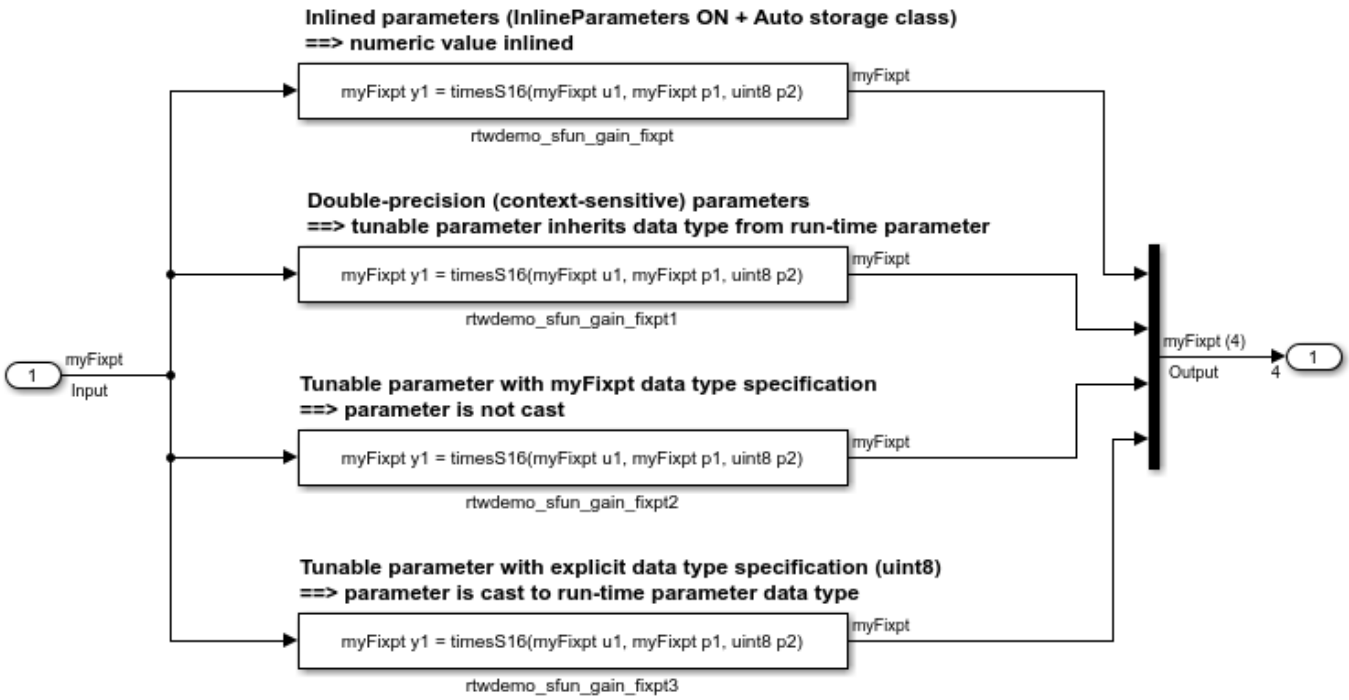
After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. You can copy the block to an existing model.

```
legacy_code('slblock_generate', def);
```

Show the Generated Integration with Legacy Code

The model rtwdemo_lct_fixpt_params shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function.

```
open_system('rtwdemo_lct_fixpt_params')
open_system('rtwdemo_lct_fixpt_params/TestFixpt')
sim('rtwdemo_lct_fixpt_params')
```



See Also

`legacy_code`

Related Examples

- “Integrate C Functions Using Legacy Code Tool”
- “Call External C Code from Model and Generated Code” on page 53-48

Integrate External C Functions That Implement N-Dimensional Table Lookups

Integrate legacy C functions that implement N-dimensional table lookups by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The prototype of the legacy functions being called in this example is:

FLT directLookupTableND(const FLT *tableND, const UINT32 nbDims, const UINT32 *tableDims, const UINT32 *tableIdx)

FLT is a typedef to float, and UINT32 is a typedef to unsigned int32. The legacy source code is in the files your_types.h, lookupTable.h, and directLookupTableND.c.

```
defs = [];
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_dlut3D
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_dlut3D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable3D(single p1[][][], uint32 p2[3], uint32 u1[3]
def.HeaderFiles   = {'lookupTable.h'};
def.SourceFiles   = {'directLookupTableND.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_dlut4D
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_dlut4D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable4D(single p1[][][][], uint32 p2[4], uint32 u1[4
def.HeaderFiles   = {'lookupTable.h'};
def.SourceFiles   = {'directLookupTableND.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again. Set the first input to 'sfcn_cmex_generate'. The S-functions call

the legacy functions during simulation. The source code for the S-functions is in the files rtwdemo_sfun_dlut3D.c and rtwdemo_sfun_dlut4D.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again. Set the first input to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_dlut3D
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_dlut3D.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\B
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_dlut3D
### Exit

### Start Compiling rtwdemo_sfun_dlut4D
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_dlut4D.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\B
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_dlut4D
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_dlut3D.tlc and rtwdemo_sfun_dlut4D.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

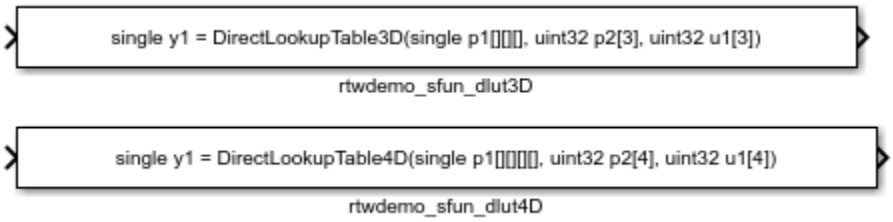
After you create the TLC block files, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. You can copy the blocks to an existing model.

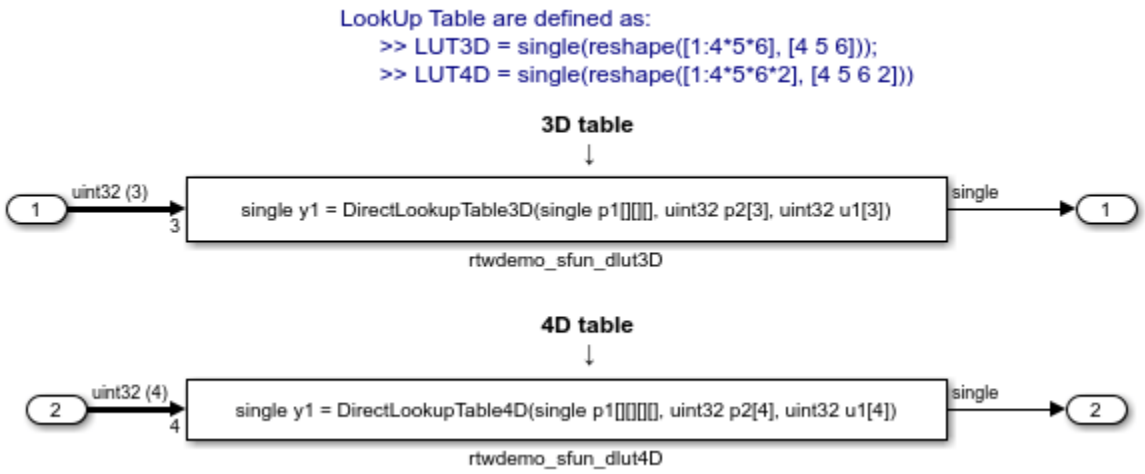
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_lut shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the Display blocks compare the output of the function with the output of the built-in Simulink® lookup blocks. The results are identical.

```
open_system('rtwdemo_lct_lut')
open_system('rtwdemo_lct_lut/TestLut1')
sim('rtwdemo_lct_lut')
```



See Also

`legacy_code`

Related Examples

- “Integrate C Functions Using Legacy Code Tool”
- “Call External C Code from Model and Generated Code” on page 53-48

Integrate External C++ Object Methods

Integrate legacy C++ object methods by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C++ MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a block TLC file and optional rtwmakecfg.m file that calls the legacy code during code generation.

Provide the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. The definition of the legacy C++ class in this example is:

```
class adder {
  private:
      int int_state;
  public:
      adder();
      int add_one(int increment);
      int get_val();
};
```

The legacy source code is in the files adder_cpp.h and adder_cpp.cpp.

```
% rtwdemo_sfun_adder_cpp
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_adder_cpp';
def.StartFcnSpec  = 'createAdder()';
def.OutputFcnSpec = 'int32 y1 = adderOutput(int32 u1)';
def.TerminateFcnSpec = 'deleteAdder()';
def.HeaderFiles   = {'adder_cpp.h'};
def.SourceFiles   = {'adder_cpp.cpp'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.Options.language = 'C++';
def.Options.useTlcWithAccel = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_adder_cpp.cpp.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_adder_cpp
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc23a_2213998_3568\ib57
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
    mex('rtwdemo_sfun_adder_cpp.cpp', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_adder_cpp
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file that supports code generation through Simulink® Coder™. If the TLC block file is not created and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_adder_cpp.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file that supports code generation through Simulink® Coder™. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

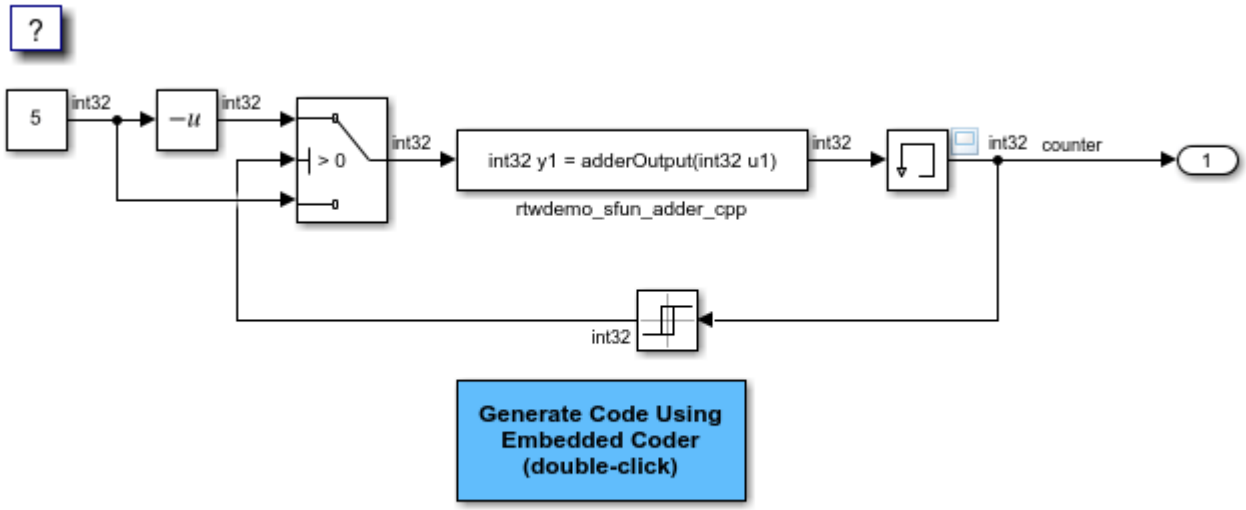
After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate a masked S-function block that is configured to call that S-function. The software places the block in a new model. You can copy the block to an existing model.

```
% legacy_code('slblock_generate', def);
```

Show the Generated Integration with Legacy Code

The model rtwdemo_lct_cpp shows integration with the legacy code.

```
open_system('rtwdemo_lct_cpp')
sim('rtwdemo_lct_cpp')
```



Copyright 1990-2020 The MathWorks, Inc.

See Also

`legacy_code`

Related Examples

- “Integrate C Functions Using Legacy Code Tool”
- “Call External C Code from Model and Generated Code” on page 53-48

External Code Integration Examples

This topic shows various scenarios of external code integration.

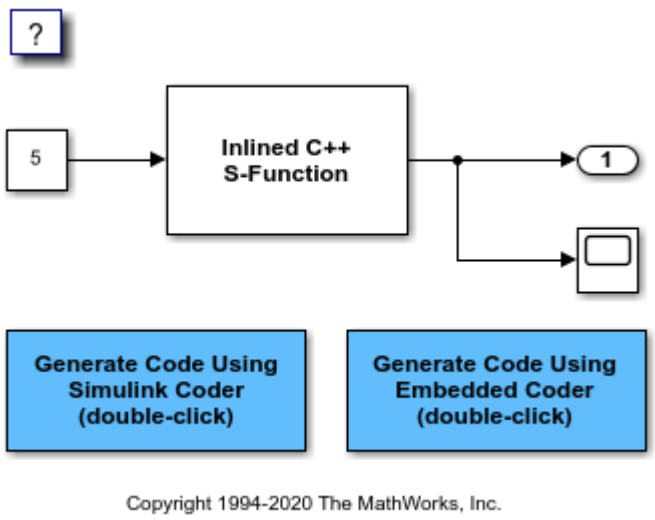
Integrate External C++ Code into a Model Using S-Functions

This example shows you how to integrate C++ code into a model by using S-functions.

Open Example Model

Open the example model `rtwdemo_cpp`.

```
open_system('rtwdemo_cpp');
```



Configure C++ Compiler

1. In the MATLAB® Command Window, enter this command:

```
mex -setup
```

2. When prompted, select a C++ compiler that is installed on your development computer.

Integrate C++ Code

You can integrate C++ code into a model by using S-functions.

1. Prepare the S-function source.

- Open sfun_cppcount_cpp.cpp
- Open sfun_cppcount_cpp.h

2. Open the **Simulink Coder** or **Embedded Coder** app.

3. Configure S-function source files and folders. Open the Model Configuration Parameter dialog box, navigate to the **Code Generation > Custom Code** pane, and add the S-function source files and `include` folders.

4. Inline the S-function to eliminate overhead of S-functions in the generated code.

Open sfun_cppcount_cpp.tlc

Configure Model for C++ Code Generation

Set model configuration parameter **Language** to `C++`.

Generate and review code.

Generate and reviw the C++ code.

Related Topics

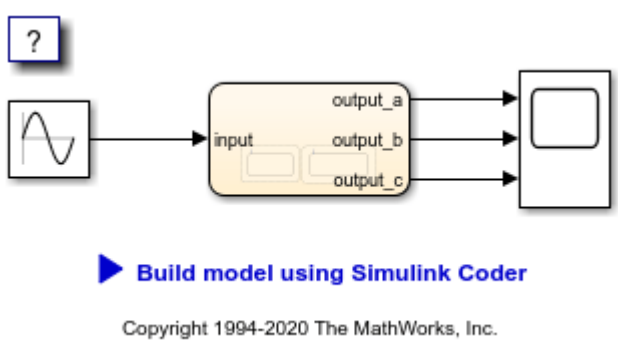
- “Programming Language Considerations”
- “C++ Language Support Limitations”

Insert External C and C++ Code into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external code into a model.

Open Model

```
model='rtwdemo_sfcustom';
open_system(model);
```



Integrate Code

1. The example includes the custom header file `my_header.c` and the custom source file `my_function.c`.

```
%Open files my_header.h and my_function.c
eval('edit my_header.h')
eval('edit my_function.c')
```

2. On the Model Configuration Parameters dialog box **Simulation Target** pane, enter the custom source file and header file. Also enter additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Simulation Target** pane.

```
%Open Configuration Parameters dialog box
configSet = getActiveConfigSet(model);
openDialog(configSet);
```

3. To generate code, on the Model Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the same custom source file and header file. Also enter the same additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Code Generation > Custom Code** pane.

Open the **Code Generation > Custom Code** pane of the Model Configuration Parameters dialog box.

```
configSet = getActiveConfigSet(model);
openDialog(configSet);
```

Generate Code

```
slbuild('rtwdemo_sfcustom')
```

```
### Starting build procedure for: rtwdemo_sfcustom
### Successful completion of build procedure for: rtwdemo_sfcustom

Build Summary

Top model targets built:

Model             Action                         Rebuild Reason
==========================================================================================
rtwdemo_sfcustom  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 8.4901s
```

Call C Code from Stateflow

To call custom C code functions from Stateflow, use the same syntax as graphical function calls: `result = my_custom_function(in_args);`

To call variables of structure type, use the dot notation: `result = my_var.my_field;`

See Also

- “Configure Custom Code Settings for Simulation” (Stateflow)
- “Access Custom C++ Code in Stateflow Charts” (Stateflow)

Close Model

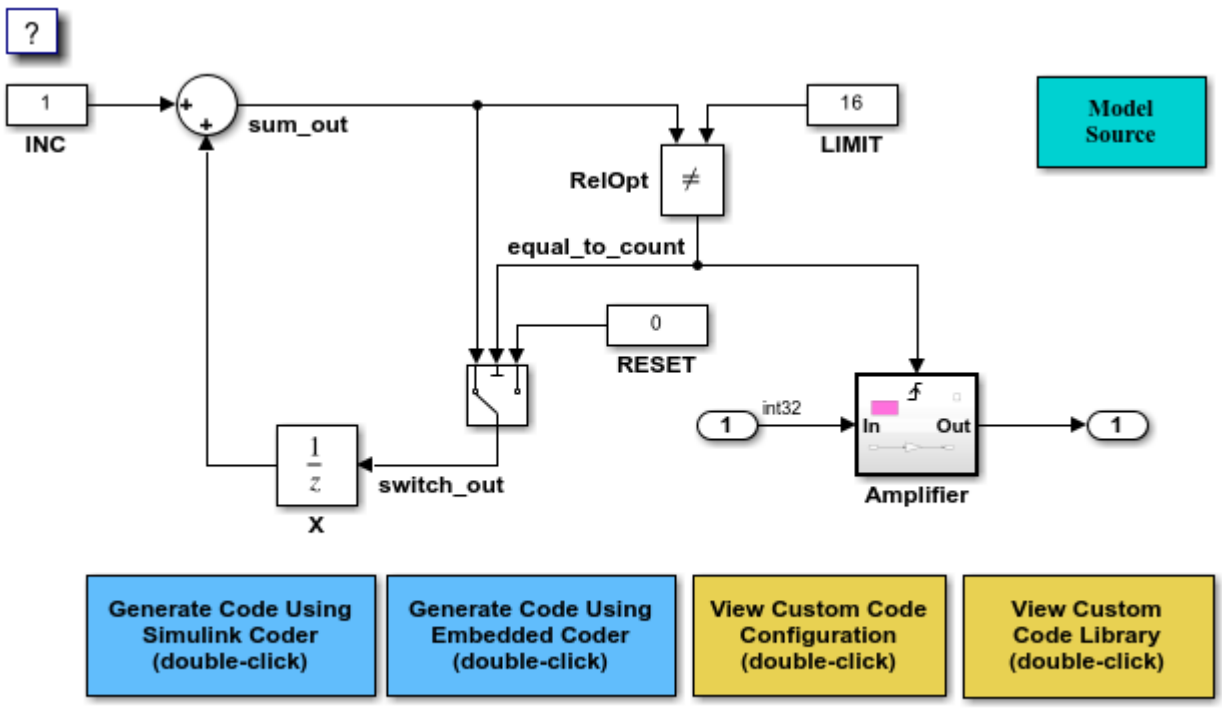
```
rtwdemoclean;
close_system('rtwdemo_sfcustom',0);
```

Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



2. Open the **Simulink Coder** or **Embedded Coder** app.

3. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

4. On the **Additional source code** tab, examine the settings for model configuration parameters **Additional code** and **Initialize code**.

- **Additional code** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize code** initializes the variable `GLOBAL_INT2` to 1.

5. Close the dialog box.

6. Double-click the Model Source block. Block parameter **Top of Model Source** specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

7. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and initializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

8. Generate code and a code generation report.

9. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Additional code** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;

int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Outputs block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;

*intPtr = -1;

rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;

*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize code** configuration parameter.

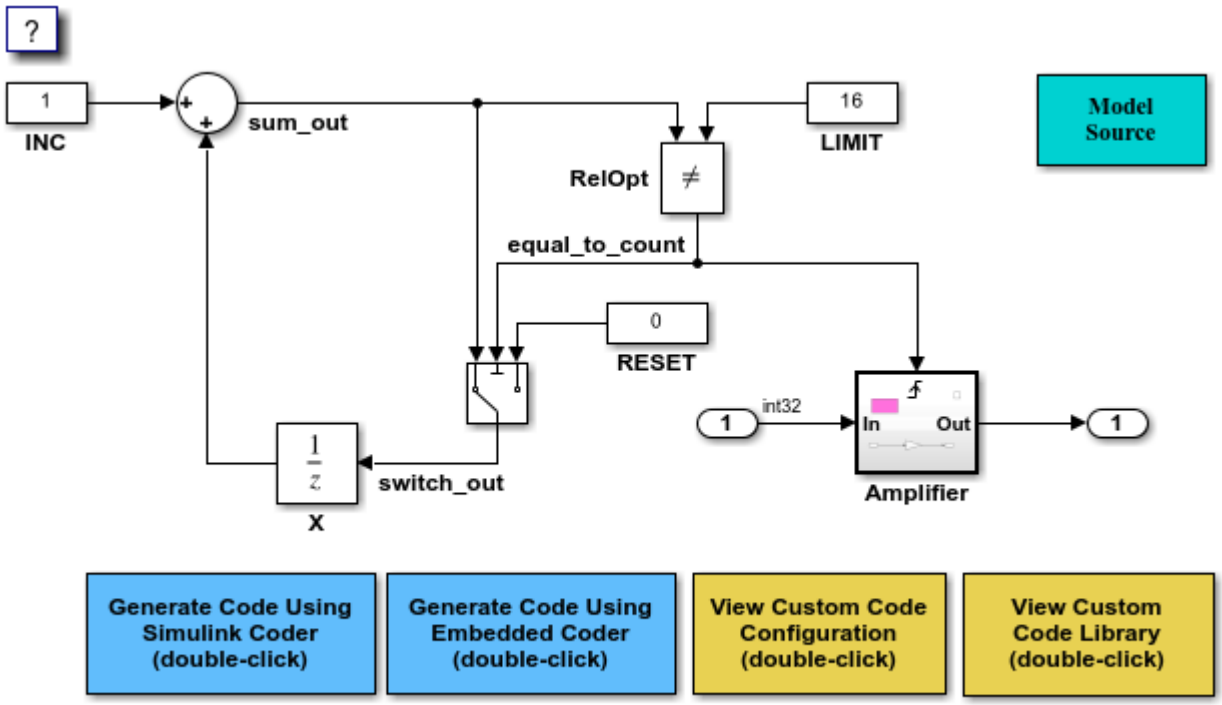
```
GLOBAL_INT2 = 1;
```

Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



Copyright 1994-2022 The MathWorks, Inc.

2. Open the **Simulink Coder** or **Embedded Coder** app.

3. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

4. On the **Additional source code** tab, examine the settings for model configuration parameters **Additional code** and **Initialize code**.

- **Additional code** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize code** initializes the variable `GLOBAL_INT2` to 1.

5. Close the dialog box.

6. Double-click the Model Source block. Block parameter **Top of Model Source** specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

7. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and initializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

8. Generate code and a code generation report.

9. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Additional code** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;
```

```
int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Outputs block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;
```

```
*intPtr = -1;
```

```
rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;
```

```
*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize code** configuration parameter.

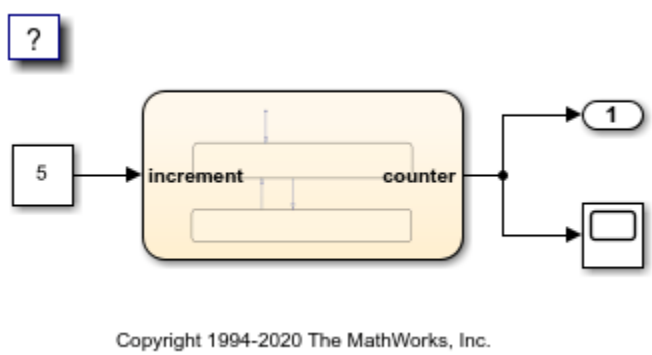
```
GLOBAL_INT2 = 1;
```

Insert External C++ Code into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external C++ code into a model.

Open Model

```
model='rtwdemo_sfcpp';
open_system(model);
```



To setup a C++ compiler, on the command line, enter `mex -setup`.

Integrate C++ Code

1. Create C-style wrapper functions or access macros for each method called from the C action language.

```
%Open example wrapper functions.
filename=fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'custom_cpp_src', 'adder_cpp.cpp');
eval('edit(filename)');
clear filename;

filename=fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'custom_cpp_src', 'adder_cpp.h');
eval('edit(filename)');
clear filename;
```

2. On the Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the S-Function source files and include directories.

```
%Open Configuration Parameters dialog box
configSet = getActiveConfigSet(model);
openDialog(configSet);
```

3. On the Configuration Parameters dialog box **Simulation Target > Custom Code** pane, enter the S-Function source files and include directories.

```
%Open Configuration Parameters dialog box
configSet = getActiveConfigSet(model);
openDialog(configSet);
```

Generate C++ Code

On the Configuration Parameters dialog box **Code Generation** pane, set **Language** to `C++`. To generate code, select **Build**.

```
%Open Configuration Parameters dialog box
configSet = getActiveConfigSet(model);
openDialog(configSet);
clear model;
```

See Also

- “Programming Language Considerations”
- “C++ Language Support Limitations”

Close Model

```
rtwdemoclean;
close_system('rtwdemo_sfcpp',0);
```

S-Functions That Support Expression Folding

Use expression folding to increase the efficiency of code generated by your own inlined S-Function blocks, by calling macros provided in the S-Function API.

The S-Function API lets you specify whether a given S-Function block should nominally accept expressions at a given input port. A block should not always accept expressions. For example, if the address of the signal at the input is used, expressions should not be accepted at that input, because it is not possible to take the address of an expression.

The S-Function API also lets you specify whether an expression can represent the computations associated with a given output port. When you request an expression at a block's input or output port, the Simulink engine determines whether or not it can honor that request, given the block's context. For example, the engine might deny a block's request to output an expression if the destination block does not accept expressions at its input, if the destination block has an update function, or if multiple output destinations exist.

The decision to honor or deny a request to output an expression can also depend on the category of output expression the block uses.

To take advantage of expression folding in your S-functions, you should understand when to request acceptance and generation of expressions for specific blocks. You do not have to understand the algorithm by which the Simulink engine chooses to accept or deny these requests. However, if you want to trace between the model and the generated code, it is helpful to understand some of the more common situations that lead to denial of a request.

Categories of Output Expressions

When you implement a C MEX S-function, you can specify whether the code corresponding to a block's output is to be generated as an expression. If the block generates an expression, you must specify that the expression is *constant*, *trivial*, or *generic*.

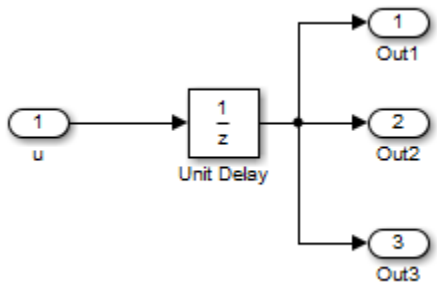
A *constant* output expression is a direct access to one of the block's parameters. For example, the output of a Constant block is defined as a constant expression because the output expression is simply a direct access to the block's `Value` parameter.

A *trivial* output expression is an expression that can be repeated, without a performance penalty, when the output port has multiple output destinations. For example, the output of a Unit Delay block is defined as a trivial expression because the output expression is simply a direct access to the block's state. Because the output expression does not have computations, it can be repeated more than once without degrading the performance of the generated code.

A *generic* output expression is an expression that should be assumed to have a performance penalty if repeated. As such, a generic output expression is not suitable for repeating when the output port has multiple output destinations. For instance, the output of a Sum block is a generic rather than a trivial expression because it is costly to recompute a Sum block output expression as an input to multiple blocks.

Examples of Trivial and Generic Output Expressions

Consider this block diagram. The Delay block has multiple destinations, yet its output is designated as a trivial output expression, so that it can be used more than once without degrading the efficiency of the code.

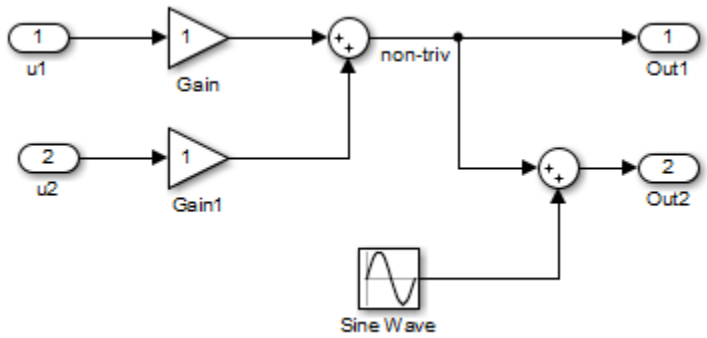


This code excerpt shows the code generated from the Unit Delay block in this block diagram. The three root outputs are directly assigned from the state of the Unit Delay block, which is stored in a field of the global data structure `rtDWork`. Since the assignment is direct, without expressions, there is no performance penalty associated with using the trivial expression for multiple destinations.

```
void MdlOutputs(int_T tid)
{
   ...
  /* Outport: <Root>/Out1 incorporates:

   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out1 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <Root>/Out2 incorporates:
   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out2 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <Root>/Out3 incorporates:
   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out3 = rtDWork.Unit_Delay_DSTATE;

   ...
}
```

The code generated shows how code is generated for Sum blocks with single and multiple destinations.

On the other hand, consider the Sum blocks in this model:



The upper Sum block in the model generates the signal labeled `non_triv`. Computation of this output signal involves two multiplications and an addition. If the Sum block's output were permitted to generate an expression even when the block had multiple destinations, the block's operations would be duplicated in the generated code. In this case, the generated expressions would proliferate to four multiplications and two additions. This would degrade the efficiency of the program. Accordingly the output of the Sum block is not allowed to be an expression because it has multiple destinations.

The Simulink engine does not permit the output of the upper Sum block to be an expression because the signal `non_triv` is routed to two output destinations. Instead, the result of the multiplication and addition operations is stored in a temporary variable `(rtb_non_triv)` that is referenced twice in the statements that follow, as seen in the code excerpt below.

In contrast, the lower Sum block, which has only a single output destination (`Out2`), does generate an expression.

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_non_triv;
  real_T rtb_Sine_Wave;

  /* Sum: <Root>/Sum incorporates:
   *   Gain: <Root>/Gain
   *   Inport: <Root>/u1
   *   Gain: <Root>/Gain1
   *   Inport: <Root>/u2
   *
   * Regarding <Root>/Gain:
   *   Gain value: rtP.Gain_Gain
   *
   * Regarding <Root>/Gain1:
   *   Gain value: rtP.Gain1_Gain
   */
  rtb_non_triv = (rtP.Gain_Gain * rtU.u1) + (rtP.Gain1_Gain *
rtU.u2);

  /* Outport: <Root>/Out1 */
  rtY.Out1 = rtb_non_triv;

  /* Sin Block: <Root>/Sine Wave */

  rtb_Sine_Wave = rtP.Sine_Wave_Amp *
    sin(rtP.Sine_Wave_Freq * rtmGetT(rtM_model) +
    rtP.Sine_Wave_Phase) + rtP.Sine_Wave_Bias;

  /* Outport: <Root>/Out2 incorporates:
   *   Sum: <Root>/Sum1
   */
  rtY.Out2 = (rtb_non_triv + rtb_Sine_Wave);
}
```

Specify the Category of an Output Expression

The S-Function API provides macros that let you declare whether an output of a block should be an expression, and if so, to specify the category of the expression. This table specifies when to declare a block output to be a constant, trivial, or generic output expression.

Types of Output Expressions

Category of Expression	When to Use
Constant	Use only if block output is a direct memory access to a block parameter.
Trivial	Use only if block output is an expression that can appear multiple times in the code without reducing efficiency (for example, a direct memory access to a field of the `DWork` vector, or a literal).
Generic	Use if output is an expression, but not constant or trivial.

You must declare outputs as expressions in the `mdlSetWorkWidths` function using macros defined in the S-Function API. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's `SimStruct`.
- `int idx:` zero-based index of the output port.
- `bool value:` pass in TRUE if the port generates output expressions.

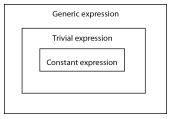
The following macros are available for setting an output to be a constant, trivial, or generic expression:

- `void ssSetOutputPortConstOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortOutputExprInRTW(SimStruct *S, int idx, bool value)`

The following macros are available for querying the status set by prior calls to the macros above:

- `bool ssGetOutputPortConstOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortOutputExprInRTW(SimStruct *S, int idx)`

The set of generic expressions is a superset of the set of trivial expressions, and the set of trivial expressions is a superset of the set of constant expressions.



Therefore, when you query an output that has been set to be a constant expression with `ssGetOutputPortTrivialOutputExprInRTW`, it returns `True`. A constant expression is considered a trivial expression because it is a direct memory access that can be repeated without degrading the efficiency of the generated code.

Similarly, an output that has been configured to be a constant or trivial expression returns `True` when queried for its status as a generic expression.

Acceptance or Denial of Requests for Input Expressions

A block can request that its output be represented in code as an expression. Such a request can be denied if the destination block cannot accept expressions at its input port. Furthermore, conditions independent of the requesting block and its destination blocks can prevent acceptance of expressions.

A block should not be configured to accept expressions at its input port under the following conditions:

- The block must take the address of its input data. It is not possible to take the address of most types of input expressions.
- The code generated for the block references the input more than once (for example, the Abs or Max blocks). This would lead to duplication of a potentially complex expression and a subsequent degradation of code efficiency.

If a block refuses to accept expressions at an input port, then a block that is connected to that input port is not permitted to output a generic or trivial expression.

A request to output a constant expression is not denied, because there is no performance penalty for a constant expression, and the software can take the parameter's address.

S-Function API to Specify Input Expression Acceptance

The S-Function API provides macros that let you:

- Specify whether a block input should accept nonconstant expressions (that is, trivial or generic expressions).
- Query whether a block input accepts nonconstant expressions.

By default, block inputs do not accept nonconstant expressions.

You should call the macros in your `mdlSetWorkWidths` function. The macros have these arguments:

- `SimStruct *S`: pointer to the block's SimStruct.
- `int idx:` zero-based index of the input port.
- `bool value:` pass in TRUE if the port accepts input expressions; otherwise pass in FALSE.

The macro available for specifying whether or not a block input should accept a nonconstant expression is:

```
void ssSetInputPortAcceptExprInRTW(SimStruct *S, int portIdx, bool value)
```

The corresponding macro available for querying the status set by any prior calls to `ssSetInputPortAcceptExprInRTW` is:

```
bool ssGetInputPortAcceptExprInRTW(SimStruct *S, int portIdx)
```

Denial of Block Requests to Output Expressions

Even after a specific block requests that it be allowed to generate an output expression, that request can be denied for generic reasons. These reasons include, but are not limited to:

- The output expression is nontrivial, and the output has multiple destinations.
- The output expression is nonconstant, and the output is connected to at least one destination that does not accept expressions at its input port.
- The output is a test point.
- The output has been assigned an external storage class.
- The output must be stored using global data (for example is an input to a merge block or a block with states).

- The output signal is complex.

You do not need to consider these generic factors when deciding whether or not to utilize expression folding for a particular block. However, these rules can be helpful when you are examining generated code and analyzing cases where the expression folding optimization is suppressed.

Expression Folding in a TLC Block Implementation

To take advantage of expression folding, modify the TLC block implementation of an inlined S-Function such that it informs the Simulink engine whether it generates or accepts expressions at its

- Input ports, as explained in “S-Function API to Specify Input Expression Acceptance” on page 13-68.
- Output ports, as explained in “Categories of Output Expressions” on page 13-64.

This topic discusses required modifications to the TLC implementation.

Expression Folding Compliance

In the `BlockInstanceSetup` function of your S-function, register your block to be compliant with expression folding. Otherwise, expression folding requested or allowed at the block's outputs or inputs will be disabled, and temporary variables will be used.

To register expression folding compliance, call the TLC library function `LibBlockSetIsExpressionCompliant(block)`, which is defined in *`matlabroot`*`/rtw/c/tlc/lib/utillib.tlc`. For example:

```
%% Function: BlockInstanceSetup ==========================================
%%
%function BlockInstanceSetup(block, system) void
  %%
  %<LibBlockSetIsExpressionCompliant(block)>
  %%
%endfunction
```

You can conditionally disable expression folding at the inputs and outputs of a block by making the call to this function conditionally.

If you override one of the TLC block implementations provided by the code generator with your own implementation, you should not make the preceding call until you have updated your implementation.

Output Expressions

The `BlockOutputSignal` function is used to generate code for a scalar output expression or one element of a nonscalar output expression. If your block outputs an expression, you should add a `BlockOutputSignal` function. The prototype of the `BlockOutputSignal` is

```
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
```

The arguments to `BlockOutputSignal` are these:

- `block`: the record for the block for which an output expression is being generated
- `system`: the record for the system containing the block
- `portIdx`: zero-based index of the output port for which an expression is being generated
- `ucv`: user control variable defining the output element for which code is being generated

- `lcv`: loop control variable defining the output element for which code is being generated
- `idx`: signal index defining the output element for which code is being generated
- `retType`: character vector defining the type of signal access desired:

  `"Signal"` specifies the contents or address of the output signal

  `"SignalAddr"` specifies the address of the output signal

The `BlockOutputSignal` function returns a character vector for the output signal or address. The character vector should enforce the precedence of the expression by using opening and terminating parentheses, unless the expression consists of a function call. The address of an expression can only be returned for a constant expression; it is the address of the parameter whose memory is being accessed. The code implementing the `BlockOutputSignal` function for the Constant block is:

```
%% Function: BlockOutputSignal ================================================
%% Abstract:
%%      Return the reference to the parameter.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
  %switch retType
    %case "Signal"
      %return LibBlockParameter(Value,ucv,lcv,idx)
    %case "SignalAddr"
      %return LibBlockParameterAddr(Value,ucv,lcv,idx)
    %default
      %assign errTxt = "Unsupported return type: %<retType>"
      %<LibBlockReportError(block,errTxt)>
  %endswitch
%endfunction
```

The code implementing the `BlockOutputSignal` function for the Relational Operator block is:

```
%% Function: BlockOutputSignal ================================================
%% Abstract:
%%      Return an output expression.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
%switch retType
%case "Signal"
%assign logicOperator = ParamSettings.Operator
 %if ISEQUAL(logicOperator, "~=")
 %assign op = "!="
elseif ISEQUAL(logicOperator, "==") %assign op = "=="
  %else
%assign op = logicOperator
%endif
 %assign u0 = LibBlockInputSignal(0, ucv, lcv, idx)
%assign u1 = LibBlockInputSignal(1, ucv, lcv, idx)
  %return "(%<u0> %<op> %<u1>)"
 %default
 %assign errTxt = "Unsupported return type: %<retType>"
 %<LibBlockReportError(block,errTxt)>
%endswitch
%endfunction
```

Expression Folding for Blocks with Multiple Outputs

When a block has a single output, the `Outputs` function in the block's TLC file is called only if the output port is not an expression. Otherwise, the `BlockOutputSignal` function is called.

If a block has multiple outputs, the `Outputs` function is called if any output port is not an expression. The `Outputs` function should guard against generating code for output ports that are expressions.

This is achieved by guarding sections of code corresponding to individual output ports with calls to `LibBlockOutputSignalIsExpr()`.

For example, consider an S-Function with two inputs and two outputs, where

- The first output, y0, is equal to two times the first input.
- The second output, y1, is equal to four times the second input.

The `Outputs` and `BlockOutputSignal` functions for the S-function are shown in this code excerpt:

```
%% Function: BlockOutputSignal ==================================================
%% Abstract:
%%      Return an output expression.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
%switch retType
%case "Signal"
   %assign u = LibBlockInputSignal(portIdx, ucv, lcv, idx)
 %case "Signal"
 %if portIdx == 0
  %return "(2 * %<u>)"
%elseif portIdx == 1
 %return "(4 * %<u>)"
%endif
%default
%assign errTxt = "Unsupported return type: %<retType>"
 %<LibBlockReportError(block,errTxt)>
%endswitch
%endfunction
%%
%% Function: Outputs ============================================================
%% Abstract:
%%      Compute output signals of block
%%
%function Outputs(block,system) Output
%assign rollVars = ["U", "Y"]
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
%assign u0 = LibBlockInputSignal(0, "", lcv, sigIdx)
 %assign u1 = LibBlockInputSignal(1, "", lcv, sigIdx)
 %assign y0 = LibBlockOutputSignal(0, "", lcv, sigIdx)
 %assign y1 = LibBlockOutputSignal(1, "", lcv, sigIdx)
%if !LibBlockOutputSignalIsExpr(0)
%<y0> = 2 * %<u0>;
%endif
%if !LibBlockOutputSignalIsExpr(1)
 %<y1> = 4 * %<u1>;
%endif
%endroll
%endfunction
```

Comments for Blocks That Are Expression-Folding-Compliant

In the past, blocks preceded their outputs code with comments of the form

```
/* %<Type> Block: %<Name> */
```

When a block is expression-folding-compliant, the initial line shown above is generated automatically. Do not include the comment as part of the block's TLC implementation. Register additional information by using the `LibCacheBlockComment` function.

The `LibCacheBlockComment` function takes a character vector as an input, defining the body of the comment, except for the opening header, the final newline of a single or multiline comment, and the closing trailer.

The following TLC code illustrates registering a block comment. Note the use of the function `LibBlockParameterForComment`, which returns a character vector, suitable for a block comment, specifying the value of the block parameter.

```
%openfile commentBuf
  $c(*) Gain value: %<LibBlockParameterForComment(Gain)>
  %closefile commentBuf
  %<LibCacheBlockComment(block, commentBuf)>
```

See Also

Related Examples

- “S-Functions That Specify Port Scope and Reusability” on page 13-73
- “S-Functions That Specify Sample Time Inheritance Rules” on page 13-77
- “S-Functions for Code Reuse” on page 13-81

S-Functions That Specify Port Scope and Reusability

You can use the following `SimStruct` macros in the `mdlInitializeSizes` method to specify the scope and reusability of the memory used for your S-function's input and output ports:

- `ssSetInputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function input port.
- `ssSetOutputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function output port.
- `ssSetInputPortOverWritable`: Specify whether one of your S-function's input ports can be overwritten by one of its output ports.
- `ssSetOutputPortOverwritesInputPort`: Specify whether an output port can share its memory buffer with an input port.

You declare an input or output as local or global, and indicate its reusability, by passing one of the following four options to the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros:

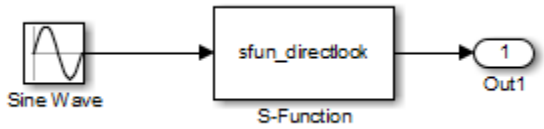
- `SS_NOT_REUSABLE_AND_GLOBAL`: Indicates that the input and output ports are stored in separate memory locations in the global block input and output structure.
- `SS_NOT_REUSABLE_AND_LOCAL`: Indicates that the code generator can declare individual local variables for the input and output ports.
- `SS_REUSABLE_AND_LOCAL`: Indicates that the code generator can reuse a single local variable for these input and output ports.
- `SS_REUSABLE_AND_GLOBAL`: Indicates that these input and output ports are stored in a single element in the global block input and output structure.

Note Marking an input or output port as a local variable does not imply that the code generator uses a local variable in the generated code. If your S-function accesses the inputs and outputs only in its `mdlOutputs` routine, the code generator declares the inputs and outputs as local variables. However, if the inputs and outputs are used elsewhere in the S-function, the code generator includes them in the global block input and output structure.

The reusability setting indicates if the memory associated with an input or output port can be overwritten. To reuse input and output port memory:

1. Indicate the ports are reusable using either the `SS_REUSABLE_AND_LOCAL` or `SS_REUSABLE_AND_GLOBAL` option in the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros.
2. Indicate the input port memory is overwritable using `ssSetInputPortOverWritable`.
3. If your S-function has multiple input and output ports, use `ssSetOutputPortOverwritesInputPort` to indicate which output and input ports share memory.

The following example shows how different scope and reusability settings affect the generated code. The following model contains an S-function block pointing to the C MEX S-function *`matlabroot/`*`toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c`, which models a direct 1-D lookup table.



The S-function's `mdlInitializeSizes` method declares the input port as reusable, local, and overwritable and the output port as reusable and local, as follows:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* snip */
    ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);
    ssSetInputPortOverWritable(S, 0, TRUE);

/* snip */
    ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

/* snip */
}
```

The generated code for this model stores the input and output signals in a single local variable `rtb_SFunction`, as shown in the following output function:

```
static void sl_directlook_output(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_SFunction[2];

  /* Sin: '<Root>/Sine Wave' */
  rtb_SFunction[0] = sin(((real_T)sl_directlook_DWork.counter[0] +
    sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
    sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[0] +
    sl_directlook_P.SineWave_Bias;
  rtb_SFunction[1] = sin(((real_T)sl_directlook_DWork.counter[1] +
    sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
    sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[1] +
    sl_directlook_P.SineWave_Bias;

  /* S-Function Block: <Root>/S-Function */
  {
    const real_T *xData = &sl_directlook_P.SFunction_XData[0];
    const real_T *yData = &sl_directlook_P.SFunction_YData [0];
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction[0] <= xData[0] ) {
      rtb_SFunction[0] = yData[0];
    } else if (rtb_SFunction[0] >= yData[20] ) {
      rtb_SFunction[0] = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction[0] - xData[0] ) / spacing );
      rtb_SFunction[0] = yData[idx];
    }

    if (rtb_SFunction[1] <= xData[0] ) {
      rtb_SFunction[1] = yData[0];
    } else if (rtb_SFunction[1] >= yData[20] ) {
      rtb_SFunction[1] = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction[1] - xData[0] ) / spacing );
      rtb_SFunction[1] = yData[idx];
    }
  }

  /* Outport: '<Root>/Out1' */
  sl_directlook_Y.Out1[0] = rtb_SFunction[0];
  sl_directlook_Y.Out1[1] = rtb_SFunction[1];
  UNUSED_PARAMETER(tid);
}
```

This table shows variations of the code generated for this model when using the generic real-time target (GRT). Each row explains a different setting for the scope and reusability of the S-function's input and output ports.

Scope and reusability	S-function `mdlInitializeSizes` code	Generated code

Inputs: Local, reusable, overwritable
 	
Outputs: Local, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares a local variable in the output function.
		
		`/* local block i/o variables */`
		`  real_T rtb_SFunction[2];`

Inputs: Global, reusable, overwritable
 	
Outputs: Global, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signals structure with a single element to store the S-function's input and output.
		
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`  real_T SFunction[2];`
		`} BlockIO_sl_directlook;`
		
		The *`model`*`.c` file uses this element of the structure in calculations of the S-function's input and output signals.
		
		`/* Sin: '<Root>/Sine Wave' */`
		`sl_directlook_B.SFunction[0] = sin ...`
		`/* snip */`
		`/*S-Function Block:<Root>/S-Function*/`
		`{`
		`const real_T *xData =`
		`  &sl_directlook_P.SFunction_XData[0]`

Inputs: Local, not reusable
 	
Outputs: Local, not reusable
	`ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0, FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares local variables for the S-function's input and output in the output function
		
		`/* local block i/o variables */`
		`  real_T rtb_SineWave[2];`
		`  real_T rtb_SFunction[2];`

Inputs: Global, not reusable
 	
Outputs: Global, not reusable
	`ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0, FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signal structure with individual elements to store the S-function's input and output.
		
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`  real_T SineWave[2];`
		`  real_T SFunction[2];`
		`} BlockIO_sl_directlook;`
		
		The *`model`*`.c` file uses the different elements in this structure when calculating the S-function's input and output.
		
		`/* Sin: '<Root>/Sine Wave' */`
		`sl_directlook_B.SineWave[0] = sin ...`
		`/* snip */`
		`/*S-Function Block:<Root>/S-Function*/`
		`{`
		`const real_T *xData =`
		`  &sl_directlook_P.SFunction_XData[0]`

To check if the S-function input or output buffer reuse takes place, you can use `LibBlockInputSignalBufferDstPort` function.

See Also

Related Examples

- “S-Functions That Support Expression Folding” on page 13-64
- “S-Functions That Specify Sample Time Inheritance Rules” on page 13-77
- “S-Functions for Code Reuse” on page 13-81

S-Functions That Specify Sample Time Inheritance Rules

For the Simulink engine to determine whether a model can inherit a sample time from a parent model, the S-functions in the model need to specify how they use sample times. You can specify this information by calling the macro `ssSetModelReferenceSampleTimeInheritanceRule` from `mdlInitializeSizes` or `mdlSetWorkWidths`. To use this macro:

1. Check whether the S-function calls any of these macros:
   - `ssGetSampleTime`
   - `ssGetInputPortSampleTime`
   - `ssGetOutputPortSampleTime`
   - `ssGetInputPortOffsetTime`
   - `ssGetOutputPortOffsetTime`
   - `ssGetInputPortSampleTimeIndex`
   - `ssGetOutputPortSampleTimeIndex`
2. Check for these in your S-function TLC code:
   - `LibBlockSampleTime`
   - `CompiledModel.SampleTime`
   - `LibBlockInputSignalSampleTime`
   - `LibBlockInputSignalOffsetTime`
   - `LibBlockOutputSignalSampleTime`
   - `LibBlockOutputSignalOffsetTime`
3. Depending on your search results, use `ssSetModelReferenceSampleTimeInheritanceRule` as indicated in this table.

If...	Use...	Example
None of the macros or functions are present, the S-function does not preclude the model from inheriting a sample time.	`ssSetModelReferenceSampleTimeInheritanceRule (S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`	

If...	Use...	Example

Any of the macros or functions are used for:
• Throwing errors if sample time is inherited, continuous, or constant
• Checking `ssIsSampleHit`
• Checking whether sample time is inherited in either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` before setting
	`ssSetModelReferenceSampleTimeInheritanceRule(S,USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`	Consider this `mdlOutputs` code:
		`static void mdlOutputs(SimStruct *S, int`
		`{`
		`  const real_T *u = (const real_T*)`
		`  ssGetInputPortSignal(S,0);`
		`  real_T  *y = ssGetOutputPortSig`
		`  y[0] = ssGetSampleTime(S,tid) * u[0]`
		`}`

If...	Use...	Example
The S-function uses its sample time for computing parameters, outputs, and so on.	`ssSetModelReferenceSampleTimeInheritanceRule(S, DISALLOW_SAMPLE_TIME_INHERITANCE)`	Consider the `mdlOutputs` code from the S-function example `sfun_multirate.c`:

```
static void mdlOutputs(SimStruct *S, int
{
    InputRealPtrsType enablePtrs;
    int               *enabled = ssGetIW
    int enableTid = ssGetInputPortSample
    int signalTid = ssGetInputPortSample
    real_T enableTs       = ssGetInputPo
    real_T enableTsOffset = ssGetInputPo
    if (enableTs == CONTINUOUS_SAMPLE_TI
        if (ssIsMajorTimeStep(S) && ssIs
            if (signalTid == enableTid |
                ssIsSpecialSampleHit(S,
                enablePtrs = ssGetInputP
                *enabled = (*enablePtrs[
            }
        }
    } else {
        int enableTid = ssGetInputPortSa
        if (ssIsSampleHit(S, enableTid,
            if (enableTid == signalTid |
                ssIsSpecialSampleHit(S,
                enablePtrs = ssGetInputP
                *enabled = (*enablePtrs[
            }
        }
    }

    if (ssIsSampleHit(S, signalTid, tid)
        InputRealPtrsType uPtrs  = ssGet
        real_T            signal = *uPtr
        int               i;

        for (i = 0; i < NOUTPUTS; i++) {
            int outTid = ssGetOutputPort
            if (outTid==signalTid ||
                ssIsSpecialSampleHit(S,
                real_T *y = ssGetOutputP
                *y = signal;
            }
        }
    }

} /* end mdlOutputs */
```

Note If an S-function does not set the `ssSetModelReferenceSampleTimeInheritanceRule` macro, by default the Simulink engine assumes that the S-function does not preclude the model containing that S-function from inheriting a sample time. However, the engine issues a warning indicating that the model includes S-functions for which this macro is not set.

You can use settings in the Configuration Parameters on the **Diagnostics > Sample Time** pane to control how the Simulink engine responds when it encounters S-functions that have unspecified sample time inheritance rules. Toggle the **Unspecified inheritability of sample time** diagnostic to `none`, `warning`, or `error`. The default is `warning`.

For information about Model block sample time inheritance, see “Referenced Model Sample Times”.

See Also

Related Examples

- “Specify S-Function Sample Times”
- “S-Functions That Support Expression Folding” on page 13-64
- “S-Functions That Specify Port Scope and Reusability” on page 13-73
- “S-Functions for Code Reuse” on page 13-81

S-Functions for Code Reuse

You can reuse the generated code for identical subsystems that occur in multiple instances within a model and across referenced models. For more information about code generation of subsystems for code reuse, see “Generate Subsystem Code as Separate Function and Files”.

S-Function Code Reuse Requirements

To use S-function for code reuse for a subsystem, the S-function must meet these requirements:

- The S-function must be inlined.
- Code generated from the S-function must not use static variables.
- The S-function must initialize its pointer work vector in only `mdlStart` and not before.
- The S-function must not be a sink that logs data to the workspace.
- The S-function must register its parameters as run-time parameters in `mdlSetWorkWidths`. For this purpose, the S-function must not use `ssWriteRTWParameters` in its `mdlRTW` function.
- The S-function must not be a device driver.

Additionally, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the ssSetOptions function. This flag indicates that your S-function meets the requirements for subsystem code reuse. If the flag is set and your S-function does not meet the requirements, the code generator does not generate a reusable function and you see a warning.

S-Functions Reused Across Models

You can place S-functions inside reusable library subsystems. In the S-function file, set both of these flags:

- `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the `ssSetOptions` function
- `ssSetSupportedForCodeReuseAcrossModels` set to `1` or `true`

If you use the `legacy_code` tool to generate your S-function, the S-function option `supportCodeReuseAcrossModels` specifies code reuse across models for the subsystem that contains the S-function.

Configure the S-function for code reuse across models only if the S-function does not access any model-specific data structures. This configuration is necessary because the generated code in `_sharedutils` folder is compiled separately from the generated code in `model_ert_rtw folder`. The reusable library subsystem code generated in the `_sharedutils` folder does not have access to types and macros that are declared in `model.h`. The reusable library subsystem code must be independent. For more information, see “Generate Reusable Code from Library Subsystems Shared Across Models”.

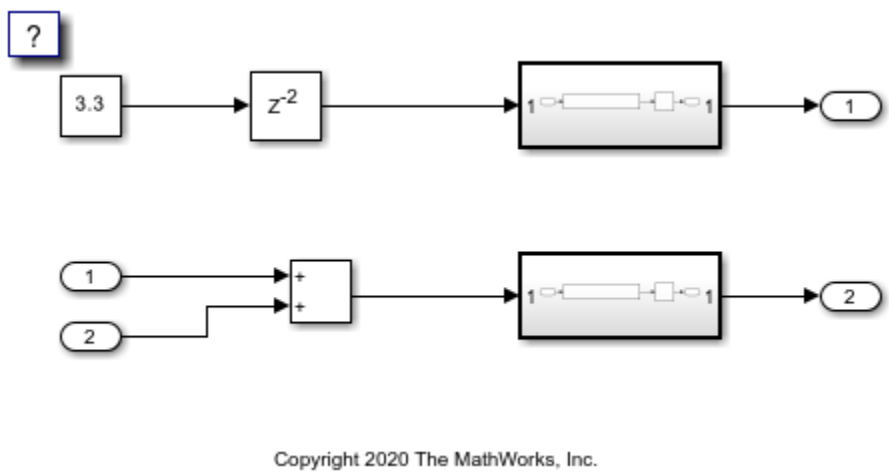
If your S-function uses custom functions defined in your external header files, add the “LibAddtoSystemCustomIncludes(system, incFileName)” function in your S-function's TLC file. For example, when you add `LibAddtoSystemCustomIncludes("company_math.h")` in your `.tlc` file, the reusable library subsystem code in the `_sharedutils` folder includes the `#company_math.h` header file.

1. Create a new MATLAB® working folder. In the MATLAB Command Window, copy these external code files to your current MATLAB working folder:

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','doubleIt.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','doubleIt.h'));
```

2. Open example model `rtwdemo_sfcn_rls`. This example model uses two instances of the reusable library subsystem `rtwdemo_sfcn_rls_lib`. The S-function `sfun_mySrc` inside the reusable library subsystem uses external files `doubleIt.c` and `doubleIt.h`.

```
rtwdemo_sfcn_rls;
```



3. Generate the S-function code by using legacy code tool. Use these commands in your MATLAB Command window:

```
def = legacy_code('initialize');
def.SFunctionName = 'sfun_mySrc';
def.SourceFiles = {'doubleIt.c'};
def.HeaderFiles = {'doubleIt.h'};
def.OutputFcnSpec = 'double y1 = doubleIt(double u1)';
```

4. Specify the S-function for code reuse:

```
def.Options.supportCodeReuseAcrossModels = true;
```

5. Generate the S-function code TLC block file. Then compile the S-function.

```
legacy_code('sfcn_tlc_generate',def);
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def);
```

```
### Start Compiling sfun_mySrc
    mex('-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\6\tp03c82afd_8d2d_45df_8671_53b0234eeff4\coderde
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('sfun_mySrc.c', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\6\tp03c82afd_8d2d_45df_8671_53b0
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling sfun_mySrc
### Exit
```

6. The code generator creates the S-function file `sfun_mySrc.c` and its TLC file `sfun_mySrc.tlc` in your MATLAB working folder. In the `sfun_mySrc.c` file, the code generator adds these specifications:

```
file = fullfile('sfun_mySrc.c');
rtwdemodbtype(file,'* All options have the form SS_OPTION_<name>', ...
    'ssSetSupportedForCodeReuseAcrossModels(S, 1);',1,1);
```

```
     * All options have the form SS_OPTION_<name> and are documented in
     * matlabroot/simulink/include/simstruc.h. The options should be
     * bitwise or'd together as in
     *    ssSetOptions(S, (SS_OPTION_name1 | SS_OPTION_name2))
     */
    ssSetOptions(S,
        SS_OPTION_USE_TLC_WITH_ACCELERATOR |
        SS_OPTION_CAN_BE_CALLED_CONDITIONALLY |
        SS_OPTION_EXCEPTION_FREE_CODE |
        SS_OPTION_WORKS_WITH_CODE_REUSE |
        SS_OPTION_SFUNCTION_INLINED_FOR_RTW |
        SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME
    );
    ssSetSupportedForCodeReuseAcrossModels(S, 1);
```

In the `sfun_mySrc.c`, the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag is set in the `ssSetOptions` function and `ssSetSupportedForCodeReuseAcrossModels` is set to true.

7. On the Simulink® toolstrip, open the **Embedded Coder** or **Simulink Coder** app.

8. Generate code for the model. On the **C Code** tab, click **Build**.

```
evalc('slbuild(''rtwdemo_sfcn_rls'')');
```

The code generator produces a reusable function that can be reused across models in the `_sharedutils` folder. Models in a model reference hierarchy share this `_sharedutils` folder to reuse code. For this example, the reusable function is generated in the `slprj/grt/_sharedutils/sfunc.c` file:

```
file = fullfile('slprj','grt','_sharedutils','sfunc.c');
rtwdemodbtype(file,'Output and update for atomic system', ...
    'rtb_sfun_mySrc',1,1);
```

```
 * Output and update for atomic system:
 *    'ReusableSubsystem' ('rtwdemo_sfcn_rls_lib:60')
 *    'ReusableSubsystem' ('rtwdemo_sfcn_rls_lib:60')
 */
void sfunc(real_T rtu_In1, B_sfunc_T *localB, DW_sfunc_T *localDW)
{
  /* local block i/o variables */
  real_T rtb_sfun_mySrc;
```

S-Functions Reused Within a Model

By default, S-functions inside a subsystem are not configured for code reuse within a model. You can specify the S-functions inside a subsystem for code reuse within a model. In the S-function file, set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the `ssSetOptions` function. This flag indicates that your S-function meets the requirements for subsystem code reuse and is configured for code

reuse. If the flag is set and your S-function does not meet the requirements, the code generator does not generate a reusable function and you see a warning.

1. Create a new MATLAB working folder and copy these external code files to your current MATLAB working folder:

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','doubleIt.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','doubleIt.h'));
```

2. For the example model `rtwdemo_sfcn_rls`, generate the S-function code by using legacy code tool. Do not include the S-function option `supportCodeReuseAcrossModels`. Use these commands in your MATLAB Command window:

```
def = legacy_code('initialize');
def.SFunctionName = 'sfun_mySrc';
def.SourceFiles = {'doubleIt.c'};
def.HeaderFiles = {'doubleIt.h'};
def.OutputFcnSpec = 'double y1 = doubleIt(double u1)';
```

3. Generate the S-function code TLC block file. Then compile the S-function.

```
legacy_code('sfcn_tlc_generate',def);
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def);
```

```
### Start Compiling sfun_mySrc
    mex('-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\6\tpe8dae8e1_b5b0_4655_9e33_b26f18c1e739\coderde
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('sfun_mySrc.c', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\6\tpe8dae8e1_b5b0_4655_9e33_b26
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling sfun_mySrc
### Exit
```

4. The code generator creates the S-function file `sfun_mySrc.c` and its TLC file `sfun_mySrc.tlc` in your MATLAB working folder.

```
file = fullfile('sfun_mySrc.c');
rtwdemodbtype(file,'* All options have the form SS_OPTION_<name>', ...
    'SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME',1,1);
```

```
     * All options have the form SS_OPTION_<name> and are documented in
     * matlabroot/simulink/include/simstruc.h. The options should be
     * bitwise or'd together as in
     *    ssSetOptions(S, (SS_OPTION_name1 | SS_OPTION_name2))
     */
    ssSetOptions(S,
        SS_OPTION_USE_TLC_WITH_ACCELERATOR |
        SS_OPTION_CAN_BE_CALLED_CONDITIONALLY |
        SS_OPTION_EXCEPTION_FREE_CODE |
        SS_OPTION_WORKS_WITH_CODE_REUSE |
        SS_OPTION_SFUNCTION_INLINED_FOR_RTW |
        SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME
```

In the `sfun_mySrc.c` file, the code generator adds `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the `ssSetOptions` function by default but does not include the `ssSetSupportedForCodeReuseAcrossModels` specification that prevents the S-function from being reused across models.

5. On the Simulink toolstrip, open the **Embedded Coder** or **Simulink Coder** app.

6. Generate code for the model. On the **C Code** tab, click **Build**.

```
evalc('slbuild(''rtwdemo_sfcn_rls'')');
```

The code generator produces a reusable function for the subsystem in the `rtwdemo_sfcn_rls.c` file, which is outside the `_sharedutils` folder. This code can be reused within the model but not across the model reference hierarchy:

```
file = fullfile('rtwdemo_sfcn_rls_grt_rtw','rtwdemo_sfcn_rls.c');
rtwdemodbtype(file,'void sfunc', ...
    'localDW->UnitDelay_DSTATE = rtb_sfun_mySrc;',1,1);
```

```
void sfunc_Init(DW_sfunc_T *localDW)
{
  /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
  localDW->UnitDelay_DSTATE = 0.0;
}

/*
 * Output and update for atomic system:
 *    '<Root>/ReusableSubsystem1'
 *    '<Root>/ReusableSubsystem2'
 */
void sfunc(real_T rtu_In1, B_sfunc_T *localB, DW_sfunc_T *localDW)
{
  /* local block i/o variables */
  real_T rtb_sfun_mySrc;

  /* UnitDelay: '<S1>/Unit Delay' */
  localB->UnitDelay = localDW->UnitDelay_DSTATE;

  /* S-Function (sfun_mySrc): '<S1>/sfun_mySrc' */
  rtb_sfun_mySrc = doubleIt(rtu_In1);

  /* Update for UnitDelay: '<S1>/Unit Delay' */
  localDW->UnitDelay_DSTATE = rtb_sfun_mySrc;
```

See Also

Related Examples

- “S-Functions That Support Expression Folding” on page 13-64
- “S-Functions That Specify Port Scope and Reusability” on page 13-73
- “S-Functions That Specify Sample Time Inheritance Rules” on page 13-77

S-Functions for Multirate Multitasking Environments

In this section...

"About S-Functions for Multirate Multitasking Environments" on page 13-86
"Rate Grouping Support in S-Functions" on page 13-86
"Create Multitasking, Multirate, Port-Based Sample Time S-Functions" on page 13-86

About S-Functions for Multirate Multitasking Environments

S-functions can be used in models with multiple sample rates and deployed in multitasking target environments. Likewise, S-functions themselves can have multiple rates at which they operate. The code generator produces code for multirate multitasking models using an approach called *rate grouping*. In code generated for ERT-based targets, rate grouping generates separate `model_step` functions for the base rate task and each subrate task in the model. Although rate grouping is a code generation feature found in ERT targets only, your S-functions can use it in other contexts when you code them as explained below.

Rate Grouping Support in S-Functions

To take advantage of rate grouping, you must inline your multirate S-functions if you have not done so. You need to follow certain Target Language Compiler protocols to exploit rate grouping. Coding TLC to exploit rate grouping does not prevent your inlined S-functions from functioning properly in GRT. Likewise, your inlined S-functions will still generate valid ERT code even if you do not make them rate-grouping-compliant. If you do so, however, they will generate more efficient code for multirate models.

For instructions and examples of Target Language Compiler code illustrating how to create and upgrade S-functions to generate rate-grouping-compliant code, see "Rate Grouping Compliance and Compatibility Issues" on page 71-12.

For each multirate S-function that is not rate grouping-compliant, the code generator issues the following warning when you build:

```
Warning:  Simulink Coder: Code of output function for multirate block
'<Root>/S-Function' is guarded by sample hit checks rather than being rate
grouped. This will generate the same code for all rates used by the block,
possibly generating dead code. To avoid dead code, you must update the TLC
file for the block.
```

You will also find a comment such as the following in code generated for each noncompliant S-function:

```
/* Because the output function of multirate block
   <Root>/S-Function is not rate grouped,
   the following code might contain unreachable blocks of code.
   To avoid this, you must update your block TLC file. */
```

The words "update function" are substituted for "output function" in these warnings.

Create Multitasking, Multirate, Port-Based Sample Time S-Functions

The following instructions show how to support both data determinism and data integrity in multirate S-functions. They do not cover cases where there is no determinism nor integrity. Support for frame-based processing does not affect the requirements.

Note The slow rates must be multiples of the fastest rate. The instructions do not apply when two rates being interfaced are not multiples or when the rates are not periodic.

Rules for Properly Handling Fast-to-Slow Transitions

The rules that multirate S-functions should observe for inputs are

- The input should only be read at the rate that is associated with the input port sample time.
- Generally, the input data is written to `DWork`, and the `DWork` can then be accessed at the slower (downstream) rate.

The input can be read at every sample hit of the input rate and written into `DWork` memory, but this `DWork` memory cannot then be directly accessed by the slower rate. `DWork` memory that will be read by the slow rate must only be written by the fast rate when there is a *special sample hit*. A special sample hit occurs when both this input port rate and rate to which it is interfacing have a hit. Depending on their requirements and design, algorithms can process the data in several locations.

The rules that multirate S-functions should observe for outputs are

- The output should not be written by a rate other than the rate assigned to the output port, except in the optimized case described below.
- The output should always be written when the sample rate of the output port has a hit.

If these conditions are met, the S-Function block can specify that the input port and output port can both be made local and reusable.

You can include an optimization when little or no processing needs to be done on the data. In such cases, the input rate code can directly write to the output (instead of by using DWork) when there is a special sample hit. If you do this, however, you must declare the outport port to be *global* and *not reusable*. This optimization results in one less `memcpy` but does introduce nonuniform processing requirements on the faster rate.

Whether you use this optimization or not, the most recent input data, as seen by the slower rate, is the value when both the faster and slower rate had their hits (and possible earlier input data as well, depending on the algorithm). Subsequent steps by the faster rate and the associated input data updates are not seen by the slower rate until the next hit for the slow rate occurs.

Pseudocode Examples of Fast-to-Slow Rate Transition

The pseudocode below abstracts how you should write your C MEX code to handle fast-to-slow transitions, illustrating with an input rate of 0.1 second driving an output rate of one second. A similar approach can be taken when inlining the code. The block has following characteristics:

- File: `sfun_multirate_zoh.c`, Equation: `y = u(tslow)`
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSepcialSampleHit("1")) {
        DWork = u;
    }
}
```

```
if (ssIsSampleHit("1")) {
    y = DWork;
}
```

An alternative, slightly optimized approach for simple algorithms:

- Input: local and reusable
- Output: global and not reusable because it needs to persist between special sample hits
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        y = u;
    }
}
```

Example adding a simple algorithm:

- File: `sfun_multirate_avg.c`; Equation: `y = average(u)`
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

  (Assume `DWork[0:10]` and `DWork[mycounter]` are initialized to zero)

```
OutputFcn
if (ssIsSampleHit(".1")) {
    /* In general, processing on 'u' could be done here,
        it runs on every hit of the fast rate. */
    DWork[DWork[mycounter]++] = u;
    if (ssIsSpecialSampleHit("1")) {
    /* In general, processing on DWork[0:10] can be done
       here, but it does cause the faster rate to have
       nonuniform processing requirements (every 10th hit,
       more code needs to be run).*/
        DWork[10] = sum(DWork[0:9])/10;
        DWork[mycounter] = 0;
    }
}
if (ssIsSampleHit("1")) {
    /* Processing on DWork[10] can be done here before
       outputing. This code runs on every hit of the
       slower task. */
    y = DWork[10];
}
```

Rules for Properly Handling Slow-to-Fast Transitions

When output rates are faster than input rates, input should only be read at the rate that is associated with the input port sample time, observing the following rules:

- Always read input from the update function.
- Use no special sample hit checks when reading input.
- Write the input to a DWork.
- When there is a special sample hit between the rates, copy the DWork into a second DWork in the output function.
- Write the second DWork to the output at every hit of the output sample rate.

The block can request that the input port be made local but it cannot be set to reusable. The output port can be set to local and reusable.

As in the fast-to-slow transition case, the input should not be read by a rate other than the one assigned to the input port. Similarly, the output should not be written to at a rate other than the rate assigned to the output port.

An optimization can be made when the algorithm being implemented is only required to run at the slow rate. In such cases, you use only one DWork. The input still writes to the DWork in the update function. When there is a special sample hit between the rates, the output function copies the same DWork directly to the output. You must set the output port to be global and not reusable in this case. This optimization results in one less `memcpy` operation per special sample hit.

In either case, the data that the fast rate computations operate on is always delayed, that is, the data is from the previous step of the slow rate code.

Pseudocode Examples of Slow-to-Fast Rate Transition

The pseudocode below abstracts what your S-function needs to do to handle slow-to-fast transitions, illustrating with an input rate of one second driving an output rate of 0.1 second. The block has following characteristics:

- File: `sfun_multirate_delay.c`, Equation: `y = u(tslow-1)`
- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
        DWork[1] = DWork[0];
    }
    y = DWork[1];
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork[0] = u;
}
```

An alternative, optimized approach can be used by some algorithms:

- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: global and not reusable because it needs to persist between special sample hits.
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
        y = DWork;
    }
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork = u;
}
```

Example adding a simple algorithm:

- File: `sfun_multirate_modulate.c`, Equation: `y = sin(tfast) + u(tslow-1)`
- Input: Set to local, will be local if output/update are combined (an ERT feature) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
    /* Processing not likely to be done here. It causes
     * the faster rate to have nonuniform processing
     * requirements (every 10th hit, more code needs to
     * be run).*/
        DWork[1] = DWork[0];
    }
    /* Processing done at fast rate */
    y = sin(ssGetTaskTime(".1")) + DWork[1];
}
UpdateFcn
if (ssIsSampleHit("1")) {
    /* Processing on 'u' can be done here. There is a delay of
       one slow rate period before the fast rate sees it.*/
    DWork[0] = u;}
```

Write Noninlined S-Function

A noninlined S-function is a C or C++ MEX S-function that is treated identically by the Simulink engine and by the generated code. You implement your algorithm once according to the S-function API. The Simulink engine and generated code call the S-function routines (for example, `mdlOutputs`) during model execution.

Noninlined S-functions are identified by the absence of an *`sfunction`*`.tlc` file for your S-function. The file name varies depending on your platform. For example, on a 64-bit Microsoft® Windows® system, the file name is *`sfunction.mexw64`*. In the MATLAB Command Window, type `mexext` to see which extension your system uses.

Guidelines for Writing Noninlined S-Functions

- The MEX-file cannot call MATLAB functions.
- If the MEX-file uses functions in the MATLAB External Interface libraries, include the header file `cg_sfun.h` instead of `mex.h` or `simulink.c`. For the header file `cg_sfun.h`, at the end of your S-function, include these lines:

```
#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c"  /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"   /* Code generation registration function */
#endif
```

- Use only the MATLAB API function that the code generator supports. The supported API functions are:
  - `mxGetDoubles`
  - `mxGetEps`
  - `mxGetInf`
  - `mxGetM`
  - `mxGetN`
  - `mxGetNaN`
  - `mxGetPr` (Not recommended. Use `mxGetDoubles` instead.)
  - `mxGetScalar`
  - `mxGetString`
  - `mxIsEmpty`
  - `mxIsFinite`
  - `mxIsInf`
- MEX library calls are not supported in generated code. To use such calls in the MEX-file and not in the generated code, add the following condition:

```
#ifdef MATLAB_MEX_FILE
#endif
```

- Use only full matrices that contain only real data.
- Do not specify a return value for calls to `mxGetString`. If you do specify a return value, the MEX-file does not compile. Instead, use the second input argument of the function, which returns a pointer to a character vector.

- Use the correct `#define` *`s-function_name`* statement. The S-function name that you specify must match the S-function file name.
- If possible, use the data types `real_T` and `int_T` instead of `double` and `int`. The data types `real_T` and `int_T` are more generic and can be used in multiple environments.
- Provide the build process with the names of the modules used to build the S-function. Use a template make file, the `set_param` function, or the `S-function modules` field of the S-Function block parameters dialog box. For example, suppose that you build your S-function with this command:

  ```
  mex sfun_main.c sfun_module1.c sfun_module2.c
  ```

  You can then use the following call to `set_param` to include the required modules:

  ```
  set_param(sfun_block, "SFunctionModules","sfun_module1 sfun_module2")
  ```

  When you are ready to generate code, force the code generator to rebuild the top model. For more information, see “Control Regeneration of Top Model Code”.

Noninlined S-Function Parameter Type Limitations

Parameters to noninlined S-functions can be of the following types only:

- Double precision
- Characters in scalars, vectors, or 2-D matrices

For more flexibility in the type of parameters that you can supply to S-functions or the operations in the S-function, inline your S-function and consider using an `mdlRTW` S-function routine.

Use of other functions from the MATLAB `matrix.h` API or other MATLAB APIs, such as `mex.h` and `mat.h`, are not supported. If you call unsupported APIs from an S-function source file, compiler errors occur. For details on supported MATLAB API functions, see the files *`matlabroot`*`/rtw/c/src/rt_matrx.h` and *`matlabroot`*`/rtw/c/src/rt_matrx.c`

If you use `mxGetPr` on an empty matrix, the function does not return `NULL`. It returns a random value. Therefore, you must protect calls to `mxGetPr` by using `mxIsEmpty`.

See Also

Related Examples

- “Inlined and Noninlined S-Function Code”
- “S-Functions and Code Generation”

Write Wrapper S-Function and TLC Files

Create S-functions that work seamlessly with the Simulink and code generator products by using the wrapper concept. You can:

- Interface your algorithms in Simulink models by writing MEX S-function wrappers (*sfunction*.mex).
- Direct the code generator to insert your algorithm into the generated code by creating a TLC S-function wrapper (*sfunction*.tlc).

MEX S-Function Wrapper

Creating S-functions by using an S-function wrapper enables you to insert C/C++ code algorithms in Simulink models and the generated code with little or no change to your original C/C++ function. A MEX S-function wrapper is an S-function that calls code, which resides in another module.

Note Use a MEX S-function wrapper only in the MATLAB version in which you created the wrapper.

Suppose that you have an algorithm (that is, a C function) called `my_alg` that resides in the file `my_alg.c`. You can integrate `my_alg` into a Simulink model by creating a MEX S-function wrapper (for example, `wrapsfcn.c`). A Simulink model can then call `my_alg` from an S-Function block. The Simulink S-function contains a set of empty functions that the Simulink engine requires for various API related purposes. For example, although only `mdlOutputs` calls `my_alg`, the engine calls `mdlTerminate`, even though this S-function routine performs no action.

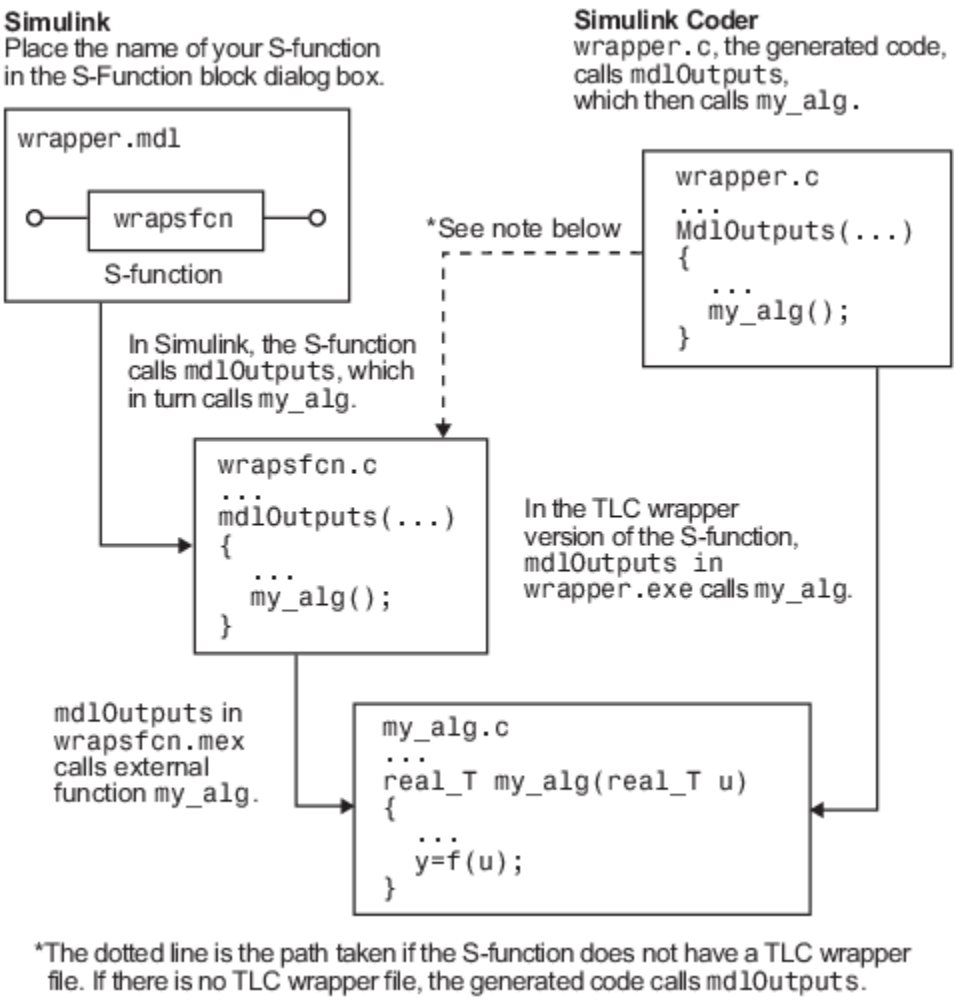
You can embed the call to `my_alg` in the generated code by creating a TLC S-function wrapper (for example, `wrapsfcn.tlc`). You can eliminate the empty function calls. You can avoid the overhead of executing the `mdlOutputs` function and you can then eliminate the `my_alg` function.

Wrapper S-functions are useful when you are creating algorithms that are procedural or when you are integrating legacy code into a Simulink model. If you want to create code that is:

- Interpretive in nature (that is, highly parameterized by operating modes)
- Heavily optimized (that is, no extra tests to decide what mode the code is operating in)

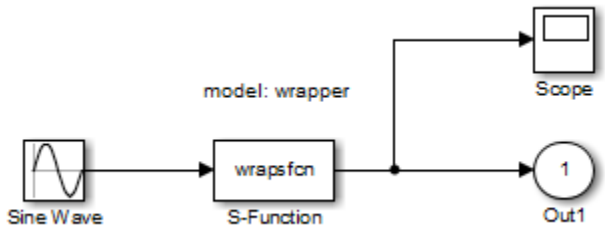
then you must create a fully inlined TLC file for your S-function.

The next figure shows the wrapper S-function concept.



Using an S-function wrapper to import algorithms into your Simulink model means that the S-function serves as an interface that calls your C/C++ algorithms from `mdlOutputs`. You can quickly integrate large standalone C/C++ programs into your model without having to change the code.

This sample model includes an S-function wrapper.



Two files are associated with the `wrapsfcn` block: the S-function wrapper and the C/C++ code that contains the algorithm. The first three statements:

1. Define the name of the S-function (what you enter in the Simulink S-Function block dialog box).

2. Specify that the S-function is using the level 2 format.
3. Provide access to the `SimStruct` data structure. `SimStruct` contains pointers to data used during simulation and code generation and defines macros that store data in and retrieve data from the `SimStruct`.

```
#define S_FUNCTION_NAME wrapsfcn
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

extern real_T my_alg(real_T u);  /* Declare my_alg as extern */

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes( S, 1);
}

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    *y = my_alg(*uPtrs[0]); /* Call my_alg in mdlOutputs */
 }
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

For more information, see “Templates for C S-Functions”.

The S-function routine `mdlOutputs` contains a function call to `my_alg`, which is the C function containing the algorithm that the S-function performs. For `my_alg.c`, the code is:

```
#ifdef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
real_T my_alg(real_T u)
{
return(u * 2.0);
}
```

For more information, see “Manage Build Process File Dependencies”.

The wrapper S-function `wrapsfcn` calls `my_alg`, which computes `u * 2.0`. To build `wrapsfcn.mex`, use this command:

```
mex wrapsfcn.c my_alg.c
```

TLC S-Function Wrapper

A TLC S-function wrapper is a TLC file that specifies how the code generator calls your code. For example, you can inline the call to `my_alg` in the `mdlOutputs` section of the generated code. In the “MEX S-Function Wrapper” example, the call to `my_alg` is embedded in the `mdlOutputs` section as:

```
*y = my_alg(*uPtrs[0]);
```

When you are creating a TLC S-function wrapper, the goal is to embed the same type of call in the generated code.

Look at how the code generator executes S-functions that are not inlined. A noninlined S-function is identified by the absence of the file *`sfunction`*`.tlc` and the existence of *`sfunction`*`.mex`. When generating code for a noninlined S-function, the code generator produces a call to `mdlOutputs` through a function pointer that, in this example, then calls `my_alg`.

The wrapper example contains one S-function, `wrapsfcn.mex`. You must compile and link an additional module, `my_alg`, with the generated code. At the MATLAB command prompt, enter:

```
set_param('wrapper/S-Function','SFunctionModules','my_alg')
```

Code Overhead for Noninlined S-Functions

The code generated when using `grt.tlc` as the system target file *without* `wrapsfcn.tlc` is:

```
<Generated code comments for wrapper model with noninlined wrapsfcn S-function>

#include <math.h>
#include <string.h>
#include "wrapper.h"
#include "wrapper.prm"

/* Start the model */
void mdlStart(void)
{
  /* (start code not required) */
}

/* Compute block outputs */
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
```

```
  {
    /* Noninlined S-functions create a SimStruct object and
     * generate a call to S-function routine mdlOutputs
     */
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}

/* Perform model update */
void mdlUpdate(int_T tid)
{
  /* (update code not required) */
}

/* Terminate function */
void mdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
  {
/* Noninlined S-functions require a SimStruct object and
     * the call to S-function routine mdlTerminate
     */
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}

#include "wrapper.reg"

/* [EOF] wrapper.c */
```

The `wrapper.reg` generated file contains the initialization of the `SimStruct` for the wrapper S-Function block. There is one child `SimStruct` for each S-Function block in your model. You can significantly reduce this overhead by creating a TLC wrapper for the S-function.

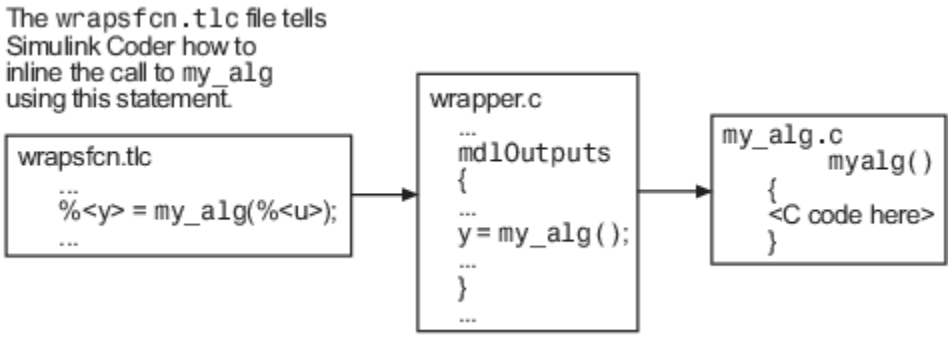
Inline a Wrapper S-Function

The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```
SimStruct *rts = ssGetSFunction(rtS, 0);
sfcnOutputs(rts, tid);
```

This call has computational overhead associated with it. The Simulink engine creates a `SimStruct` data structure for the S-Function block. The code generator constructs a call through a function pointer to execute `mdlOutputs`, then `mdlOutputs` calls `my_alg`. By inlining the call to your C/C++ algorithm, `my_alg`, you can eliminate both the `SimStruct` and the extra function call, thereby improving the efficiency and reducing the size of the generated code.

Inlining a wrapper S-function requires an *`sfunction`*`.tlc` file for the S-function. The TLC file must contain the function call to `my_alg`. The figure shows the relationships between the algorithm, the wrapper S-function, and the *`sfunction`*`.tlc` file.



To inline the call to `my_alg`, place your function call in an *`sfunction`*`.tlc` file with the same name as the S-function (in this example, `wrapsfcn.tlc`). The Target Language Compiler overrides the default method of placing calls to your S-function in the generated code.

This code is the TLC file `wrapsfcn.tlc` that inlines `wrapsfcn.c`:

```
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example inlined tlc file for S-function wrapsfcn.c
%%

%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup ======================================================
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern real_T my_alg(real_T u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
    extern real_T my_alg(real_T u); /* This line is placed in wrapper.h */
  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs ============================================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
  %% The following line is expanded and placed in mdlOutputs within wrapper.c
  %<y> = my_alg(%<u>);

%endfunction %% Outputs
```

The first section of this code inlines the `wrapsfcn` S-Function block and generates the code in C:

```
%implements "wrapsfcn" "C"
```

The next task is to inform the code generator that the routine `my_alg` must be declared as external in the generated `wrapper.h` file for any `wrapsfcn` S-Function blocks in the model. Do this declaration once for all `wrapsfcn` S-Function blocks by using the `BlockTypeSetup` function. In this function, you direct the Target Language Compiler to create a buffer and cache the `my_alg` as `extern` in the `wrapper.h` generated header file.

The final step is the inlining of the call to the function `my_alg`. The `Outputs` function inlines the call. In this function, you access the block input and output and place a direct call to `my_alg`. The call is embedded in `wrapper.c`.

The Inlined Code

The code generated when you inline your wrapper S-function is similar to the default generated code. The `mdlTerminate` function does not contain a call to an empty function and the `mdlOutputs` function now directly calls `my_alg`.

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = my_alg(rtB.Sin); /* Inlined call to my_alg */

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}
```

`wrapper.reg` does not create a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly, eliminating over 1 KB of memory usage.

See Also

Related Examples

- “Write Noninlined S-Function”
- “Write Fully Inlined S-Functions”

Write Fully Inlined S-Functions

A fully inlined S-function builds your algorithm (block) into generated code that you cannot distinguish from a built-in block. Typically, a fully inlined S-function requires you to implement your algorithm twice: once for the Simulink model (C/C++ MEX S-function) and once for code generation (TLC file).

Using the example in “Write Wrapper S-Function and TLC Files”, you can eliminate the call to `my_alg` entirely by specifying the explicit code (that is, `2.0 * u`) in `wrapsfcn.tlc`. While this can improve performance, if you are working with a large amount of C/C++ code, the task can be lengthy. You also have to maintain your algorithm in two places, the C/C++ S-function itself and the corresponding TLC file. Consider whether the performance gains might outweigh the disadvantages. To inline the algorithm used in this example, in the `Outputs` section of your `wrapsfcn.tlc` file, instead of writing:

```
%<y> = my_alg(%<u>);
```

Use:

```
%<y> = 2.0 * %<u>;
```

This code is the code produced in `mdlOutputs`:

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = 2.0 * rtB.Sin; /* Explicit embedding of algorithm */

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}
```

The Target Language Compiler replaces the call to `my_alg` with the algorithm itself.

Multiport S-Function

A more advanced multiport inlined S-function example is `sfun_multiport.c` and `sfun_multiport.tlc`. This S-function illustrates how to create a fully inlined TLC file for an S-function that contains multiple ports.

Guidelines for Writing Inlined S-Functions

- Consider using the block property `RTWdata` (see “S-Function RTWdata”). This property is a structure of character vectors that you can associate with a block. The code generator saves the structure with the model in the *`model`*`.rtw` file and makes the `.rtw` file more readable. For example in the MATLAB Command Window, suppose you enter these commands:

  ```
  mydata.field1 = 'information for field1';
  mydata.field2 = 'information for field2';
  set_param(sfun_block, 'RTWdata', mydata);
  ```

  The `.rtw` file that the code generator produces for the block includes the comments specified in the structure `mydata`.

- Consider using the `mdlRTW` function to inline your C MEX S-function in the generated code for:
  - Renaming tunable parameters in the generated code.
  - Introducing non-tunable parameters into a TLC file.

See Also

Related Examples

- “Write Fully Inlined S-Functions with mdlRTW Routine”

Write Fully Inlined S-Functions with mdlRTW Routine

You can inline more complex S-functions by using the S-function `mdlRTW` routine. The `mdlRTW` routine provides the code generation process with more information about how the S-function is to be inlined by creating a parameter record of a non-tunable parameter for use with a TLC file. The `mdlRTW` routine places information in the *`model`*`.rtw` file. The `mdlRTW` function is described in the text file *`matlabroot`*`/simulink/src/sfuntmpl_doc.c`.

To use the `mdlRTW` function, take steps to create a direct-index lookup S-function. Lookup tables are collections of ordered data points of a function. Typically, these tables use some interpolation scheme to approximate values of the associated function between known data points. To incorporate the example lookup table algorithm into a Simulink model, the first step is to write an S-function that executes the algorithm in `mdlOutputs`. To produce the most efficient code, the next step is to create a corresponding TLC file to eliminate computational overhead and improve the speed of the lookup computations.

The Simulink product provides support for general-purpose lookup 1-D, 2-D, and n-D algorithms. You can use these algorithms as they are or create a custom lookup table S-function to fit your requirements. You can create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file (see “Target Language Compiler” for more details). You can:

- Error check of S-function parameters.
- Cache information for the S-function that does not change during model execution.
- Use the `mdlRTW` function to customize the code generator to produce the optimal code for a given set of block parameters.
- Create a `TLC` file for an S-function that either fully inlines the lookup table code or calls a wrapper function for the lookup table algorithm.

S-Function RTWdata

`RTWdata` is a property of blocks, which can be used by the Target Language Compiler when inlining an S-function. `RTWdata` is a structure of character vectors that you can attach to a block. `RTWdata` is saved with the model and placed in the *`model`*`.rtw` file when you generate code. For example, this set of MATLAB commands:

```
mydata.field1 = 'information for field1';
mydata.field2 = 'information for field2';
set_param(gcb,'RTWdata',mydata)
get_param(gcb,'RTWdata')
```

produces this result:

```
ans =

    field1: 'information for field1'
    field2: 'information for field2'
```

The information for the associated S-Function block inside the *`model`*`.rtw` file is:

```
Block {
  Type                    "S-Function"
  RTWdata {
    field1                  "information for field1"
    field2                  "information for field2"
  }
```

Note `RTWdata` is saved in the model file for S-functions that are not linked to a library. However, `RTWdata` is **not persistent** for S-Function blocks that are linked to a library.

Direct-Index Lookup Table Algorithm

The 1-D lookup table block provided in the Simulink library uses interpolation or extrapolation when computing outputs. In this example, you create a lookup table that directly indexes the output vector (*y*-data vector) based on the current input (*x*-data) point.

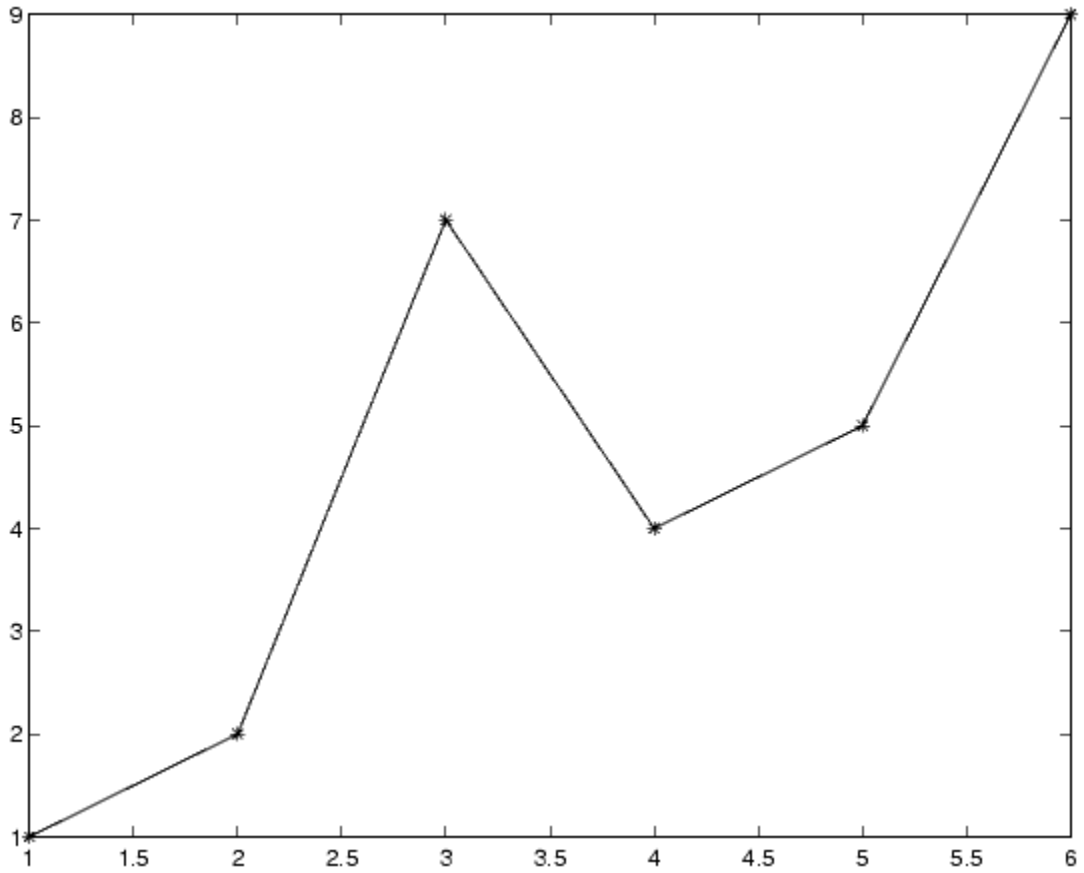
This direct 1-D lookup example computes an approximate solution *p(x)*to a partially known function *f(x)* at *x=x0*, given data point pairs *(x,y)* in the form of an *x*-data vector and a *y*-data vector. For a given data pair (for example, the *i'th* pair), *y_i = f(x_i)*. It is assumed that the *x*-data values are monotonically increasing. If *x0* is outside the range of the *x*-data vector, the first or last point is returned.

The parameters to the S-function are:

```
XData, YData, XEvenlySpaced
```

`XData` and `YData` are double vectors of equal length representing the values of the unknown function. `XDataEvenlySpaced` is a scalar, `0.0` for false and `1.0` for true. If the `XData` vector is evenly spaced, `XDataEvenlySpaced` is `1.0` and more efficient code is generated.

The graph shows how the parameters `XData=[1:6]`and `YData=[1,2,7,4,5,9]` are handled. For example, if the input (*x*-value) to the S-Function block is 3, the output (*y*-value) is 7.



Direct-Index Lookup Table Example

Improve the lookup table by inlining a direct-index S-function with a TLC file. This direct-index lookup table S-function does not require a TLC file. The example uses a TLC file for the direct-index lookup table S-function to reduce the code size and increase efficiency of the generated code.

Implementation of the direct-index algorithm with an inlined TLC file requires the S-function main module, `sfun_directlook.c` and a corresponding `lookup_index.c` module. The `lookup_index.c` module contains the `GetDirectLookupIndex` function that is used to locate the index in the `XData` for the current `x` input value when the `XData` is unevenly spaced. The `GetDirectLookupIndex` routine is called from the S-function and the generated code. The example uses the wrapper concept for sharing C/C++ code between Simulink MEX-files and the generated code.

If the `XData` is evenly spaced, then both the S-function main module and the generated code contain the lookup algorithm to compute the *y*-value of a given *x*-value because the algorithm is short.

The inlined TLC file is `sfun_directlook.tlc`, which is used to either perform a wrapper call or embed the optimal C/C++ code for the S-function. (See the example in “mdlRTW Usage”).

Error Handling

In `sfun_directlook.tlc`, the `mdlCheckParameters` routine verifies that:

- The new parameter settings are valid.
- `XData` and `YData` are vectors of the same length containing real, finite numbers.
- `XDataEvenlySpaced` is a scalar.
- The `XData` vector is a monotonically increasing vector and evenly spaced.

The `mdlInitializeSizes` function explicitly calls `mdlCheckParameters` after it verifies the number of parameters passed to the S-function. After the Simulink engine calls `mdlInitializeSizes`, it then calls `mdlCheckParameters` whenever you change the parameters or reevaluate them.

User Data Caching

In `sfun_directlook.tlc`, the `mdlStart` routine shows how to cache information that does not change during the simulation or while the generated code is executing. The example caches the value of the `XDataEvenlySpaced` parameter in `UserData`, a field of the `SimStruct`. The following line in `mdlInitializeSizes` instructs the Simulink engine to disallow changes to `XDataEvenlySpaced`.

```
ssSetSFcnParamTunable(S, iParam, SS_PRM_NOT_TUNABLE);
```

During execution, `mdlOutputs` accesses the value of `XDataEvenlySpaced` from `UserData` rather than calling the `mxGetPr` MATLAB API function.

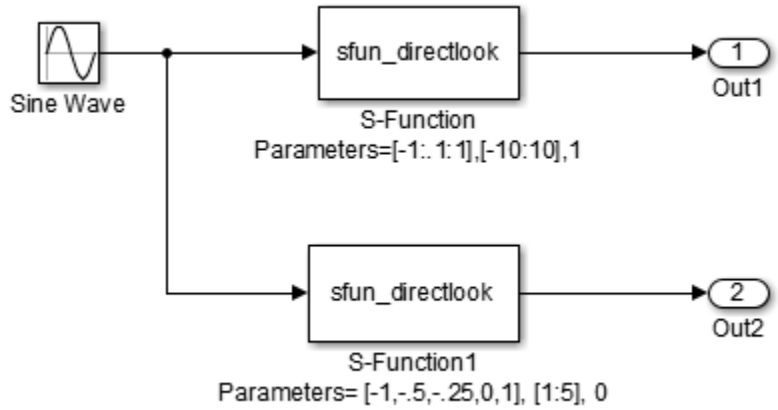
mdlRTW Usage

The code generator calls the `mdlRTW` routine while generating the *`model`*`.rtw` file. To produce optimal code for your Simulink model, you can add information to the *`model`*`.rtw` file about the mode in which your S-Function block is operating.

The example adds parameter settings to the *`model`*`.rtw` file. The parameter settings do not change during execution. In this case, the `XDataEvenlySpaced` S-function parameter cannot change during execution (`ssSetSFcnParamTunable` was specified as false (`0`) for it in `mdlInitializeSizes`). The parameter setting (`XSpacing`) uses the function `ssWriteRTWParamSettings`.

Because `xData` and `yData` are registered as run-time parameters in `mdlSetWorkWidths`, the code generator writes to the *`model`*`.rtw` file automatically.

Before examining the S-function and the inlined TLC file, consider the generated code for this model.



The model uses evenly spaced `XData` in the top S-Function block and unevenly spaced `XData` in the bottom S-Function block. When creating this model, specify the following commands for each S-Function block.

```
set_param('sfun_directlook_ex/S-Function','SFunctionModules','lookup_index')
set_param('sfun_directlook_ex/S-Function1','SFunctionModules','lookup_index')
```

The build process uses the module `lookup_index.c` when creating the executable.

When generating code for this model, the code generator uses the S-function `mdlRTW` method to generate a *`model`*`.rtw` file with the value `EvenlySpaced` for the `XSpacing` parameter for the top S-Function block and the value `UnEvenlySpaced` for the `XSpacing` parameter for the bottom S-Function block. The TLC-file uses the value of `XSpacing` to determine what algorithm to include in the generated code. The generated code contains the lookup algorithm when the `XData` is evenly spaced, but calls the `GetDirectLookupIndex` routine when the `XData` is unevenly spaced. The generated *`model`*`.c` or *`model`*`.cpp` code for the lookup table example model is similar to this code:

```
/*
 * sfun_directlook_ex.c
 *
 * Code generation for Simulink model
 * "sfun_directlook_ex.slx".
 *
...
 */

#include "sfun_directlook_ex.h"
#include "sfun_directlook_ex_private.h"

/* External outputs (root outports fed by signals with auto storage) */
ExtY_sfun_directlook_ex_T sfun_directlook_ex_Y;

/* Real-time model */
RT_MODEL_sfun_directlook_ex_T sfun_directlook_ex_M_;
RT_MODEL_sfun_directlook_ex_T *const sfun_directlook_ex_M =
  &sfun_directlook_ex_M_;

/* Model output function */
void sfun_directlook_ex_output(void)
{
  /* local block i/o variables */
  real_T rtb_SFunction;
  real_T rtb_SFunction1;

  /* Sin: '<Root>/Sine Wave' */
  rtb_SFunction1 = sin(sfun_directlook_ex_M->Timing.t[0]);
```

```
/* Code that is inlined for the top S-function block in the
   * sfun_directlook_ex model
   */
  /* S-Function (sfun_directlook): '<Root>/S-Function' */
  {
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction_YData;
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction1 <= xData[0] ) {
      rtb_SFunction = yData[0];
    } else if (rtb_SFunction1 >= yData[20] ) {
      rtb_SFunction = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction1 - xData[0] ) / spacing );
      rtb_SFunction = yData[idx];
    }
  }

  /* Outport: '<Root>/Out1' */
  sfun_directlook_ex_Y.Out1 = rtb_SFunction;

/* Code that is inlined for the bottom S-function block in the
   * sfun_directlook_ex model
   */
  /* S-Function (sfun_directlook): '<Root>/S-Function1' */
  {
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction1_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction1_YData;
    int_T idx;
    idx = GetDirectLookupIndex(xData, 5, rtb_SFunction1);
    rtb_SFunction1 = yData[idx];
  }

  /* Outport: '<Root>/Out2' */
  sfun_directlook_ex_Y.Out2 = rtb_SFunction1;
}

/* Model update function */
void sfun_directlook_ex_update(void)
{
  /* signal main to stop simulation */
  {                                   /* Sample time: [0.0s, 0.0s] */
    if ((rtmGetTFinal(sfun_directlook_ex_M)!=-1) &&
        !((rtmGetTFinal(sfun_directlook_ex_M)-sfun_directlook_ex_M->Timing.t[0])
          > sfun_directlook_ex_M->Timing.t[0] * (DBL_EPSILON))) {
      rtmSetErrorStatus(sfun_directlook_ex_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++sfun_directlook_ex_M->Timing.clockTick0)) {
    ++sfun_directlook_ex_M->Timing.clockTickH0;
  }

  sfun_directlook_ex_M->Timing.t[0] = sfun_directlook_ex_M->Timing.clockTick0 *
    sfun_directlook_ex_M->Timing.stepSize0 +
    sfun_directlook_ex_M->Timing.clockTickH0 *
    sfun_directlook_ex_M->Timing.stepSize0 * 4294967296.0;
}
...
```

See Also

Related Examples

- “Write Fully Inlined S-Functions”

What is the Target Language Compiler?

- "Target Language Compiler Basics" on page 14-2
- "Why Use the Target Language Compiler?" on page 14-6
- "The Advantages of Inlining S-Functions" on page 14-8

Target Language Compiler Basics

Target Language Compiler Overview

Target Language Compiler (TLC) is an integral part of the code generator. It enables you to customize generated code. Through customization, you can produce platform-specific code, or you can incorporate your own algorithmic changes for performance, code size, or compatibility with existing methods.

The TLC includes:

- Files corresponding to a subset of the provided Simulink blocks.
- Files for model-wide information that specify header and parameter information.

The TLC files are ASCII files that explicitly control the way that code is generated. By editing a TLC file, you can alter the way that the code is generated.

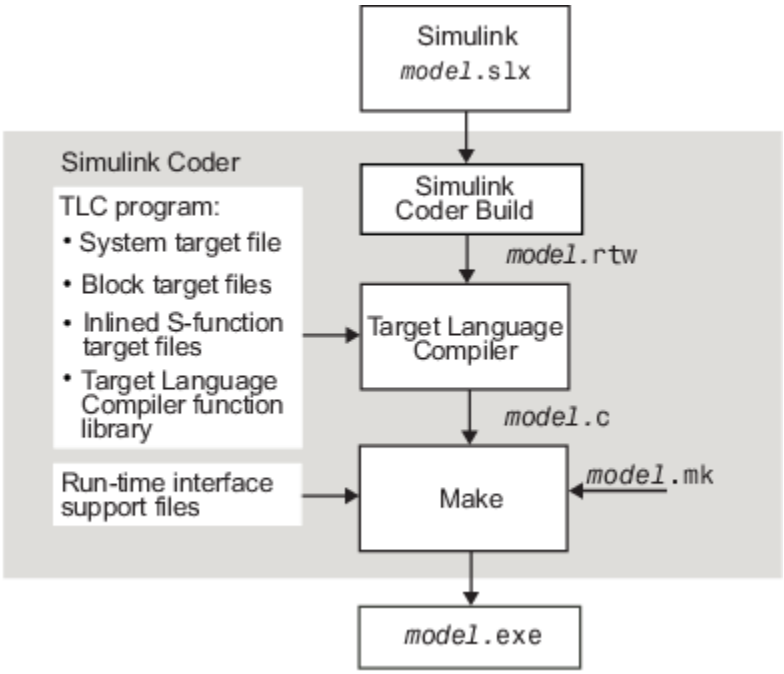
The Target Language Compiler provides a complete set of ready-to-use TLC files for generating ANSI C or C++ code. You can view the TLC files and make minor or extensive changes to them. This open environment provides tremendous flexibility for customizing the generated code.

For more information, see “Implement C/C++ S-Functions”, which describes how to write wrapped and fully inlined S-functions, with emphasis on the `mdlRTW()` function.

Note Do not customize TLC files in the folder *`matlabroot`*`/rtw/c/tlc`, even though the capability exists. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Overview of the TLC Process

This top-level diagram shows how the Target Language Compiler fits in with the code generation process.



The Target Language Compiler (TLC) is designed to convert the model description file *model*.rtw (or similar files) into target-specific code or text.

The Target Language Compiler transforms a representation of a Simulink block diagram, called *model*.rtw, into C or C++ code. The *model*.rtw file contains a partial representation of the model. The representation describes the execution semantics of the block diagram in a high-level language. For more information, see “model.rtw File and Scopes” on page 18-2.

After reading the *model*.rtw file, the Target Language Compiler generates its code based on target files, which specify particular code for each block, and model-wide files, which specify the overall code style. The TLC uses the target files and the *model*.rtw file to generate ANSI C or C++ code.

To create a target-specific application, the code generator requires a template makefile that specifies a C or C++ compiler and compiler options for the build process. The code generator transforms the template makefile into a target makefile (*model*.mk) by performing token expansion specific to a given model. The target makefile is a modified version of the generic rt_main file (or grt_main). You must modify grt_main to conform to the target’s specific requirements, such as interrupt service routines. See “Template Makefiles and Make Options” and “Customize Template Makefiles”.

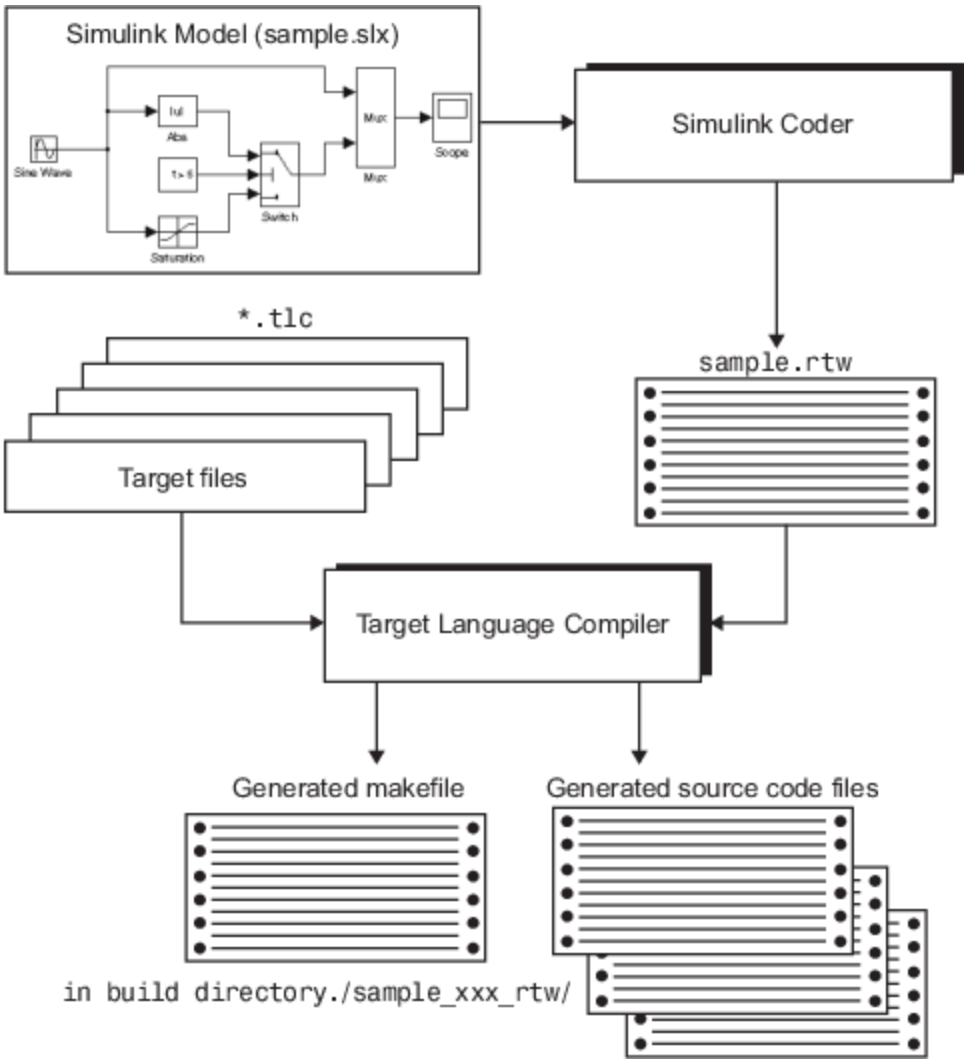
The Target Language Compiler has similarities with HTML, Perl, and MATLAB. It has markup syntax similar to HTML, the power and flexibility of Perl and other scripting languages, and the data handling power of MATLAB (TLC can invoke MATLAB functions). The code that TLC generated is highly optimized and fully commented. With TLC, you can generate code from linear, nonlinear, continuous, discrete, or hybrid Simulink models. The models can include Simulink blocks that are automatically converted to code. Exceptions are MATLAB function blocks and S-function blocks that invoke MATLAB files. The Target Language Compiler uses block target files to transform each block in the *model*.rtw file and a model-wide target file for global customization of the code.

You can incorporate C MEX S-functions, with the generated code into the program executable. You can write a target file for your C MEX S-function to inline the S-function (see “Inline C MEX S-Functions” on page 21-8), to improve performance by eliminating function calls to the S-function

itself and the memory overhead of the`SimStruct` of the S-function. Inlining an S-function incorporates the S-function block code into the generated code for the model. When a TLC target file is not present for the S-function, its C or C++ code file is invoked through a function call. See “Inline S-Functions” on page 21-2. You can also write target files for MATLAB language files or Fortran S-functions.

Overview of the Code Generation Process

The Target Language Compiler works with its target files and the code generator output to produce code.



When generating code from a Simulink model, the first step in the automated process is to generate a *`model`*`.rtw` file. The *`model`*`.rtw` file includes the model-specific information required for generating code from the Simulink model. *`model`*`.rtw` is passed to the Target Language Compiler, which uses it in combination with a set of included system target files and block target files to generate the code.

Only the final executable file is written directly to the current folder. For other files created during code generation, including the *`model`*`.rtw` file, a build folder is used. This folder is created in the

current folder and is named *model*_*target*_rtw. *target* is the abbreviation for the target environment grt that is a generic real-time target.

Files placed in the build folder include:

- The body for the generated C or C++ source code (*model*.c or *model*.cpp)
- Header files (*model*.h)
- Header file *model*_private.h defining parameters and data structures private to the generated code
- A makefile, *model*.mk, for building the application
- Additional files, described in “Manage Build Process Files”

See Also

Related Examples

- “Why Use the Target Language Compiler?” on page 14-6
- “Advice About TLC Tutorials” on page 16-2
- “TLC Files” on page 17-11
- “Inline S-Functions with TLC” on page 16-17

Why Use the Target Language Compiler?

If you simply need to produce ANSI C or C++ code from Simulink models, you do not need to know how to prepare files for the Target Language Compiler. If you need to customize the output, you must run the Target Language Compiler. With the Target Language Compiler, you can:

- Customize the set of options specified by your system target file.
- Inline the code for S-Function blocks.
- Generate additional or different types of files.

The MATLAB Function block and the Embedded Coder product facilitate code customization in a variety of ways. You might be able to accomplish what you need with them, without the need to write TLC files. However, you do need to prepare TLC files if you intend to inline S-functions.

See the following sections.

Customizing Output

To produce customized output using the Target Language Compiler, it helps if you understand how blocks perform their functions, what data types are being manipulated, the structure of the `model.rtw` file, and how to modify target files to produce the desired output. Directives and Built-In Functions topics on “Target Language Compiler”, describe the target language directives and their associated constructs. You will use the Target Language Compiler directives and constructs to modify existing target files or create new ones, depending on your needs. See “TLC Files” on page 17-11 for more information about target files.

Note You should not customize TLC files in the folder `matlabroot/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Inlining S-Functions

The Target Language Compiler provides a great deal of freedom for altering, optimizing, and enhancing the generated code. One of the most important TLC features is that it lets you inline S-functions that you write to add your own algorithms, device drivers, and custom blocks to a Simulink model.

To create an S-function, you write code following a well-defined application program interface (API). By default, the Target Language Compiler will generate noninlined code for S-functions that invokes them using this same API. This generalized interface incurs a fair amount of overhead due to the presence of a large data structure called the `SimStruct` for each instance of each S-Function block in your model. In addition, extra run-time overhead is involved whenever methods (functions) within your S-function are called. You can eliminate this overhead by using the Target Language Compiler to inline the S-function, by creating a TLC file named `sfunction_name.tlc` that generates source code for the S-function as if it were a built-in block. Inlining an S-function improves the efficiency of the generated code and reduces memory usage.

Defining Advanced Storage Classes

Certain data layouts, such as nested structures, cannot be generated using the standard `Unstructured` and `FlatStructure` storage class types. You can define an *advanced storage class* if

you want to generate other types of data. Creating advanced storage classe requires understanding TLC programming and using a special advanced mode of the Custom Storage Class Designer. For more information, see “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23. Note that this support requires an Embedded Coder license.

See Also

Related Examples

- “Target Language Compiler Process” on page 15-6
- “Code Architecture” on page 15-2

The Advantages of Inlining S-Functions

The goals of generated code usually include compactness and speed. On the other hand, S-functions are run-time-loadable extension modules for adding block-level functionality to Simulink. As such, the S-function interface is optimized for flexibility in configuring and using blocks in a simulation environment with capability to allow run-time changes to a block's operation via parameters. These changes typically take the form of algorithm selection and numerical constants for the block algorithms.

While switching algorithms is a desirable feature in the design phase of a system, when the time comes to generate code, this type of flexibility is often dropped in favor of optimal calculation speed and code size. The Target Language Compiler is designed to allow the generation of code that is compact and fast by selectively generating only the code you need for one instance of a block's parameter set.

When To Avoid Inlining

You might decide not to inline C MEX S-functions that have:

- Few or no numerical parameters.
- One algorithm that is already fixed in capability. For example, it has no optional modes or alternate algorithms.
- Support for only one data type.
- A significant or large code size in the `mdlOutputs()` function.
- Multiple instances of S-Function block in your models.

Whenever you encounter this situation, the effort of inlining the block might not improve execution speed and could actually increase the size of the generated code. The tradeoff is in the size of the block's body code generated for each instance versus the size of the child SimStruct created for each instance of a noninlined S-function in the generated code.

Alternatively, you can use a hybrid inlining method known as a C MEX wrapped S-function, where the block target file simply generates a call to a custom code function that the S-function itself also calls. This approach might be the optimal solution for code generation in the case of a large piece of existing code. See "Write Wrapper S-Function and TLC Files" for the procedure and an example of a wrapped S-function.

Inlining Process

The strategy for improving code from blocks centers on determining what part of a block's operations are active and used in the generated code and what parts can be predetermined or left out.

In practice, this means the TLC code in the block target file will select an algorithm that is a subset of the algorithms contained in the S-function itself and then selectively hard-code numerical parameters that are not to be changed at run time. This reduces code memory size and results in code that is often much faster than its S-function counterpart when mode selection is a significant part of S-function processing. Additionally, function-call overhead is eliminated for inlined S-functions, as the code is generated directly in the body of the code unless there is an explicit call to a library function in the generated code.

The algorithm selections and parameters for each block are output in the initial phase of the code generation process from the registered S-function parameter set or the `mdlRTW()` function (if

present), which results in entries in the model's `.rtw` file for that block at code generation time. A file written in the target language for the block is then called to read the entries in the *`model`*`.rtw` file and compute the generated code for this instance of the block. This TLC code is contained in the block target file.

One special case for inlined S-functions is for the case of I/O blocks and drivers such as A/D converters or communications ports. For simulation, the I/O driver is typically coded in the S-function as a pure source, a pass-through, or a pure sink. In the generated code, however, an actual interface to the I/O device must be made, typically through direct coding with the common `_in()`, `_out()` functions, inlined assembly code, or a specific set of I/O library calls unique to the device and target environment.

Search Algorithm for Locating TLC Files

The Target Language Compiler uses the following search order to locate TLC files:

1. Current folder.
2. Locations specified by `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.
3. Locations specified by `-I` options. The compiler evaluates multiple `-I` options from *right to left*.

For inlined S-function TLC files, the build process supports the following locations:

- The folder where the S-function executable (MEX or MATLAB) file is located.
- S-function folder subfolder `./tlc_c` (for C or C++ language targets).
- The current folder when the build process is initiated.

> **Note** Note: Placing the inlined S-function TLC file elsewhere is not supported, even if the location is in the TLC include path.

The first target file encountered with the required name that implements the specified language is used in processing the S-function *`model`*`.rtw` file entry.

> **Note** The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

Availability for Inlining and Noninlining

S-functions can be written in MATLAB language, Fortran, C, and C++. TLC inlining of S-functions is available as indicated in this table.

Inline TLC Support by S-Function Type

S-Function Type	Noninlining Supported	Inlining Supported
MATLAB language	No	Yes
Fortran MEX	No	Yes
C	Yes	Yes
C++	Yes	Yes

See Also

Related Examples

- “Inlining S-Functions” on page 15-8

Getting Started

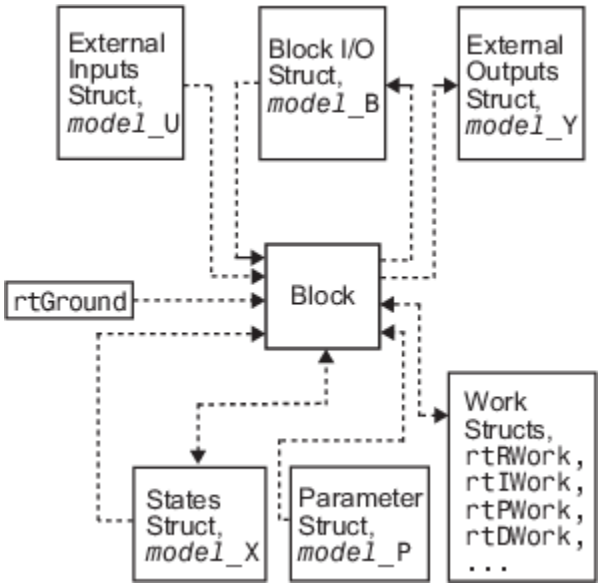
- “Code Architecture” on page 15-2
- “Target Language Compiler Process” on page 15-6
- “Inlining S-Functions” on page 15-8

Code Architecture

Before investigating the specific code generation pieces of the Target Language Compiler (TLC), consider how Target Language Compiler generates code for a simple model. From the next figure, you see that blocks place code into `Mdl` routines. This shows `MdlOutputs`.

```
static void simple_output(int_T tid)
{

  /* Sin Block: '<Root>/Sine Wave' */

  simple_B.SineWave_d = simple_P.SineWave_Amp *
    sin(simple_P.SineWave_Freq * simple_M->Timing.t[0] +
    simple_P.SineWave_Phase) + simple_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  simple_B.Gain_d = simple_B.SineWave_d * simple_P.Gain_Gain;

  /* Outport: '<Root>/Out1' */
  simple_Y.Out1 = simple_B.Gain_d;
}
```

Blocks have inputs, outputs, parameters, states, plus other general properties. For example, block inputs and outputs are generally written to a block I/O structure (generated with identifiers of the type *`model`*`_B`, where *`model`* is the model name). Block inputs can also come from the external input structure (*`model`*`_U`) or the state structure when connected to a state port of an integrator (*`model`*`_X`), or ground (`rtGround`) if unconnected or grounded. Block outputs can also go to the external output structure, (*`model`*`_Y`). The following diagram shows the general block data mappings.



This discussion should give you a general sense of what the block object looks like. Now, you can look at specific pieces of the code generation process that are specific to the Target Language Compiler.

Code Generation Process

The code generator invokes the Target Language Compiler after a model is compiled into a partial representation of the model (*model*`.rtw`) suitable for code generation. To generate code, the Target Language Compiler uses its library of functions to transform two classes of target files:

- System target files
- Block target files

System target files are used to specify the overall structure of the generated code, tailoring for specific target environments. Block target files are used to implement the functionality of Simulink blocks, including user-defined S-function blocks.

You can create block target files for C MEX, Fortran, and MATLAB language S-functions to fully inline block functionality into the body of the generated code. C MEX S-functions can be noninlined, wrapper-inlined, or fully inlined. Fortran S-functions must be wrapper-inlined or fully inlined.

How TLC Determines S-Function Inlining Status

Whenever the Target Language Compiler encounters an entry for an S-function block in the *model*`.rtw` file, it must decide whether to generate a call to the S-function or to inline it.

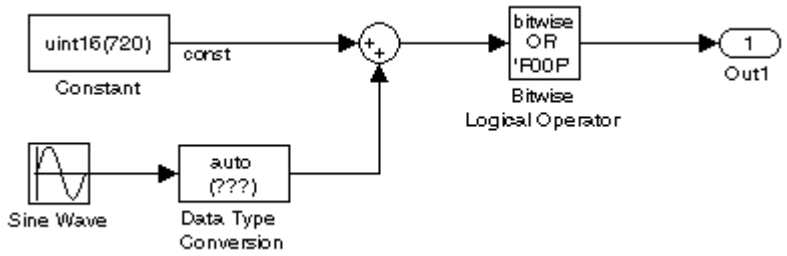
Because they cannot use `SimStructs`, Fortran and MATLAB language S-functions must be inlined. This inlining can either be in the form of a full block target file or a one-line block target file that refers to a substitute C MEX S-function source file.

The Target Language Compiler selects a C MEX S-function for inlining if an explicit `mdlRTW()` function exists in the S-function code or if a target file for the current target language for the current block is in the TLC file search path. If a C MEX S-function has an explicit `mdlRTW()` function, there must be a corresponding target file or an error condition results.

The target file for an S-function must have the same root name as the S-function and must have the extension `.tlc`. For example, the target file for a C MEX S-function named `sfix_bitop` has the filename `sfix_bitop.tlc`.

Inlined and Noninlined S-Function Code

This example focuses on a C MEX S-function named `sfix_bitop`. The code generation options are set to allow reuse of signal memory for signal lines that were not set as tunable signals.



The code generated for the bit-wise operator block reuses a temporary variable that is set up for the output of the sum block to save memory. This results in one very efficient line of code, as seen here.

```
/* Bitwise Logic Block: <Root>/Bitwise Logical Operator */
/* [input] OR 'F00F' */
rtb_temp2 |= 0xF00F;
```

Initialization or setup code is not required for this inlined block.

If this block were not inlined, the source code for the S-function itself with its various options would be added to the generated code base, memory would be allocated in the generated code for the block's `SimStruct` data, and calls to the S-function methods would be generated to initialize, run, and terminate the S-function code. To execute the `mdlOutputs` function of the S-function, code would be generated like this:

```
/* Level2 S-Function Block: <Root>/Bitwise Logical Operator (sfix_bitop) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
```

The entire `mdlOutputs` function is called and runs just as it does during simulation. That's not everything, though. There is also registration, initialization, and termination code for the noninlined S-function. The initialization and termination calls are similar to the fragment above. Then, the registration code for an S-function with just one inport and one outport is 72 lines of C code generated as part of file *`model`*`_reg.h`.

```
/*Level2 S-Function Block: <Root>/Bitwise Logical Operator (sfix_bitop) */
    {
      extern void untitled_sf(SimStruct *rts);
      SimStruct *rts = ssGetSFunction(rtS, 0);

      /* timing info */
      static time_T sfcnPeriod[1];
      static time_T sfcnOffset[1];
      static int_T sfcnTsMap[1];

      {
        int_T i;

        for(i = 0; i < 1; i++) {
          sfcnPeriod[i] = sfcnOffset[i] = 0.0;
        }
      }
      ssSetSampleTimePtr(rts, &sfcnPeriod[0]);
      ssSetOffsetTimePtr(rts, &sfcnOffset[0]);
      ssSetSampleTimeTaskIDPtr(rts, sfcnTsMap);
      ssSetMdlInfoPtr(rts, ssGetMdlInfoPtr(rtS));

      /* inputs */
      {
        static struct _ssPortInputs inputPortInfo[1];

        _ssSetNumInputPorts(rts, 1);
        ssSetPortInfoForInputs(rts, &inputPortInfo[0]);

        /* port 0 */
        {

          static real_T const *sfcnUPtrs[1];
          sfcnUPtrs[0] = &rtU.In1;
          ssSetInputPortSignalPtrs(rts, 0, (InputPtrsType)&sfcnUPtrs[0]);
          _ssSetInputPortNumDimensions(rts, 0, 1);
          ssSetInputPortWidth(rts, 0, 1);
        }
      }
.
.
.
```

This continues until S-function sizes and methods are declared, allocated, and initialized. The amount of registration code generated is essentially proportional to the number and size of the input ports and output ports.

A noninlined S-function will typically have a significant impact on the size of the generated code, whereas an inlined S-function can be close to the handwritten size and performance of the generated code.

See Also

Related Examples

- “Write Noninlined S-Function”
- “Write Fully Inlined S-Functions”

Target Language Compiler Process

To write TLC code for your S-function, you need to understand the Target Language Compiler process for code generation. As previously described, the Simulink software generates a *model*.rtw file that contains a partial representation of the execution semantics of the block diagram. The *model*.rtw file is an ASCII file that contains a data structure in the form of a nested set of TLC records. The records comprise property name/property value pairs. The Target Language Compiler reads the *model*.rtw file and converts it into an internal representation.

Next, the Target Language Compiler runs (interprets) the TLC files, starting first with the system target file, for example, grt.tlc. This is the entry point to the system TLC and block files, that is, other TLC files included in or generated from the TLC file passed to Target Language Compiler on its command line (grt.tlc). As the TLC code in the system and block target files is run, it uses, appends to, and modifies the existing property name/property value pairs and records initially loaded from the *model*.rtw file.

model.rtw Structure

The structure of the *model*.rtw file mirrors the block diagram's structure:

- For each nonvirtual system in the model, there is a corresponding system record in the *model*.rtw file.
- For each nonvirtual block within a nonvirtual system, there is a block record in the *model*.rtw file in the corresponding system.

The basic structure of *model*.rtw is

```
CompiledModel {
  System {
    Block {
      DataInputPort {
        ...
      }
      DataOutputPort{
        ...
      }
      ParamSettings {
        ...
      }
      Parameter {
        ...
      }
    }
  }
}
```

Operating Sequence

For each occurrence of a given block in the model, a corresponding block record exists in the *model*.rtw file. The system target file TLC code loops through block records and calls the functions in the corresponding block target file for that block type. For inlined S-functions, it calls the inlining TLC file.

There is a method for getting block-specific information (internal block information, as opposed to inputs, outputs, parameters, etc.) into the block record in the *model*`.rtw` file for a block by using the `mdlRTW` function in the C MEX function of the block.

Among other things, the `mdlRTW` function allows you to write out parameter settings (`ParamSettings`), that is, unique information pertaining to this block. For parameter settings in the block TLC file, direct accesses to these fields are made from the block TLC code and can be used to alter the generated code as desired.

See Also

Related Examples

- “S-Functions and Code Generation”
- “Write Fully Inlined S-Functions with mdlRTW Routine”

Inlining S-Functions

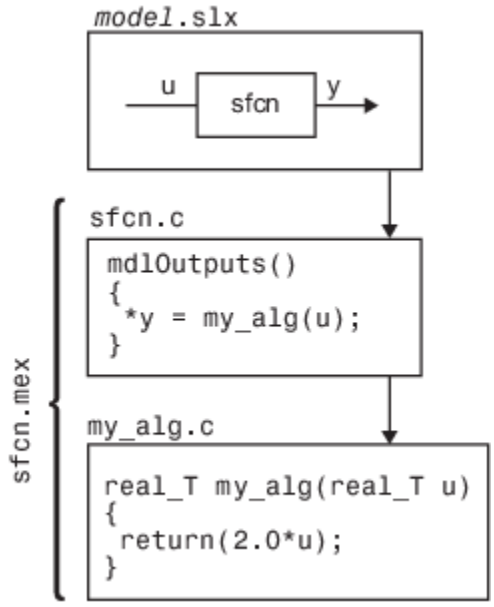
Inlining an S-function

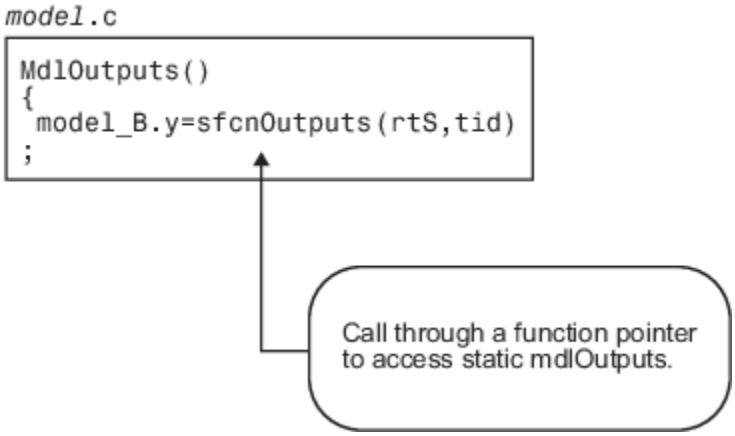
To inline an S-function means to provide a TLC file for an S-Function block that will replace the C, C++, Fortran, or MATLAB language version of the block that was used during simulation.

Noninlined S-Function

If an inlining TLC file is not provided, most targets support the block by recompiling the C MEX S-function for the block. As discussed earlier, there is overhead in memory usage and speed when using a C/C++ coded S-function and a limited subset of `mx*` API calls supported within the code generator context. If you want the most efficient generated code, you must inline S-functions by writing a TLC file for them.

When the simulation needs to execute one of the functions for an S-function block, it calls the MEX-file for that function. When the code generator executes a noninlined S-function, it does so in a similar manner, as this diagram illustrates.





Types of Inlining

It is helpful to define two categories of inlining:

- Fully inlined S-functions
- Wrapper inlined S-functions

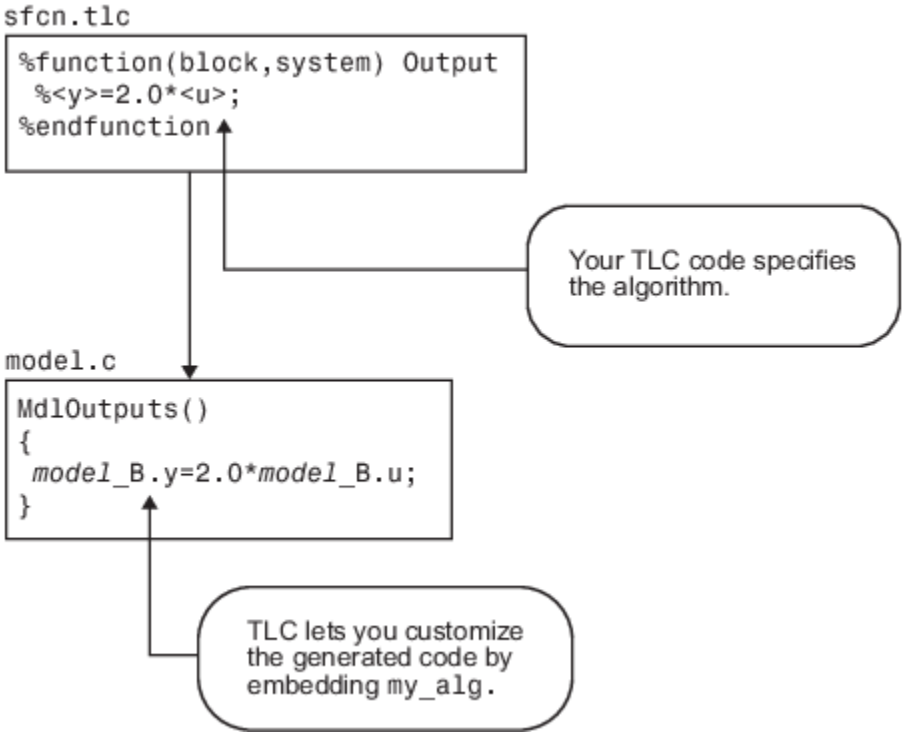
While both inline the S-function and remove the overhead of a noninlined S-function, the two approaches are different. The first example below, using `timestwo.tlc`, is considered a fully inlined TLC file, where the full implementation of the block is contained in the TLC file for the block.

The second example uses a wrapper TLC file. Instead of generating the algorithmic code in place, this example calls a C function that contains the body of code. There are several potential benefits for using the wrapper TLC file:

- It provides a way for the C MEX S-function and the generated code to share the C code. You do not need to write the code twice.
- The called C function is an optimized routine.
- Several of the blocks might exist in the model, and it is more efficient in terms of code size to have them call a function, as opposed to each creating identical algorithmic code.
- It provides a way to incorporate legacy C code seamlessly into generated code.

Fully Inlined S-Function Example

Inlining an S-function provides a mechanism to directly embed code for an S-function block into the generated code for a model. Instead of calling into a separate source file via function pointers and maintaining a separate data structure (`SimStruct`) for it, the code appears “inlined” as the next figure shows.



The S-function `timestwo.c` provides a simple example of a fully inlined S-function. This block multiplies its input by 2 and outputs it. The C MEX version of the block is in the file *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/timestwo.c`, and the inlining TLC file for the block is *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/timestwo.tlc`.

timestwo.tlc

```
%implements "timestwo" "C"

%% Function: Outputs ========================================
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
  %endroll
%endfunction
```

TLC Block Analysis

The `%implements` directive is required by TLC block files and is used by the Target Language Compiler to verify the block type and language supported by the block. The `%function` directive starts a function declaration and shows the name of the function, `Outputs`, and the arguments passed to it, `block` and `system`. These are the relevant records from the *`model`*`.rtw` file for this instance of the block.

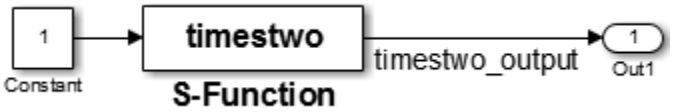
The last piece of the prototype is `Output`. This means that any line that is not a TLC directive is output by the function to the current file that is selected in TLC. So, nondirective lines in the `Outputs` function become generated code for the block.

The most complicated piece of this TLC block example is the `%roll` directive. TLC uses this directive to provide automatic generation of `for` loops, depending on input/output widths and whether the inputs are contiguous in memory. This example uses the typical form of accessing outputs and inputs from within the body of the roll, using `LibBlockOutputSignal` and `LibBlockInputSignal` to access the outputs and inputs and perform the multiplication and assignment. Note that this TLC file supports any valid signal width.

The only function used to implement this block is `Outputs`. For more complicated blocks, other functions are declared as well. You can find examples of more complicated inlining TLC files in the folders *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c` (open) and *`matlabroot`*`/toolbox/simulink/blocks/tlc_c` (open), and by looking at the code for built-in blocks in the folder *`matlabroot/rtw/c/tlc/blocks`* (open).

The timestwo Model

This simple model uses the `timestwo` S-function and shows the `MdlOutputs` function from the generated *`model`*`.c` file, which contains the inlined S-function code.

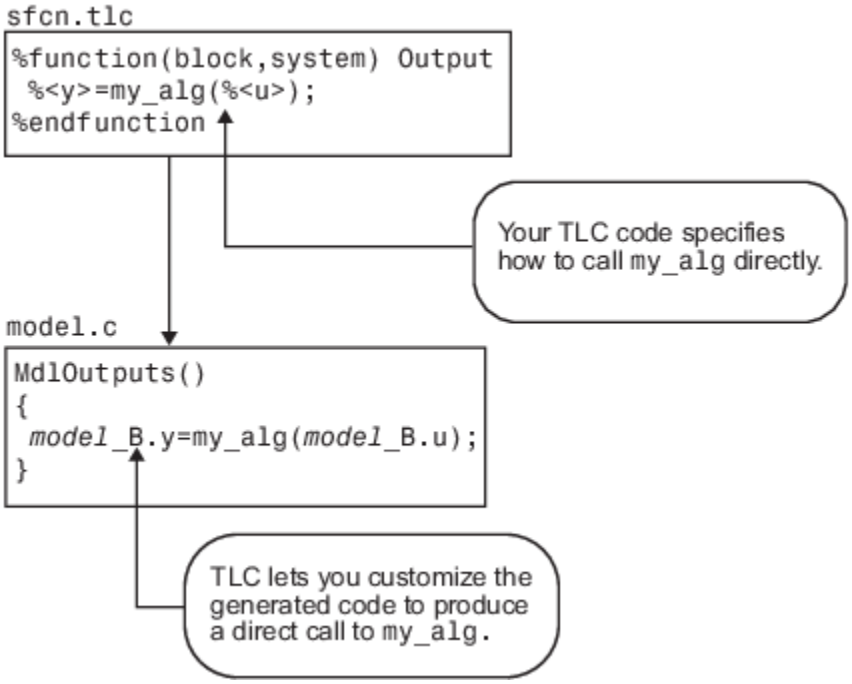


Model Outputs Code

```
/* Model output function */
static void timestwo_ex_output(int_T tid)
{

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by two */
  timestwo_ex_B.timestwo_output = timestwo_ex_P.Constant_Value
     * 2.0;

  /* Outport: '<Root>/Out1' */
  timestwo_ex_Y.Out1 = timestwo_ex_B.timestwo_output;
}
```

Wrapper Inlined S-Function Example

The following diagram illustrates inlining an S-function as a wrapper. The algorithm is directly called from the generated model code, removing the S-function overhead but maintaining the user function.



This is the inlining TLC file for a wrapper version of the `timestwo` block.

```
%implements "timestwo" "C"

%% Function: BlockTypeSetup ==================================
%%
%function BlockTypeSetup(block, system) void
  %% Add function prototype to model's header file
  %<LibCacheFunctionPrototype...
   ("extern void mytimestwo(real_T* in,real_T* out,int_T els);")>
  %% Add file that contains "myfile" to list of files to be compiled
  %<LibAddToModelSources("myfile")>
%endfunction

%% Function: Outputs ==========================================
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign outPtr = LibBlockOutputSignalAddr(0, "", "", 0)
  %assign inPtr = LibBlockInputSignalAddr(0, "", "",0)
  %assign numEls = LibBlockOutputSignalWidth(0)
  /* Multiply input by two */
  mytimestwo(%<inPtr>,%<outPtr>,%<numEls>);

%endfunction
```

Analysis

The function `BlockTypeSetup` is called once for each type of block in a model; it doesn't produce output directly like the `Outputs` function. Use `BlockTypeSetup` to include a function prototype in the *`model`*`.h` file and to tell the build process to compile an additional file, `myfile.c`.

Instead of performing the multiplication directly, the `Outputs` function now calls the function `mytimestwo`. All instances of this block in the model call the same function to perform the multiplication. The resulting model function, `MdlOutputs`, then becomes

```
static void timestwo_ex_output(int_T tid)
{
```

```
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by two */
  mytimestwo(&model_B.Constant_Value,&model_B.S_Function,1);

  /* Outport Block: <Root>/Out1 */
model_Y.Out1 = model_B.S_Function;
}
```

See Also

Related Examples

- “Write Noninlined S-Function”
- “Write Wrapper S-Function and TLC Files”
- “Write Fully Inlined S-Functions”
- “Write Fully Inlined S-Functions with mdlRTW Routine”

Target Language Compiler Tutorials

- “Advice About TLC Tutorials” on page 16-2
- “Read Record Files with TLC” on page 16-4
- “Inline S-Functions with TLC” on page 16-17
- “Explore Variable Names and Loop Rolling” on page 16-21
- “Debug Your TLC Code” on page 16-26
- “TLC Code Coverage to Aid Debugging” on page 16-32
- “Wrap User Code with TLC” on page 16-34

Advice About TLC Tutorials

The fastest and easiest way to understand the Target Language Compiler (TLC) is to run it, paying attention to how TLC scripts transform compiled Simulink models (*model*`.rtw` files) into source code. The tutorials highlight the principal reasons for and techniques of using TLC. The tutorials provide a number of TLC exercises, each one organized as a major section.

The example models, S-functions, and TLC files for the exercises are located in the folder *matlabroot*`/toolbox/rtw/rtwdemos/tlctutorial` (open). In this chapter, this folder is referred to as `tlctutorial`. Each example is located in a separate subfolder within `tlctutorial`. Within that subfolder, you can find solutions to the problem in a `solutions` subfolder.

Note Before you begin the tutorial, copy the entire `tlctutorial` folder to a local working folder. The files are together, and if you make mistakes or want fresh examples to try again, you can recopy files from the original `tlctutorial` folder.

Each tutorial exercise is limited in scope, requiring just a small amount of experimentation. The tutorial explains details about TLC that will help customize and optimize code for code generation projects.

Note You should not customize TLC files in the folder *matlabroot*`/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

The tutorials progress in difficulty from basic to more advanced. To get the most out of them, you should be familiar with

- Working in the MATLAB environment
- Building Simulink models
- Using the code generator to produce code for target systems
- High-level language concepts (for example, C or Fortran programming)

If you encounter terms in the tutorials that you do not understand, it may be helpful to read “Code Generation Concepts” on page 17-7 to acquaint yourself with the basic goals and methods of TLC programming. Similarly, if you see TLC keywords, built-in functions, or directives that you would like to know more about, see the corresponding topics on “Target Language Compiler”.

The examples used in the tutorial are:

Example	Description
guide	Illustrative record file
timesN	An example C file S-function for multiplying an input by N
tlcdebug	An example using TLC Debugger
wrapper	Example TLC file for S-function `wrapsfcn.c`

See Also

Related Examples

- “Read Record Files with TLC” on page 16-4

Read Record Files with TLC

Tutorial Overview

Objective: Understand the structure of record files and learn how to parse them with TLC directives.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/guide (open)

In this tutorial you interpret a simple file of structured records with a series of TLC scripts. You will learn how records are structured, and how TLC `%assign` and `%<>` token expansion directives are used to process them. In addition, the tutorial illustrates loops using `%foreach`, and scoping using `%with`.

The tutorial includes these steps, which you should follow sequentially:

1. **Structure of Record Files** — Some background and a simple example
2. **Interpret Records** — Presenting contents of the record file
3. **Anatomy of a TLC Script** — Deconstructing the presentation
4. **Modify `read-guide.tlc`** — Experiment with TLC
5. **Pass and Use a Parameter**— Pass parameters from the command line to TLC files
6. **Review**

Structure of Record Files

The code generator compiles models into a structured form called a record file, referred to as *model*`.rtw`. Such compiled model files are similar in syntax and organization to source model files, in that they contain a series of hierarchically nested records of the form

```
recordName {itemName itemValue}
```

Item names are alphabetic. Item values can be strings or numbers. Numeric values can be scalars, vectors, or matrices. Curly braces set off the contents of each record, which may contain one or more items, delimited by space, tab, or return characters.

In a *model*`.rtw` file, the top-level (first) record's name is `CompiledModel`. Each block is represented by a subrecord within it, identified by the block's name. TLC can parse well-formed record files, as this exercise illustrates.

The following listing is a valid record file that TLC can parse, although not one for which it can generate code. Comments are indicated by a pound sign (#):

```
#
# File: guide.rtw    Illustrative record file, which can't be used by Simulink
#                                   Note: string values MUST be in quotes
Top {                             # Outermost Record, called Top
  Date         "21-Aug-2008"      # Name/Value pair named Top.Date
  Employee {                      # Nested record within the Top record
    FirstName  "Arthur"           #   Alpha field Top.Employee.FirstName
    LastName   "Dent"             #   Alpha field Top.Employee.LastName
    Overhead    1.78              #   Numeric field Top.Employee.Overhead
    PayRate     11.50             #   Numeric field Top.Employee.PayRate
    GrossRate   0.0               #   Numeric Field Top.Employee.GrossRate
  }                               # End of Employee record
  NumProject   3                  # Indicates length of following list
  Project {                       # First list item, called Top.Project[0]
```

```
    Name       "Tea"                #   Alpha field Name, Top.Project[0].Name
    Difficulty 3                    #   Numeric field Top.Project[0].Difficulty
  }                                 # End of first list item
  Project {                         # Second list item, called Top.Project[1]
    Name       "Gillian"            #   Alpha field Name, Top.Project[1].Name
    Difficulty 8                    #   Numeric field Top.Project[1].Difficulty
  }                                 # End of second list item
  Project {                         # Third list item, called Top.Project[2]
    Name       "Zaphod"             #   Alpha field Name, Top.Project[2].Name
    Difficulty 10                   #   Numeric field Top.Project[2].Difficulty
  }                                 # End of third list item
}                                   # End of Top record and of file
```

As long as programmers know the names of records and fields, and their expected contents, they can compose TLC statements to read, parse, and manipulate record file data.

Interpret Records

Here is the output from a TLC program script that reads `guide.rtw`, interprets its records, manipulates field data, and formats descriptions, which are directed to the MATLAB Command Window:

```
Using TLC you can:
* Directly access a field's value, e.g.
  %<Top.Date> -- evaluates to:
  "21-Aug-2008"

* Assign contents of a field to a variable, e.g.
  "%assign worker = Top.Employee.FirstName"
  worker expands to Top.Employee.FirstName = "Arthur"

* Concatenate string values, e.g.
  "%assign worker = worker + " " + Top.Employee.LastName"
  worker expands to worker + " " + Top.Employee.LastName = "Arthur Dent"

* Perform arithmetic operations, e.g.
  "%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead"
  wageCost expands to Top.Employee.PayRate * Top.Employee.Overhead <- 11.5 * 1.78 = 20.47

* Put variables into a field, e.g.
  Top.Employee.GrossRate starts at 0.0
  "%assign Top.Employee.GrossRate = wageCost"
  Top.Employee.GrossRate expands to wageCost = 20.47

* Index lists of values, e.g.
  "%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name..."
  "+ ", " + Top.Project[2].Name"
  projects expands to Top.Project[0].Name + ", " + Top.Project[1].Name
  + ", " + Top.Project[2].Name = Tea, Gillian, Zaphod

* Traverse and manipulate list data via loops, e.g.
  - At top of Loop, Project = Tea; Difficulty = 3
  - Bottom of Loop, i = 0; diffSum = 3.0
  - At top of Loop, Project = Gillian; Difficulty = 8
  - Bottom of Loop, i = 1; diffSum = 11.0
  - At top of Loop, Project = Zaphod; Difficulty = 10
  - Bottom of Loop, i = 2; diffSum = 21.0
  Average Project Difficulty expands to diffSum / Top.NumProject = 21.0 / 3 = 7.0
```

This output from `guide.rtw` was produced by invoking TLC from the MATLAB Command Window, executing a script called `read-guide.tlc`. Do this yourself now, by following these steps:

1. In MATLAB, change folder (`cd`) to your copy of `tlctutorial/guide` within your working folder.
2. To produce the output just listed, process `guide.rtw` with the TLC script `read-guide.tlc` by typing the following command:

   ```
   tlc -v -r guide.rtw read-guide.tlc
   ```

Note command usage:

- The `-r` switch (for read) identifies the input data file, in this case `guide.rtw`.
- The TLC script handling the data file is specified by the last token typed.
- The `-v` switch (for verbose) directs output to the command window, unless the TLC file handles this itself.

Anatomy of a TLC Script

You now dissect the script you just ran. Each "paragraph" of output from `guide.tlc` is discussed in sequence in the following brief sections:

- "Coding Conventions" on page 16-6 — Before you begin
- "File Header" on page 16-6 — Header info and a formatting directive
- "Token Expansion" on page 16-7— Evaluating field and variable identifiers
- "General Assignment" on page 16-7 — Using the `%assign` directive
- "String Processing Plus" on page 16-8 — Methods of assembling strings
- "Arithmetic Operations" on page 16-9 — Computations on fields and variables
- "Modify Records" on page 16-9 — Changing, copying, appending to records
- "Index Lists" on page 16-10 — Referencing list elements with subscripts
- "Loop Over Lists" on page 16-10 — Details on loop construction and behavior

Coding Conventions

These are some basic TLC syntax and coding conventions:

`%% Comment`	TLC comment, which is not output
`/* comment */`	Comment, to be output
`%keyword`	TLC directive (keyword), start with "%"
`%<expr>`	TLC token operator
`.` (period)	Scoping operator, for example, `Top.Lev2.Lev3`
`...` (at end-of-line)	Statement continuation (line break is not output)
`\` (at end-of-line)	Statement continuation (line break is output)
`localvarIdentifier`	Local variables start in lowercase
`GlobalvarIdentifier`	Global variables start in uppercase
`RecordIdentifier`	Record identifiers start in uppercase
`EXISTS()`	TLC built-in functions are named in uppercase **Note:** TLC identifiers are case-sensitive.

For further information, see "TLC Coding Conventions" on page 21-17.

File Header

The file `read-guide.tlc` begins with:

```
%% File: read-guide.tlc   (This line is a TLC Comment, and will not print)
%%
%% To execute this file, type: tlc -v -r guide.rtw read-guide.tlc
%% Set format for displaying real values (default is "EXPONENTIAL")
%realformat "CONCISE"
```

- Lines 1 through 4 — Text on a line following the characters `%%` is treated as a comment (ignored, not interpreted or output).
- Line 5 — As explained in the text of the fourth line, is the TLC directive (keyword) `%realformat`, which controls how subsequent floating-point numbers are formatted when displayed in output. Here we want to minimize the digits displayed.

Token Expansion

The first section of output is produced by the script lines:

```
Using TLC you can:
* Directly access a field's value, e.g.
%assign td = "%" + "<Top.Date>"
  %<td> -- evaluates to:
  "%<Top.Date>"
```

- Lines 1 and 2 — (and lines that contain no TLC directives or tokens) are simply echoed to the output stream, including leading and trailing spaces.
- Line 3 — Creates a variable named `td` and assigns the string value `%<Top.Date>` to it. The `%assign` directive creates new and modifies existing variables. Its general syntax is:

  ```
  %assign ::variable = expression
  ```

  The optional double colon prefix specifies that the variable being assigned to is a global variable. In its absence, TLC creates or modifies a local variable in the current scope.
- Line 4 — Displays

  ```
  %<Top.Date> -- evaluates to:
  ```

  The preceding line enables TLC to print `%<Top.Date>` without expanding it. It constructs the string by pasting together two literals.

  ```
  %assign td = "%" + "<Top.Date>"
  ```

  As discussed in “String Processing Plus” on page 16-8, the plus operator concatenates strings as and adds numbers, vectors, matrices, and records.
- Line 5 — Evaluates (expands) the record `Top.Date`. More precisely, it evaluates the field `Date` which exists in scope `Top`. The syntax `%<expr>` causes expression `expr` (which can be a record, a variable, or a function) to be evaluated. This operation is sometimes referred to as an *eval*.

Note You cannot nest the `%<expr>` operator (that is, `%<foo%<bar>>` is not allowed).

Note When you use the `%<expr>` operator within quotation marks, for example, `"%<Top.Date>"`, TLC expands the expression and then encloses the result in quotation marks. However, placing `%assign` within quotation marks, for example, `"%assign foo = 3"`, simply echoes the statement enclosed in quotation marks to the output stream. No assignment results (the value of `foo` remains unchanged or undefined).

General Assignment

The second section of output is produced by the script lines:

```
* Assign contents of a field to a variable, e.g.
%assign worker = Top.Employee.FirstName
  "%assign worker = Top.Employee.FirstName"
  worker expands to Top.Employee.FirstName = %<worker>
```

- Line 1 — Echoed to output.
- Line 2 — An assignment of field `FirstName` in the `Top.Employee` record scope to a new local variable called `worker`.
- Line 3 — Repeats the previous statement, producing output by enclosing it in quotation marks.
- Line 4 — Explains the following assignment and illustrates the token expansion. The token `%<worker>` expands to `Arthur`.

String Processing Plus

The next section of the script illustrates string concatenation, one of the uses of the "+" operator:

```
* Concatenate string values, e.g.
%assign worker = worker + " " + Top.Employee.LastName
  "%assign worker = worker + " " + Top.Employee.LastName"
  worker expands to worker + " " + Top.Employee.LastName = "%<worker>"
```

- Line 1 — Echoed to output.
- Line 2 — Performs the concatenation.
- Line 3 — Echoes line 2 to the output.
- Line 4 — Describes the operation, in which a variable is concatenated to a field separated by a space character. An alternative way to do this, without using the + operator, is

  ```
  %assign worker = "%<Top.Employee.FirstName> %<Top.Employee.LastName>"
  ```

  The alternative method uses evals of fields and is equally efficient.

The + operator, which is associative, also works for numeric types, vectors, matrices, and records:

- Numeric Types — Add two expressions together; both operands must be numeric. For example:

  Output:

  ```
  * Numeric Type example, e.g.
    Top.Employee.PayRate = 11.5
    Top.Employee.Overhead = 1.78
    td = Top.Employee.PayRate + Top.Employee.Overhead
    td evaluates to 13.28
  ```

- Vectors — If the first argument is a vector and the second is a scalar value, TLC appends the scalar value to the vector. For example:

  Output:

  ```
  * Vector example, e.g.
    v1 is [0, 1, 2, 3]
    Top.Project[1].Difficulty is 8
    v2 = v1 + Top.Project[1].Difficulty
    v2 evaluates to: [0, 1, 2, 3, 8]
  ```

- Matrices — If the first argument is a matrix and the second is a vector of the same column-width as the matrix, TLC appends the vector as another row to the matrix. For example:

  Output:

  ```
  * Matrices example, e.g.
    mx1 is [ [4, 5, 6, 7]; [8, 9, 10, 11] ]
    v1 is [0, 1, 2, 3]
    mx = mx1 + v1
    mx evaluates to [ [4, 5, 6, 7]; [8, 9, 10, 11]; [0, 1, 2, 3] ]
  ```

- Records — If the first argument is a record, TLC adds the second argument as a parameter identifier (with its current value). For example:

  Output:

  ```
  * Record example, e.g.
    StartDate is August 28, 2008
    Top + StartDate
    Top.StartDate evaluates to August 28, 2008
  ```

Arithmetic Operations

TLC provides a full complement of arithmetic operators for numeric data. In the next portion of our TLC script, two numeric fields are multiplied:

```
* Perform arithmetic operations, e.g.
%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead
  "%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead"
  wageCost expands to Top.Employee.PayRate * Top.Employee.Overhead ...
 <- %<Top.Employee.PayRate> * %<Top.Employee.Overhead> = %<wageCost>
```

- Line 1 — Echoed to output.
- Line 2 — `%assign` statement that computes the value, which TLC stores in local variable `wageCost`.
- Line 3 — Echoes the operation in line 2.
- Lines 4 and 5 — Compose a single statement. The ellipsis (typed as three consecutive periods, for example, `...`) signals that a statement is continued on the following line, but if the statement has output, TLC does not insert a line break. To continue a statement and insert a line break, replace the ellipsis with a backslash (`\`).

Modify Records

Once read into memory, you can modify and manipulate records just like variables you create by assignment. The next segment of `read-guide.tlc` replaces the value of record field `Top.Employee.GrossRate`:

```
* Put variables into a field, e.g.
%assign Top.Employee.GrossRate = wageCost
  "%assign Top.Employee.GrossRate = wageCost"
  Top.Employee.GrossRate expands to wageCost = %<Top.Employee.GrossRate>
```

Such changes to records are nonpersistent (because record files are inputs to TLC; other file types, such as C source code, are outputs), but can be useful.

You can use several TLC directives besides `%assign` to modify records:

`%createrecord`	Creates new top-level records, and might also specify subrecords within them, including name/value pairs.
`%addtorecord`	Adds fields to an existing record. The new fields can be name/value pairs or aliases to existing records.
`%mergerecord`	Combines one or more records. The first record contains itself plus copies of the other records' contents specified by the command, in sequence.
`%copyrecord`	Creates a new record as `%createrecord` does, except the components of the record come from the existing record you specify.
`%undef var`	Removes (deletes) `var` (a variable or a record) from scope. If `var` is a field in a record, TLC removes the field from the record. If `var` is a record array (list), TLC removes the first element of the array; the remaining elements remain accessible. You can remove only records you create with `%createrecord` or `%copyrecord`.

See “Target Language Compiler Directives” on page 19-2 for details on these directives.

Index Lists

Record files can contain lists, or sequences of records having the same identifier. Our example contains a list of three records identified as `Project` within the `Top` scope. List references are indexed, numbered from 0, in the order in which they appear in the record file. Here is TLC code that compiles data from the `Name` field of the `Project` list:

```
* Index lists of values, e.g.
%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name...
+ ", " + Top.Project[2].Name
  "%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name..."
  "+ ", " + Top.Project[2].Name"
  projects expands to Top.Project[0].Name + ", " + Top.Project[1].Name
  + ", " + Top.Project[2].Name = %<projects>
```

The `Scope.Record[n].Field` syntax is similar to that used in C to reference elements in an array of structures.

While explicit indexing, such as the above, is perfectly acceptable, it is often preferable to use a loop construct when traversing entire lists, as shown in “Loop Over Lists” on page 16-10.

Loop Over Lists

By convention, the section of a record file that a list occupies is preceded by a record that indicates how many list elements are present. In *`model`*`.rtw` files, such parameters are declared as `Num`*`Ident`*, where *`Ident`* is the identifier used for records in the list that follows. In `guide.rtw`, the `Project` list looks like this:

```
  NumProject    3                  # Indicates length of following list
  Project {                        # First list item, called Top.Project[0]
    Name       "Tea"               #   Alpha field Name, Top.Project[0].Name
    Difficulty  3                  #   Numeric field Top.Project[0].Difficulty
  }                                # End of first list item
  Project {                        # Second list item, called Top.Project[1]
    Name       "Gillian"           #   Alpha field Name, Top.Project[1].Name
    Difficulty  8                  #   Numeric field Top.Project[1].Difficulty
  }                                # End of second list item
  Project {                        # Third list item, called Top.Project[2]
    Name       "Zaphod"            #   Alpha field Name, Top.Project[2].Name
```

```
    Difficulty  10                  #   Numeric field Top.Project[2].Difficulty
  }                                 # End of third list item
```

Thus, the value of `NumProject` describes how many `Project` records occur.

Note `model.rtw` files might also contain records that start with `Num` but are not list-size parameters. TLC does not require that list size parameters start with `Num`. Therefore you need to be cautious when interpreting `Num`*`Ident`* record identifiers. The built-in TLC function `SIZE()` can determine the number of records in a specified scope, hence the length of a list.

The last segment of `read-guide.tlc` uses a `%foreach` loop, controlled by the `NumProject` parameter, to iterate the `Project` list and manipulate its values.

```
* Traverse and manipulate list data via loops, e.g.
%assign diffSum = 0.0
%foreach i = Top.NumProject
  - At top of Loop, Project = %<Top.Project[i].Name>; Difficulty =...
 %<Top.Project[i].Difficulty>
  %assign diffSum = diffSum + Top.Project[i].Difficulty
  - Bottom of Loop, i = %<i>; diffSum = %<diffSum>
%endforeach
%assign avgDiff = diffSum / Top.NumProject
  Average Project Difficulty expands to diffSum / Top.NumProject = %<diffSum> ...
 / %<Top.NumProject> = %<avgDiff>
```

As you may recall, the TLC output looks like this:

```
* Traverse and manipulate list data via loops, e.g.
- At top of Loop, Project = Tea; Difficulty = 3
  - Bottom of Loop, i = 0; diffSum = 3.0
  - At top of Loop, Project = Gillian; Difficulty = 8
  - Bottom of Loop, i = 1; diffSum = 11.0
  - At top of Loop, Project = Zaphod; Difficulty = 10
  - Bottom of Loop, i = 2; diffSum = 21.0
Average Project Difficulty expands to diffSum / Top.NumProjects = 21.0 / 3 = 7.0
```

After initializing the summation variable `diffSum`, a `%foreach` loop is entered, with variable `i` declared as the loop counter, iterating up to `NumProject`. The scope of the loop is all statements encountered until the corresponding `%endforeach` is reached (`%foreach` loops may be nested).

Note Loop iterations implicitly start at zero and range to one less than the index that specifies the upper bound. The loop index is local to the loop body.

Modify read-guide.tlc

Now that you have studied `read-guide.tlc`, it is time to modify it. This exercise introduces two important TLC facilities, *file control* and *scoping control*. You implement both within the `read-guide.tlc` script.

File Control Basics

TLC scripts almost invariably produce output in the form of streams of characters. Output is normally directed to one or more buffers and files, collectively called *streams*. So far, you have directed output from `read-guide.tlc` to the MATLAB Command Window because you included the `-v` switch on the command line. Prove this by omitting `-v` when you run `read-guide.tlc`. Type

```
tlc -r guide.rtw read-guide.tlc
```

Nothing appears to happen. In fact, the script was executed, but output was directed to a null device (sometimes called the “bit bucket”).

There is one active output file, even if it is null. To specify, open, and close files, use the following TLC directives:

```
%openfile streamid ="filename" , "mode"
%closefile streamid
%selectfile streamid
```

If you do not give a file name, subsequent output flows to the memory buffer named by `streamid`. If you do not specify a mode, TLC opens the file for writing and deletes any existing content (subject to system-level file protection mechanisms). Valid mode identifiers are `a` (append) and `w` (write, the default). Enclose these characters in quotes.

The `%openfile` directive creates a file/buffer (in `w` mode), or opens an existing one (in `a` mode). Note the required equals sign for file specification. Multiple streams can be open for writing, but only one can be active at one time. To switch output streams, use the `%selectfile` directive. You do not need to close files until you are done with them.

The default output stream, which you can respecify with the stream ID `NULL_FILE`, is `null`. Another built-in stream is `STDOUT`. When activated using `%selectfile`, `STDOUT` directs output to the MATLAB Command Window.

Note The streams `NULL_FILE` and `STDOUT` are always open. Specifying them with `%openfile` generates errors. Use `%selectfile` to activate them.

The directive `%closefile` closes the current output file or buffer. Until an `%openfile` or a `%selectfile` directive is encountered, output goes to the previously opened stream (or, if none exists, to null). Use `%selectfile` to designate an open stream for reading or writing. In practice, many TLC scripts write pieces of output data to separate buffers, which are then selected in a sequence and their contents spooled to one or more files.

Implement Output File Control

In your `tlctutorial/guide` folder, find the file `read-guide-file-src.tlc`. The supplied version of this file contains comments and three lines of text added. Edit this file to implement output file control, as follows:

1. Open `read-guide-file-src.tlc` in your text editor.
2. Save the file as `read-guide-file.tlc`.
3. Note five comment lines that begin with `%% ->`.

   Under each of these comments, insert a TLC directive as indicated.
4. Save the edited file as `read-guide-file.tlc`.
5. Execute `read-guide-file.tlc` with the following command:

   ```
   tlc -r guide.rtw read-guide-file.tlc
   ```

   If you succeeded, TLC creates the file `guidetext.txt` which contains the expected output, and the MATLAB Command Window displays

```
*** Output being directed to file: guidetext.txt
*** We're almost done . . .
*** Processing completed.
```

If you did not see these messages, or if a text file was not produced, review the material and try again. If problems persist, inspect `read-guide-file.tlc` in the `guide/solutions` subfolder to see how you should specify file control.

Scope Basics

"Structure of Record Files" on page 16-4 explains the hierarchical organization of records. Each record exists within a scope defined by the records in which it is nested. The example file, `guide.rtw`, contains the following scopes:

```
Top
Top.Employee
Top.Project[0]
Top.Project[1]
Top.Project[2]
```

To refer to a field or a record, specify its scoping, even if no other context that contains the identifier exists. For example, in `guide.rtw`, the field `FirstName` exists only in the scope `Top.Employee`. You must refer to it as `Top.Employee.FirstName` whenever accessing it.

When models present scopes that are deeply nested, this can lead to extremely long identifiers that are tedious and error prone to type. For example:

```
CompiledModel.BlockOutputs.BlockOutput.ReusedBlockOutput
```

This identifier has a scope that is long and has similar item names that you could easily enter incorrectly.

The `%with/%endwith` directive eases the burden of coding TLC scripts and clarifies their flow of control. The syntax is

```
%with RecordName
  [TLC statements]
%endwith
```

Every `%with` is eventually followed by an `%endwith`, and these pairs might be nested (but not overlapping). If `RecordName` is below the top level, you need not include the top-level scope in its description. For example, to make the current scope of `guide.rtw` `Top.Employee`, you can specify

```
%with Employee
  [TLC statements]
%endwith
```

Naturally, `%with Top.Employee` is also valid syntax. Once bracketed by `%with/%endwith`, record identifiers in TLC statements do not require you to specify their outer scope. However, note the following conditions :

- You can access records outside of the current `%with` scope, but you must qualify them fully (for example, using record name and fields).
- Whenever you make assignments to records inside a `%with` directive, you must qualify them fully.

Change Scope Using %with

In the last segment of this exercise, you modify the TLC script by adding a `%with/%endwith` directive. You also need to edit record identifier names (but not those of local variables) to account for the changes of scope resulting from the `%with` directives.

1. Open the TLC script `read-guide-scope-src.tlc` in the text editor.
2. Save the file as `read-guide-scope.tlc`.
3. Note comment lines that commence with `%% ->`.

   Under each of these comments, insert a TLC directive or modify statements already present, as indicated.
4. Save the edited file as `read-guide-scope.tlc`.
5. Execute `read-guide-scope.tlc` with the following command:

   ```
   tlc -v -r guide.rtw read-guide-scope.tlc
   ```

The output should be exactly the same as from `read-guide.tlc`, except possibly for white space that you might have introduced by indenting sections of code inside `%with/%endwith` or by eliminating blank lines.

Fully specifying a scope inside a `%with` context is not an error, it is simply unnecessary. However, failing to fully specify its scope when assigning it to a record (for example, `%assign GrossRate = wageCost`) is invalid.

If errors result from running the script, review the discussion of scoping above and edit `read-guide-scope.tlc` to eliminate them. As a last resort, inspect `read-guide-scope.tlc` in the `/solutions` subfolder to see how you should have handled scoping in this exercise.

For additional information, see “Scopes in the model.rtw File” on page 18-2 and “Variable Scoping” on page 19-39.

Pass and Use a Parameter

You can use the TLC commands and built-in functions to pass parameters from the command line to the TLC file being executed. The most general command switch is `-a`, which assigns arbitrary variables. For example:

```
tlc -r input.rtw -avar=1 -afoo="abc" vars.tlc
```

The result of passing this pair of strings via `-a` is the same as declaring and initializing local variables in the file being executed (here, `vars.tlc`). For example:

```
%assign var = 1
%assign foo = "abc"
```

You do not need to declare such variables in the TLC file, and they are available for use when set with `-a`. However, errors result if the code assigns undeclared variables that you do not specify with the `-a` switch when invoking the file. Also note that (in contrast to the `-r` switch) a space should not separate `-a` from the parameter you are declaring.

In the final section of this tutorial, you use the built-in function `GET_COMMAND_SWITCH()` to print the name of the record file being used in the TLC script, and provide a parameter to control whether or

not the code is suppressed. By default the code is executed, but is suppressed if the command line contains `-alist=0`:

1. Open the TLC script `read-guide-param-src.tlc` in your text editor.
2. Save the file as `read-guide-param.tlc`.
3. To enable your program to access the input filename from the command line, do the following:

   a. Below the line `%selectfile STDOUT`, add the line:

   ```
   %assign inputfile = GET_COMMAND_SWITCH ("r")
   ```

   The `%assign` directive declares and sets variables. In this instance, it holds a string filename identifier. `GET_COMMAND_SWITCH()` returns whatever string argument follows a specified TLC command switch. You must use UPPERCASE for built-in function names.

   b. Change the line "`*** WORKING WITH RECORDFILE`" to read as follows:

   ```
   *** WORKING WITH RECORDFILE %<inputfile>
   ```

4. To control whether or not a section of TLC code is executed, do the following:

   a. Below the line "`%assign inputfile = GET_COMMAND_SWITCH ("r")`", add:

   ```
   %if (!EXISTS(list))
       %assign list = 1
   %endif
   ```

   The program checks whether a list parameter has been declared, via the intrinsic (built-in) function `EXISTS()`. If no list variable exists, the program assigns one. This defines `list` and by default its value is `TRUE`.

   b. Enclose lines of code within an `%if` block.

   ```
   %if (list)
       * Assign contents of a field to a variable, e.g.
         %assign worker = FirstName
         "%assign worker = FirstName"
         worker expands to FirstName = %<worker>
   %endif
   ```

   Now the code to assign `worker` is sent to the output only when `list` is `TRUE`.

   c. Save `read-guide-param.tlc`.

5. Execute `read-guide-param.tlc` and examine the output, using the command

   ```
   tlc -r guide.rtw read-guide-param.tlc
   ```

   This yields

   ```
   *** WORKING WITH RECORDFILE [guide.rtw]
   * Assign contents of a field to a variable, e.g.
         "%assign worker = FirstName"
         worker expands to FirstName = Arthur
   ***END
   ```

6. Execute`read-guide-param.tlc` with the command:

   ```
   tlc -r guide.rtw -alist=0 read-guide-param.tlc
   ```

   With the `-alist=0` switch, the output displays only the information outside of the `if` statement.

```
*** WORKING WITH RECORDFILE [guide.rtw]
***END
```

Review

The preceding exercises examined the structure of record files, and expanded on how to use TLC directives. The following TLC directives are commonly used in TLC scripts (see "Target Language Compiler Directives" on page 19-2 for detailed descriptions):

`%addincludepath`	Enable TLC to find included files.
`%addtorecord`	Add fields to existing record. New fields can be name/value pairs or aliases to existing records.
`%assign`	Create or modify variables.
`%copyrecord`	Create new record and, if applicable, specify subrecords within them, including name/value pairs. The components of the record come from the existing record specified.
`%createrecord`	Create new top-level records and, if applicable, specify subrecords within them, including name/value pairs.
`%foreach/%endforeach`	Iterate loop variable from 0 to upper limit.
`%if/%endif`	Control whether code is executed, as in C.
`%include`	Insert one file into another, as in C.
`%mergerecord`	Combine one or more records. The first record contains itself plus copies of the other records contents specified by the command, in sequence.
`%selectfile`	Direct outputs to a stream or file.
`%undef var`	Remove (delete) `var` (a variable or a record) from scope. If `var` is a field in a record, TLC removes the field from the record. If `var` is a record array (list), TLC removes the first element of the array; the remaining elements remain accessible. Only records created via `%createrecord` or `%copyrecord` can be removed.
`%with/%endwith`	Add scope to simplify referencing blocks.

See Also

Related Examples

- "Target Language Compiler Directives" on page 19-2

Inline S-Functions with TLC

timesN Tutorial Overview

Objective: To understand how TLC works with an S-function.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/timesN (open)

In this tutorial, you generate versions of C code for existing S-function `timesN`.

The tutorial includes these steps:

1. **Noninlined Code Generation** — Via SimStructs and generic API
2. **Why Use TLC to Inline S-Functions?** — Benefits of inlining
3. **Create an Inlined S-Function** — Via custom TLC code

A later tutorial provides information and practice with “wrapping” S-functions.

Noninlined Code Generation

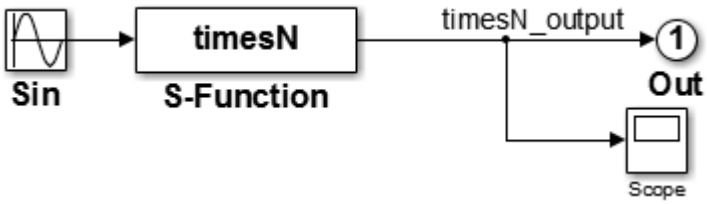
The tutorial folder `tlctutorial/timesN` in your working folder contains Simulink S-function `timesN.c`.

In this exercise, you generate noninlined code from the model `sfun_xN`.

1. Find the file `rename_timesN.tlc` in `tlctutorial/timesN`. Rename this file to `timesN.tlc`. This allows you to generate code.
2. In the MATLAB Command Window, create a MEX-file for the S-function:

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with Simulink.
3. Open the model `sfun_xN`, which uses the `timesN` S-function. The block diagram looks like this.



4. Open the Configuration Parameters dialog box and select the **Solver** pane.
5. Set **Stop time** to `10.0`.
6. Set the **Solver Options**.
   - **Type** to `Fixed-step`
   - **Solver** to `Discrete (no continuous states)`
   - **Fixed-step size** to `0.01`
7. Select the **Optimization** pane, and make sure that **Default parameter behavior** is set to `Tunable`.

8. Select the **Code Generation** > **Comments** pane, and notice that **Include comments** is checked by default.
9. Select the **Code Generation** pane and check **Generate code only**.

   Click **Apply**.
10. Press **Ctrl+B** to generate C code for the model.
11. Open the resulting file `sfun_xN_grt_rtw/sfun_xN.c` and view the `sfun_xN_output` portion, shown below.

```
/* Model output function */
static void sfun_xN_output(int_T tid)
{
  /* Sin: '<Root>/Sin' */
  sfun_xN_B.Sin = sin(sfun_xN_M->Timing.t[0] * sfun_xN_P.Sin_Freq +
                      sfun_xN_P.Sin_Phase) * sfun_xN_P.Sin_Amp +
    sfun_xN_P.Sin_Bias;

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_xN_B.timesN_output = sfun_xN_B.Sin * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_xN_Y.Out = sfun_xN_B.timesN_output;
  UNUSED_PARAMETER(tid);
}
```

Comments appear in the code because, in the **Code Generation** > **Comments** pane of the Configuration Parameters dialog box, **Include comments** is selected by default.

Why Use TLC to Inline S-Functions?

The code generator includes a generic API that you can use to invoke user-written algorithms and drivers. The API includes a variety of callback functions — for initialization, output, derivatives, termination, and so on — as well as data structures. Once coded, these are instantiated in memory and invoked during execution via indirect function calls. Each invocation involves stack frames and other overhead that adds to execution time.

In a real-time environment, especially when many solution steps are involved, generic API calls can be unacceptably slow. The code generator can speed up S-functions in standalone applications that it generates by embedding user-written algorithms within auto-generated functions, rather than indirectly calling S-functions via the generic API. This form of optimization is called inlining. TLC inlines S-functions, resulting in faster, optimized code.

You should understand that TLC is not a substitute for writing C code S-functions. To invoke custom blocks within Simulink, you still have to write S-functions in C (or as MATLAB files), since simulations do not make use of TLC files. You can, however, prepare TLC files that inline specified S-functions to make your target code much more efficient.

Create an Inlined S-Function

TLC creates an inlined S-function whenever it detects a `.tlc` file with the same name as an S-function. Assuming the `.tlc` file has the expected form, it directs construction of code that functionally duplicates the external S-function without incurring API overhead. See how this process works by completing the following steps:

1. If you have not done so already, find the file `rename_timesN.tlc` in `tlctutorial/timesN`. Rename this file to `timesN.tlc`, so you can use it to generate code. The executable portion of the file is

```
%implements "timesN" "C"

%% Function: Outputs ============================================================
%%
%function Outputs(block, system) Output
  %assign gain =SFcnParamSettings.myGain
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
  %endroll

%endfunction
```

2. Create the inline version of the S-function.

   a. On the **Optimization** pane of the Configuration Parameters dialog box, set **Default parameter behavior** to `Inlined`, and click **Apply**.

   b. Change the diagram's label from `model: sfun_xN` to `model: sfun_xN_ilp`.

   c. Save the model as `sfun_x2_ilp`.

   d. Press **Ctrl+B**. Source files are created in a new subfolder called `sfun_xN_ilp_grt_rtw`.

   e. Inspect the code in generated file `sfun_xN_ilp.c`:

```
/* Model output function */
static void sfun_xN_ilp_output(int_T tid)
{
  /* Sin: '<Root>/Sin' */
  sfun_xN_ilp_B.Sin = sin(sfun_xN_ilp_M->Timing.t[0]);

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_xN_ilp_B.timesN_output = sfun_xN_ilp_B.Sin * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_xN_ilp_Y.Out = sfun_xN_ilp_B.timesN_output;
  UNUSED_PARAMETER(tid);
}
```

   Note When the code generator produces code and builds executables, it creates or uses a specific subfolder (called the build folder) to hold source, object, and make files. By default, the build folder is named *`model`*`_grt_rtw`.

   Notice that setting **Default parameter behavior** to `Inlined` did not change the code. This is because TLC inlines S-functions.

3. Continue the exercise by creating a standalone simulation.

   a. In the **Code Generation** pane of the Configuration Parameters dialog box, clear **Generate code only** and click **Apply**.

   b. In the **Data Import/Export** pane of the Configuration Parameters dialog box, check **Output**.

This specification causes the model's output data to be logged in your MATLAB workspace.

c Press **Ctrl+B** to generate code, compile, and link the model into an executable, named `sfun_xN_ilp.exe` (or, on UNIX systems, `sfun_xN_ilp`).

d Confirm that the `timesN.tlc` file produces expected output by running the standalone executable. To run it, in the MATLAB Command Window, type

```
!sfun_xN_ilp
```

The following response appears:

```
** starting the model **
** created sfun_xN_ilp.mat **
```

e View or plot the contents of the `sfun_xN_ilp.mat` file to verify that the standalone model generated sine output ranging from -3 to +3. In the MATLAB Command Window, type

```
load sfun_xN_ilp.mat
plot (rt_yout)
```

Tip For UNIX platforms, run the executable program in the Command Window with the syntax `!./executable_name`. If preferred, run the executable program from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs".

See Also

Related Examples

- "Write Fully Inlined S-Functions"
- "Write Fully Inlined S-Functions with mdlRTW Routine"

Explore Variable Names and Loop Rolling

timesN Looping Tutorial Overview

Objective: This example shows how you can influence looping behavior of generated code.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/timesN (open)

Work with the model sfun_xN in tlctutorial/timesN. It has one source (a Sine Wave generator block), a times N gain block, an Out block, and a Scope block.

The tutorial guides you through following steps:

1. **Getting Started** — Set up the exercise and run the model
2. **Modify the Model** — Change the input width and see the results
3. **Change the Loop Rolling Threshold** — Change the threshold and see the results
4. **More About TLC Loop Rolling** — Parameterize loop behavior

Getting Started

1. Make tlctutorial/timesN your current folder, so that you can use the files provided.

   Note You must use or create a working folder outside of *matlabroot* for Simulink models you make. You cannot build models in source folders.

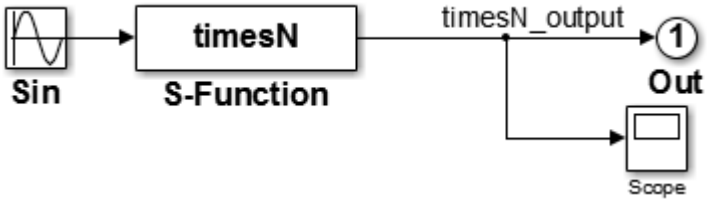
2. In the MATLAB Command Window, create a MEX-file for the S-function:

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with Simulink.

   Note An error might occur if you have not previously run mex -setup.

3. Open the model file sfun_xN.

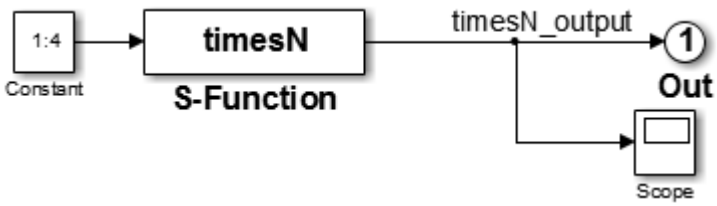


4. View the previously generated code in sfun_xN_grt_rtw/sfun_xN.c. Note that no loops exist in the code. This is because the input and output signals are scalar.

Modify the Model

1. Replace the Sine Wave block with a Constant block.
2. Set the parameter for the Constant block to 1:4, and change the top label, model: sfun_xN, to model: sfun_vec.

3 Save the edited model as `sfun_vec` (in `tlctutorial/timesN`). The model now looks like this.



4 Because the Constant block generates a vector of values, this is a vectorized model. Generate code for the model and view the `/*Model output function */` section of `sfun_vec.c` in your editor to observe how variables and `for` loops are handled. This function appears as follows:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_vec_B.timesN_output[0] = sfun_vec_P.Constant_Value[0] * 3.0;
  sfun_vec_B.timesN_output[1] = sfun_vec_P.Constant_Value[1] * 3.0;
  sfun_vec_B.timesN_output[2] = sfun_vec_P.Constant_Value[2] * 3.0;
  sfun_vec_B.timesN_output[3] = sfun_vec_P.Constant_Value[3] * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_vec_Y.Out[0] = sfun_vec_B.timesN_output[0];
  sfun_vec_Y.Out[1] = sfun_vec_B.timesN_output[1];
  sfun_vec_Y.Out[2] = sfun_vec_B.timesN_output[2];
  sfun_vec_Y.Out[3] = sfun_vec_B.timesN_output[3];
  UNUSED_PARAMETER(tid);
}
```

Notice that there are four instances of the code that generates model outputs, corresponding to four iterations.

5 Set the parameter for the Constant block to 1:10, and save the model.

6 Generate code for the model and view the `/*Model output function */` section of `sfun_vec.c` in your editor to observe how variables and `for` loops are handled. This function appears as follows:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  {
    int_T i1;
    const real_T *u0 = &sfun_vec_P.Constant_Value[0];
    real_T *y0 = sfun_vec_B.timesN_output;
    for (i1=0; i1 < 10; i1++) {
      y0[i1] = u0[i1] * 3.0;
    }
  }

  {
    int32_T i;
    for (i = 0; i < 10; i++) {
      /* Outport: '<Root>/Out' */
      sfun_vec_Y.Out[i] = sfun_vec_B.timesN_output[i];
    }
  }

  UNUSED_PARAMETER(tid);
}
```

Notice that:

- The code that generates model outputs gets “rolled” into a loop. This occurs by default when the number of iterations exceeds 5.
- Loop index `i1` runs from 0 to 9.
- Pointer `*y0` is used and initialized to the output signal array.

Change the Loop Rolling Threshold

The code generator creates iterations or loops depending on the current value of the **Loop unrolling threshold** parameter.

The default value of **Loop unrolling threshold** is `5`. To change looping behavior for blocks in a model:

1. On the **Optimization** pane of the Configuration Parameters dialog box, set **Loop unrolling threshold** to `12` and click `Apply`.

   The parameter `RollThreshold` is now `12`. Loops will be generated only when the width of signals passing through a block exceeds 12.

   Note You cannot modify `RollThreshold` for specific blocks from the Configuration Parameters dialog box.

2. Press **Ctrl+B** to regenerate the output.
3. Inspect `sfun_vec.c`. It will look like this:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_vec_B.timesN_output[0] = sfun_vec_P.Constant_Value[0] * 3.0;
  sfun_vec_B.timesN_output[1] = sfun_vec_P.Constant_Value[1] * 3.0;
  sfun_vec_B.timesN_output[2] = sfun_vec_P.Constant_Value[2] * 3.0;
  sfun_vec_B.timesN_output[3] = sfun_vec_P.Constant_Value[3] * 3.0;
  sfun_vec_B.timesN_output[4] = sfun_vec_P.Constant_Value[4] * 3.0;
  sfun_vec_B.timesN_output[5] = sfun_vec_P.Constant_Value[5] * 3.0;
  sfun_vec_B.timesN_output[6] = sfun_vec_P.Constant_Value[6] * 3.0;
  sfun_vec_B.timesN_output[7] = sfun_vec_P.Constant_Value[7] * 3.0;
  sfun_vec_B.timesN_output[8] = sfun_vec_P.Constant_Value[8] * 3.0;
  sfun_vec_B.timesN_output[9] = sfun_vec_P.Constant_Value[9] * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_vec_Y.Out[0] = sfun_vec_B.timesN_output[0];
  sfun_vec_Y.Out[1] = sfun_vec_B.timesN_output[1];
  sfun_vec_Y.Out[2] = sfun_vec_B.timesN_output[2];
  sfun_vec_Y.Out[3] = sfun_vec_B.timesN_output[3];
  sfun_vec_Y.Out[4] = sfun_vec_B.timesN_output[4];
  sfun_vec_Y.Out[5] = sfun_vec_B.timesN_output[5];
  sfun_vec_Y.Out[6] = sfun_vec_B.timesN_output[6];
  sfun_vec_Y.Out[7] = sfun_vec_B.timesN_output[7];
  sfun_vec_Y.Out[8] = sfun_vec_B.timesN_output[8];
  sfun_vec_Y.Out[9] = sfun_vec_B.timesN_output[9];
  UNUSED_PARAMETER(tid);
}
```

4. To activate loop rolling again, change the **Loop unrolling threshold** to 10 (or less) on the **Optimization** pane.

Loop rolling is an important TLC capability for optimizing generated code. Take some time to study and explore its implications before generating code for production requirements.

More About TLC Loop Rolling

The following TLC `%roll` code is the `Outputs` function of `timesN.tlc`:

```
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
  %endroll
%endfunction %% Outputs
```

Arguments for %roll

The lines between `%roll` and `%endroll` may be either repeated or looped. The key to understanding the `%roll` directive is in its arguments:

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
```

Argument	Description
`sigIdx`	Specify the index into a (signal) vector that is used in the generated code. If the signal is scalar, when analyzing that block of the *`model`*`.rtw` file, TLC determines that only a single line of code is required. In this case, it sets `sigIdx` to `0` so as to access only the first element of a vector, and no loop is constructed.
`lcv`	A control variable generally specified in the `%roll` directive as `lcv = RollThreshold`. `RollThreshold` is a global (model-wide) threshold with the default value of 5. Therefore, whenever a block contains more than five contiguous and rollable variables, TLC collapses the lines nested between `%roll` and `%endroll` into a loop. If fewer than five contiguous rollable variables exist, `%roll` does not create a loop and instead produces individual lines of code.
`block`	This tells TLC that it is operating on block objects. TLC code for S-functions use this argument.
`"Roller"`	This, specified in `rtw/c/tlc/roller.tlc`, formats the loop. Normally you pass this as is, but other loop control constructs are possible for advanced uses (see `LibBlockInputSignal` in “Input Signal Functions” on page 22-7).

`rollVars`	Tells TLC what types of items should be rolled: input signals, output signals, and/or parameters. You do not have to use all of them. In a previous line, `rollVars` is defined using `%assign`.
	`%assign rollVars = ["U", "Y"]`
	This list tells TLC that it is rolling through input signals (`U`) and output signals (`Y`). In cases where blocks specify an array of parameters instead of a scalar parameter, `rollvars` is specified as
	`%assign rollVars = ["U", "Y", "P"]`

Input Signals, Output Signals, and Parameters

Look at the lines that appear between `%roll` and `%endroll`:

```
%<LibBlockOutputSignal(0, "", lcv, idx)> = \
%<LibBlockInputSignal (0, "", lcv, idx)> * 2.0;
```

The TLC library functions `LibBlockInputSignal` and `LibBlockOutputSignal` expand to produce scalar or vector identifiers that are named and indexed. `LibBlockInputSignal`, `LibBlockOutputSignal`, and a number of related TLC functions are passed four canonical arguments:

Argument	Description
first argument — `0`	Corresponds to the input port index for a given block. The first input port has index 0. The second input port has index 1, and so on.
second argument — `"  "`	An index variable reserved for advanced use. For now, specify the second argument as an empty string. In advanced applications, you may define your own variable name to be used as an index with `%roll`. In such a case, TLC declares this variable as an integer in a location in the generated code.
third argument — `lcv`	As described previously, `lcv = RollThreshold` is set in `%roll` to indicate that a loop be constructed whenever `RollThreshold` (default value of 5) is exceeded.
fourth argument — `sigIdx`	Enables TLC to handle special cases. In the event that the `RollThreshold` is *not* exceeded (for example, if the block is only connected to a scalar input signal) TLC does not roll it into a loop. Instead, TLC provides an integer value for the index variable in a corresponding line of “inline” code. Whenever the `RollThreshold` is exceeded, TLC creates a `for` loop and uses an index variable to access inputs, outputs and parameters within the loop.

See Also

Related Examples

- “Target Language Compiler Directives” on page 19-2

Debug Your TLC Code

tlcdebug Tutorial Overview

Objective: Introduces the TLC debugger. You will learn how to set breakpoints and familiarize yourself with TLC debugger commands.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug (open)

You can cause the TLC debugger to be invoked whenever the build process is invoked. In this tutorial, you use it to detect a bug in a `.tlc` file for a model called `simple_log`. The bug causes the generated code output from the standalone version of the model to differ from its simulation output. The tutorial guides you through following steps:

1. **Getting Started** — Run the model and inspect output
2. **Generate and Run Code from the Model** — Compare compiled results to original output
3. **Start the Debugger and Use Its Commands** — Things you can do with the debugger
4. **Debug `timesN.tlc`** — Find out what went wrong
5. **Fix the Bug and Verify** — Easy ways to fix bugs and verify fixes

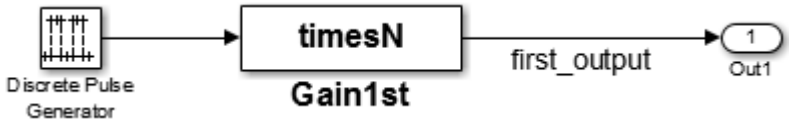
Getting Started

1. Copy the files from `tlctutorial/tlcdebug` to your current working directory.
2. In the MATLAB Command Window, create a MEX-file for the S-function:

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with your Simulink software.

   Note An error might occur if you have not previously run `mex -setup`.
3. Open the model `simple_log`. The model looks like this.



4. In the **Data Import/Export** pane of the Configuration Parameters dialog box, check **Time** and **Output**. This causes model variables to be logged to the MATLAB workspace.
5. Run the model. On the **Simulation** tab, click **Run**. Variables `tout` and `yout` appear in your MATLAB workspace.
6. Double-click `yout` in the **Workspace** pane of the MATLAB Command Window. The Variable Editor displays the 6x1 array output from `simple_log`. The display looks like this:

yout <6x1 double>

	1	2	3	4	5	6
1	3					
2	0					
3	3					
4	0					
5	3					
6	0					

Column 1 contains discrete pulse output for six time steps (3s and 0s), collected at port `out1`.

Next, you generate a standalone version of `simple_log`. You execute it and compare its results to the output from Simulink displayed above.

Note For the purpose of this exercise, the TLC file provided, `timesN.tlc`, contains a bug. This version must be in the same folder as the model that uses it.

Generate and Run Code from the Model

1. Press **Ctrl+B**.

   The code generator produces, compiles, and links C source code. The MATLAB Command Window shows the progress of the build, which ends with these messages:

   ```
   ### Created executable: simple_log.exe
   ### Successful completion of build procedure
       for model: simple_log
   ```

2. Run the standalone model just created by typing

   ```
   !simple_log
   ```

   This results in the messages

   ```
   ** starting the model **
   ** created simple_log.mat **
   ```

3. Inspect results by placing the variables in your workspace. In the **Current Folder** pane, double-click `simple_log.mat`, then double-click `rt_yout` (the standalone version of variable `yout`) in the **Workspace** pane.

   Compare `rt_yout` with `yout`. Do you notice differences? Can you surmise what caused values in `rt_yout` to change?

   A look at the generated C code that TLC placed in your build folder (`simple_log_grt_rtw`) helps to identify the problem.

4. Edit `simple_log.c` and look at its `MdlOutputs` function, which should appear as shown below:

   ```
   /* Model output function */
   static void simple_log_output(void)
   {
     /* DiscretePulseGenerator: '<Root>/Discrete Pulse Generator' */
     simple_log_B.DiscretePulseGenerator = (simple_log_DW.clockTickCounter < 1.0) &&
       (simple_log_DW.clockTickCounter >= 0) ? 1.0 : 0.0;
   ```

```
    if (simple_log_DW.clockTickCounter >= 2.0 - 1.0) {
      simple_log_DW.clockTickCounter = 0;
    } else {
      simple_log_DW.clockTickCounter++;
    }

    /* End of DiscretePulseGenerator: '<Root>/Discrete Pulse Generator' */

    /* S-Function (timesN): '<Root>/Gain1st' incorporates:
     *  Outport: '<Root>/Out1'
     */
    /* S-Function Block: <Root>/Gain1st */
    /* Multiply input by 3.0 */
    simple_log_Y.Out1 = simple_log_B.DiscretePulseGenerator * 1;
  }
```

Note the line near the end:

```
simple_log_B.first_output = simple_log_B.DiscretePulseGenerator * 1;
```

How did the incorrect product get assigned to the output when it was supposed to receive a variable that alternates between 3.0 and 0.0? Use the debugger to find out.

Start the Debugger and Use Its Commands

You use the TLC debugger to monitor the code generation process. As it is not invoked by default, you need to request the debugger explicitly.

1. Set up the TLC debugging environment and start to build the application:

   a. Select the **Configuration Parameters** > **Code Generation** pane, and select the options **Retain .rtw file** and **Start TLC debugger when generating code**. Click **OK**.

   b. Build the model.

      The MATLAB Command Window describes the building process. The build stops at the `timesN.tlc` file and displays the command prompt:

      ```
      TLC-DEBUG>
      ```

2. Type `help` to list the TLC debugger commands. Here are some things you can do in the debugger.

   - View and query various entities in the TLC scope.

     ```
     TLC-DEBUG> whos CompiledModel
     TLC-DEBUG> print CompiledModel.NumSystems
     TLC-DEBUG> print TYPE(CompiledModel.NumSystems)
     ```

   - Examine the statements in your current context.

     ```
     TLC-DEBUG> list
     TLC-DEBUG> list 10,40
     ```

   - Move to the next line of code.

     ```
     TLC-DEBUG> next
     ```

   - Step into a function.

     ```
     TLC-DEBUG> step
     ```

   - Assign a constant value to a variable, such as the input signal `%<u>`.

```
TLC-DEBUG> assign u = 5.0
```

- Set a breakpoint where you are or in some other part of the code.

```
TLC-DEBUG> break timesN.tlc:10
```

- Execute until the next breakpoint.

```
TLC-DEBUG> continue
```

- Clear breakpoints you have established.

```
TLC-DEBUG> clear 1
TLC-DEBUG> clear all
```

3. If you have tried the TLC debugger commands, execute the remaining code to finish the build process, then build `simple_log` again. The build stops at the `timesN.tlc` file and displays the command prompt:

```
TLC-DEBUG>
```

Debug timesN.tlc

Now look around to find out what is wrong with the code:

1. Set a breakpoint on line 20 of `timesN.tlc`.

```
TLC-DEBUG> break timesN.tlc:20
```

2. Instruct the TLC debugger to advance to your breakpoint.

```
TLC-DEBUG> continue
```

TLC processes input, reports its progress, advances to line 20 in `timesN.tlc`, displays the line, and pauses.

```
### Loading TLC function libraries
...
### Initial pass through model to cache user defined code
.
### Caching model source code
.
Breakpoint 1
00020:   %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
```

3. Use the `whos` command to see the variables in the current scope.

```
TLC-DEBUG> whos
Variables within: <BLOCK_LOCAL>
gain                            Real
rollVars                        Vector
block                           Resolved
system                          Resolved
```

4. Inspect the variables using the print command (names are case sensitive).

```
TLC-DEBUG> print gain
3.0

TLC-DEBUG> print rollVars
[U, Y]
```

5. Execute one step.

```
TLC-DEBUG> step
00021:     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
```

6. Because it is a built-in function, advance via the `next` command.

```
TLC-DEBUG> next
.
00022:     %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
```

This is the origin of the C statement responsible for the erroneous constant output, `simple_log_B.first_output = simple_log_B.DiscretePulseGenerator * 1;`.

7 Abandon the build by quitting the TLC debugger. Type

```
TLC-DEBUG> quit
```

An error message is displayed showing that you stopped the build by using the TLC debugger `quit` command. Close the error window.

Fix the Bug and Verify

The problem you identified is caused by evaluating a constant rather than a variable inside the TLC function `FcnEliminateUnnecessaryParams()`. This is a typical coding error and is easily repaired. Here is the code you need to fix.

```
%function Outputs(block, system) Output
   %assign gain =SFcnParamSettings.myGain
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
  %endroll

%endfunction

%% [EOF] timesN.tlc
```

1 To fix the coding error, edit `timesN.tlc`. The line

```
%<LibBlockInputSignal(0, "", lcv, idx)> * 1;
```

multiplies the evaluated input by 1. Change the line to

```
%<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
```

Save `timesN.tlc`.

2 Build the standalone model again. Complete the build by typing `continue` at each `TLC-DEBUG>` prompt.

3 Execute the standalone model by typing

```
!simple_log
```

A new version of `simple_log.mat` is created containing its output.

4 Load `simple_log.mat` and compare the workspace variable `rt_yout` with `yout`, as you did before. The values in the first column should now correspond.

See Also

Related Examples

- "TLC Code Coverage to Aid Debugging" on page 16-32

- “Using the TLC Debugger” on page 20-2
- “TLC Coverage” on page 20-6

TLC Code Coverage to Aid Debugging

tlcdebug Execute Tutorial Overview

Objective: Learn to use TLC coverage statistics to help identify bugs in TLC code.

Folder: *matlabroot*`/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug` (open)

This tutorial teaches you how to determine whether your TLC code is being executed as expected. Here it uses the same model as for the previous tutorial. As you focus on understanding flow of control in processing TLC files, you don't need to compile and execute a standalone model, only to look at code. The tutorial proceeds as follows:

1. **Getting Started** — Why and how to analyze TLC coverage
2. **Open the Model and Generate Code** — Read a coverage log file

Getting Started

The **Code Generation** > **Debug** pane provides the option **Start TLC coverage when generating code**. Selecting it results in a listing that documents how many times each line in your TLC source file was executed during code generation. The listing, *name*`.log` (where *name* is the filename of the TLC file being analyzed), is placed in your build folder.

Note A log file for every `.tlc` file invoked or included is generated in the build folder. Focus on `timesN.log`.

Open the Model and Generate Code

1. Copy the folder `tlctutorial/tlcdebug/` to your working folder and `cd` to it. *Do this even though you already have copied it*, to be sure you have the version of `timesN.tlc` that has the bug.
2. In the MATLAB Command Window, create a MEX-file for the S-function.

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with your Simulink software.
3. Open the model `simple_log`.
4. In the **Code Generation** pane of the Configuration Parameters dialog box, check **Generate code only**.
5. In the **Code Generation** > **Debug** pane of the Configuration Parameters dialog box, select **Start TLC coverage when generating code**. (Do not select **Start TLC debugger when generating code**. Invoking the debugger is unnecessary.) Click **Apply**.
6. Press **Ctrl+B**. The usual messages appear in the MATLAB Command Window, and a build folder (`simple_log_grt_rtw`) is created in your working folder.
7. Enter the build folder. Find the file `timesN.log`, and copy it to your working folder, renaming it to `timesN_ilp.log` to prevent it from being overwritten.

8. Open the log file `timesN_ilp.log` in your editor. It looks almost like `timesN.tlc`, except for a number followed by a colon at the beginning of each line. This number represents the number of times TLC executed the line in generating code. The code for `Outputs()` should look like this:

```
0: %% Function: Outputs ========================================================
     0: %%
     1: %function Outputs(block, system) Output
     1:   %assign gain =SFcnParamSettings.myGain
     1:   /* %<Type> Block: %<Name> */
     0:   %%
     1:   /* Multiply input by %<gain> */
     1:   %assign rollVars = ["U", "Y"]
     1:   %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
     1:     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
     1:     %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
     0:   %endroll
     1:
     0: %endfunction
```

   Notice that comments were not executed. TLC statements were reached, which means they output to the generated C code as many times as the number prefixed to those lines.

Changing code generation options can cause a latent issue in generated source code. Systematically changing options and observing the resulting differences in TLC coverage can facilitate the process of discovering faulty code.

See Also

Related Examples

- “Using the TLC Debugger” on page 20-2
- “TLC Coverage” on page 20-6

Wrap User Code with TLC

wrapper Tutorial Overview

Objective: Learn the architecture of wrapper S-functions and how to create an inlined wrapper S-function using TLC.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/wrapper (open)

Wrapper S-functions enable you to use existing C functions without fully rewriting them in the context of Simulink S-functions. Each wrapper you provide is an S-function "shell" that merely calls one or more existing, external functions. This tutorial explains and illustrates wrappers as follows:

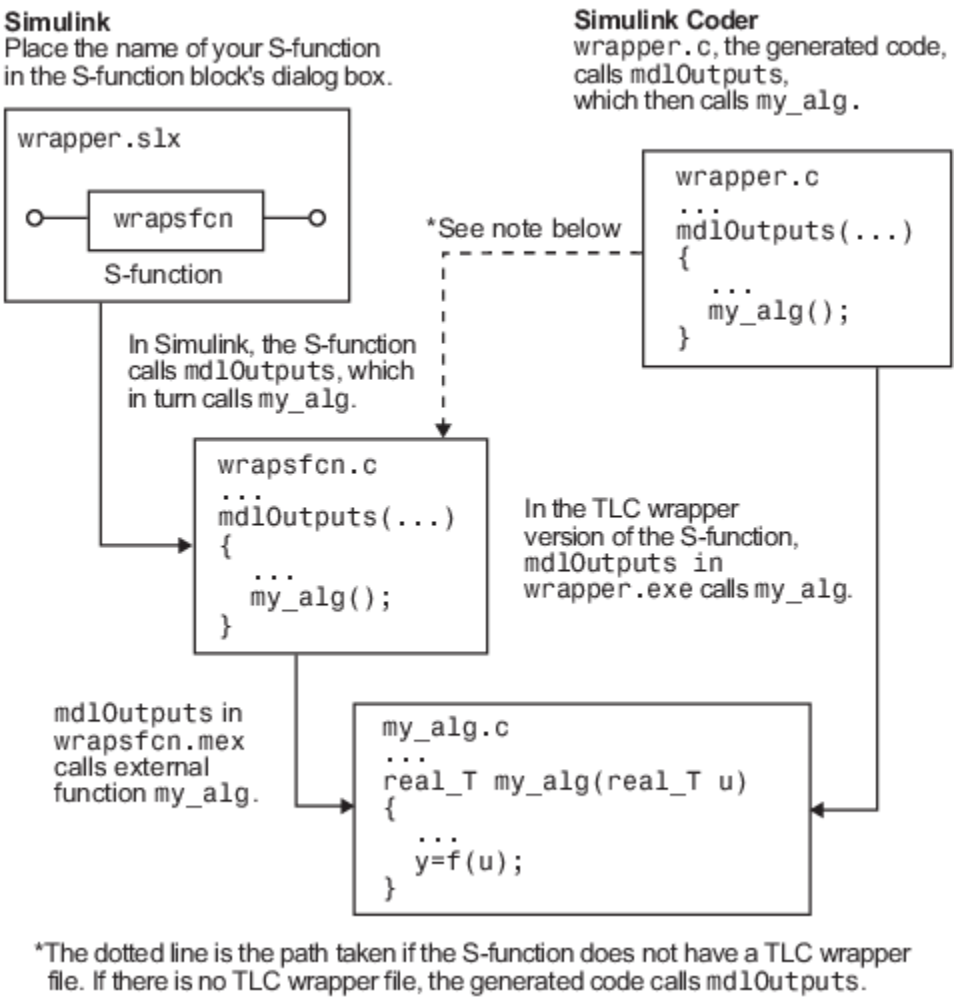
- **Why Wrap User Code?** — Reason for building TLC wrapper functions
- **Getting Started** — Set up the wrapper exercise
- **Generate Code Without a Wrapper** — How the code generator handles external functions by default
- **Generate Code Using a Wrapper** — Bypass the API overhead

Why Wrap User Code?

Many Simulink users want to build models incorporating algorithms that they have already coded, implemented, and tested in a high-level language. Typically, such code is brought into Simulink as S-functions. To generate an external application that integrates user code, you can take several approaches:

- You can construct an S-function from user code that hooks it to the Simulink generic API. This is the simplest approach, but sacrifices efficiency for standalone applications.
- You can inline the S-function, reimplementing it as a TLC file. This improves efficiency, but takes time and effort, can introduce errors into working code, and leads to two sets of code to maintain for each algorithm, unless you use the Legacy Code Tool (see "Import Calls to External Code into Generated Code with Legacy Code Tool").
- You can inline the S-function via a TLC wrapper function. By doing so, you need to create only a small amount of TLC code, and the algorithm can remain coded in its existing form.

The next figure illustrates how S-function wrappers operate.



Wrapping a function eliminates the need to recode it, requiring only a bit of extra TLC code to integrate it. Wrappers also enable object modules or libraries to be used in S-functions. This may be the only way to deploy functions for which source code is unavailable, and also allows users to distribute models to others without divulging implementation details that may be proprietary.

For example, you might have an existing object file compiled for a processor on which Simulink does not run. You can write a dummy C S-function and use a TLC wrapper that calls the external function, despite not having its source code. You could similarly access functions in a library of algorithms optimized for the target processor. Accomplishing this requires making changes to a template makefile, or otherwise providing a means to link against the library.

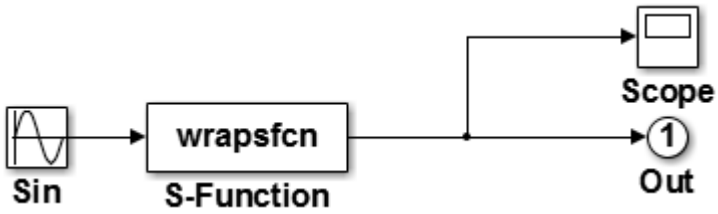
Note Object files that lack source code and are created with Microsoft Visual C and Microsoft Visual C++® Compiler (MSVC) work only with MSVC.

The only restriction on S-function wrappers is for the number of block inputs and outputs match number of inputs and outputs of the wrapped external function. Wrapper code may include computations, but usually these are limited to transforming values (for example, scaling or reformatting) passed to and from the wrapped external functions.

Getting Started

In the example folder, the “external function” is found in the file `my_alg.c`. You are also provided with a C S-function called `wrapsfcn.c` that integrates `my_alg.c` into Simulink. Set up the exercise as follows:

1. Make `tlctutorial/wrapper` your current folder.
2. In MATLAB, open the model `externalcode` from your working folder. The block diagram looks like this:



3. Activate the Scope block by double-clicking it.
4. Run the model (from the **Simulation** tab, or type **Ctrl+T**). You will get an error telling you that `wrapsfcn` does not exist. Can you figure out why?
5. The error occurs because a `mex` file does not exist for `wrapsfcn`. To rectify this, in the MATLAB Command Window type

   ```
   mex wrapsfcn.c
   ```

   Note An error might occur if you have not previously run `mex -setup`.
6. Run the simulation again with the S-function present.

The S-Function block multiplies its input by two. Looking at the Scope block, you see a sine wave that oscillates between -2.0 and 2.0. The variable `yout` that is created in your MATLAB workspace steps through these values.

In the remainder of the exercise, you build and run a standalone version of the model, then write some TLC code that allows the code generator to build a standalone executable that calls the S-function `my_alg.c` directly.

Generate Code Without a Wrapper

Before creating a wrapper, generate code that uses the Simulink generic API. The first step is to build a standalone model.

1. On the **C Code** tab, click **Build**.

   The code generator creates the standalone program in your working folder and places the source and object files in your build folder. The file will be called `externalcode.exe` on Microsoft Windows platforms or `externalcode` on UNIX platforms.

   As it generates the program, the code generator reports its progress in the MATLAB Command Window. The final lines are:

```
### Created executable: externalcode.exe
### Successful completion of build procedure
for model: externalcode
```

2 Run the standalone program to see that it behaves the same as the Simulink version. There should not be differences.

```
!externalcode

** starting the model **
** created externalcode.mat **
```

3 Notice this line in `wrapsfcn.c`:

```
#include "my_alg.c"
```

This pulls in the external function. That function consists entirely of

```
/*
 * Copyright 1994-2002 The MathWorks, Inc.
 */

double my_alg(double u)
{
    return(u * 2.0);
}
```

Inspect the `mdlOutputs()` function of the code in `wrapsfcn.c` to see how the external function is called.

```
static void mdlOutputs(SimStruct *S, int tid)
{
    int_T             i;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    int_T             width = ssGetOutputPortWidth(S,0);

    *y = my_alg(*uPtrs[0]);
}
```

Tip For UNIX platforms, run the executable program in the Command Window with the syntax `!./executable_name`. If preferred, run the executable program from an OS shell with the syntax `./executable_name`. For more information, see “Run External Commands, Scripts, and Programs”.

Generally, functions to be wrapped are either included in the wrapper, as above, or, when object modules are being wrapped, resolved at link time.

Generate Code Using a Wrapper

To create a wrapper for the external function `my_alg.c`, you need to construct a TLC file that embodies its calling function, `wrapsfcn.c`. The TLC file must generate C code that provides:

- A function prototype for the external function that returns a double, and passes the input `double u`.
- A function call to `my_alg()` in the outputs section of the code.

To create a wrapper for `my_alg()`, do the following:

1. Open the file `change_wrapsfcn.tlc` in your editor, and add lines of code where comments indicate to create a workable wrapper.
2. Save the edited file as `wrapsfcn.tlc`. It must have the same name as the S-function block that uses it or TLC is not called to inline code.
3. In MATLAB, open the model `externalcode` from your working folder. Activate the Scope block by double-clicking it, and run the model (from the **Simulation** tab, or type **Ctrl+T**). This gives you a baseline result.
4. Inform Simulink that your code has an external reference to be resolved. To update the model's parameters, in the MATLAB Command Window, do one of the following:
   - Type

     ```
     set_param('externalcode/S-Function','SFunctionModules','my_alg')
     ```
   - In the S-Function block parameters dialog box, in the `S-function modules` field, specify `'my_alg'`.
5. Create the standalone application, by entering one of the following commands in the Command Window:

   ```
   slbuild('my_alg','ForceTopModelBuild',true)
   ```

   ```
   slbuild('my_alg', 'StandaloneCoderTarget','ForceTopModelBuild',true)
   ```

   These commands force the code generator to rebuild the top model, which is required when you make changes associated with external or custom code.

   Alternatively, you can force regeneration of top model code by deleting folders in the "Code generation folder", such as `slprj` or the generated model code folder.

   For more information, see "Control Regeneration of Top Model Code".
6. Run the new standalone application and verify that it yields identical results as in the scope window.

   ```
   !externalcode
   ```

If you had problems building the application:

- Find the error messages and try to determine what files are at fault, paying attention to which step (code generation, compiling, linking) failed.
- Be sure you issued the `set_param()` command as specified above.
- Chances are that problems can be traced to your TLC file. It may be helpful to use TLC debugger to step through `wrapsfcn.tlc`.
- As a last resort, look at `wrapsfcn.tlc` in the `solutions/tlc_solution` folder, also listed below:

  ```
  %% File    : wrapsfcn.tlc
  %% Abstract:
  %%      Example tlc file for S-function wrapsfcn.c
  %%
  %% Copyright 1994-2002 The MathWorks, Inc.
  %%
  %%
  ```

```
%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup ================================
%% Abstract:
%%      Create function prototype in model.h as:
%%        "extern double my_alg(double u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
  %% ASSIGNMENT: PROVIDE A LINE OF CODE AS A FUNCTION PROTOTYPE
  %% FOR "my_alg" AS DESCRIBED IN THE WRAPPER TLC ASSIGNMENT
  extern double my_alg(double u);

  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs =======================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %% PROVIDE THE CALLING STATEMENT FOR "wrapfcn"
  %<y> = my_alg( %<u> );
%endfunction %% Outputs
```

Look at the highlighted lines. Did you declare `my_alg()` as `extern double`? Did you call `my_alg()` with the expected input and output? Fix mistakes and rebuild the model.

See Also

Related Examples

- “Write Wrapper S-Function and TLC Files”

Code Generation Architecture

- “Build Process” on page 17-2
- “Configure TLC” on page 17-5
- “Code Generation Concepts” on page 17-7
- “TLC Files” on page 17-11
- “Data Handling with TLC” on page 17-15

Build Process

Build Process Overview

TLC compiles files written in the target language. The target language is an interpreted language and the compiler operates on source files every time it executes. You can change a target file and watch the effects of your change the next time you build a model. You do not need to recompile TLC binary or other large binary to see the changes.

Because the target language is an interpreted language, some statements might not be compiled or executed (and hence not checked by the compiler). For example:

```
%if 1
   Hello
%else
   %<Invalid_function_call()>
%endif
```

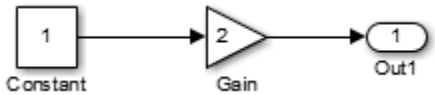
In this example, the `Invalid_function_call` statement is not executed. This example emphasizes that you should test TLC code with test cases that execute every line.

Create and Use Target Language File

This example creates a target language file that generates specific text from a model. It shows the sequence of steps that you should follow in creating and using your own target language files.

Process

To begin, create the Simulink model shown in the next figure.



1. Save the new model in a working folder as `basic`.
2. Display the Configuration Parameters dialog box.
3. Select the **Solver** pane.
4. In the **Solver** pane:
   a. Select `Fixed-step` in the **Type** field.
   b. Select `discrete (no continuous states)` in the **Solver** field.
   c. Under **Additional options**, specify `0.1` in the **Fixed-step size** field. (Otherwise, the code generator posts a warning and supplies a value when you generate code.)
5. Click **Apply**.
6. Select the **Code Generation** pane.
7. Select **Retain .rtw file**, then click **Apply**. This step lets you inspect the contents of the *`model`*`.rtw` file after the build finishes.
8. Select **Generate code only**, then click **OK**.
9. Build the model.

The build process generates code in the `basic_grt_rtw` folder. You can see the progress in the MATLAB Command Window. When code generation is complete, following message is displayed:

```
### Successful completion of code generation for model: basic
```

The slbuild Command

Invoke `slbuild` by pressing **Ctrl+B** in the model window. However, some circumstances require you to execute `slbuild` directly from the MATLAB prompt.

To generate a *`model`*`.rtw` file from the MATLAB prompt, type:

```
slbuild('model')
```

You can specify other options to `slbuild` that build or rebuild model reference simulation targets or a stand-alone executable. For more information, type:

```
help slbuild
```

at the MATLAB prompt or see `slbuild` in the Simulink documentation.

Viewing the basic.rtw File

A *`model`*`.rtw` file contains a hierarchy of labeled records and fields. Each record is delimited by brackets, and contains subordinate records and/or fields. The labels state the purpose of each record and field. The records and fields in the *`model`*`.rtw` file created for a model describe various details of the model and the Configuration Parameter settings that specify its context.

Open the file `./basic_grt_rtw/basic.rtw`, in MATLAB or a text editor.

Create the Target File

Note The following exercise is provided to give a conceptual overview of how the `.rtw` file is used in the build process. The code generator does not support manually invoking TLC with a `.rtw` file created from an earlier build. Also, the contents of the `.rtw` file are undocumented and subject to change. The `basic.tlc` file shows how information is provided in a `.rtw` file that can be accessed by the TLC files and executed as part of the build process.

Next, create a `basic.tlc` file to act as a target file for this model. Instead of generating code, simply display some information about the model using this file. The concept is the same as used in code generation.

Create a file called `basic.tlc` in the folder containing `basic`. This file should contain the following lines:

```
%with CompiledModel

My model is called %<Name>.
It was generated on %<GeneratedOn>.
It has %<NumModelOutputs> output(s) and %<NumContStates> continuous state(s).

%endwith
```

Note In the build process, the `.tlc` file specified on the command line when TLC is invoked (for example, `grt.tlc`) is referred to as the System Target File (STF). It can be selected via the **System target file** browser option in the **Code Generation** pane of the Configuration Parameters dialog box.

In this example, you generate the `.rtw` file as part of the build process and then manually run TLC using the file `basic.tlc` as an example STF. `basic.tlc` illustrates (in a limited capacity) how `.rtw` file information is used to generate an example output. To do this, enter at the MATLAB prompt:

```
set_param(0, 'RTWDataReferencesMinSize',10)
slbuild('basic')
tlc -r basic_grt_rtw/basic.rtw basic.tlc -v
```

The first line generates the `.rtw` file in the build folder `'basic_grt_rtw'`. This step is unnecessary because the file has already been generated in the previous step. However, it is useful if the model is changed and the operation has to be repeated.

The second line runs TLC on the file `basic.tlc`. The `-r` option tells TLC that it should use the file `basic.rtw` as the `.rtw` file. Note that a space must separate `-r` and the input filename. The `-v` option tells TLC to be verbose in reporting its activity.

The output of this pair of commands is (date will differ):

```
My model is called basic.
It was generated on Wed Jun 22 20:51:11 2005.
It has 1 output(s) and 0 continuous state(s).
```

You can also try changing the model (for instance, by using `rand(2,2)` as the value for the constant block) and then repeating the process to see how the output of TLC changes.

See Also

Related Examples

- “Configure TLC” on page 17-5
- “TLC Files” on page 17-11

Configure TLC

In this section...
“Set Command-Line Arguments” on page 17-5 “Configure for TLC Debugging” on page 17-6

Set Command-Line Arguments

You can enter TLC command-line arguments from the MATLAB command line using the `set_param` command, the model parameter `TLCOptions`, and the TLC option `-a`. For example, to enter the TLC command-line string `-amyConfigVariable=1`, use the following MATLAB command:

```
set_param(modelName,'TLCOptions','-amyConfigVariable=1');
```

Using `-amyConfigVariable=1` is equivalent to coding the following in your target file:

```
%assign myConfigVariable = 1
```

Alternatively, you can configure the TLC code generation process by using the `-a` option on the TLC command line. That is, you must give the TLC command interactively.

You can repeatedly use the `-a` option.

For an example of how this process works, consider the following TLC code fragment:

```
%if !EXISTS(myConfigVariable)
  %assign myConfigVariable = 0
%endif
  %if (myConfigVariable == 1)
      code fragment 1
  %else
      code fragment 2
  %endif
```

If you specify `-amyConfigVariable=1` in the command line, `code fragment 1` is generated; otherwise `code fragment 2` is generated. The `if` block starting with

```
%if !EXISTS(myConfigVariable)
```

serves to set the default value of `myConfigVariable` to 0, so that TLC does not generate an error if you forget to add `-amyConfigVariable` to the command line.

If you use the `-a` option to input a string variable, the variable must be enclosed in double quotation marks:

```
-aMyStringVariable="hello"
```

However, if the string contains white space, enclose the string within apostrophes and double quotation marks:

```
-aMyStringVariable="'hello world'"
```

Do this if apostrophes exist within the string, whether or not white space is included, and the apostrophes must be escaped (doubled):

```
-aMyStringVariable="''can''t'"
```

Configure for TLC Debugging

To configure TLC for debugging via the Configuration Parameters dialog box, search for the option **Start TLC debugger when generating code**. To activate TLC debugger, select **Start TLC debugger when generating code**. For more information, see “Using the TLC Debugger” on page 20-2 and the debugging topics in “Target Language Compiler”.

See Also

Related Examples

- “Build Process” on page 17-2
- “TLC Files” on page 17-11

Code Generation Concepts

TLC interprets a target language, which is a general programming language, and you can use it as such. It is important, however, to remember that TLC was designed for one purpose: to convert a *model*`.rtw` file to generated code. Thus, the target language provides many features that are useful for this task but does not provide some of the features that other languages like C and C++ provide.

You might find some of the following general programming examples useful to familiarize yourself with the basic constructs used within TLC.

Output Streams

The typical “Hello World” example is rather simple in the target language. Type the following in a file named `hello.tlc`:

```
%selectfile STDOUT
Hello, World
```

To run this TLC program, type

```
tlc hello.tlc
```

at the MATLAB prompt.

This simple script illustrates some important concepts underlying the purpose (and hence the design) of TLC. Since the primary purpose of TLC is to generate code, it is output (or stream) oriented. It makes it easy to handle buffers of text and output them easily. In the above script, the `%selectfile` directive tells TLC to send any following text that it generates or does not recognize to the standard output device. Syntax that TLC recognizes begins with the `%` character. Because `Hello, World` is not recognized, it is sent directly to the output. You could easily change the output destination to be a file. Do not open the `STDOUT` stream, but select to write to the Command Window.

```
%openfile foo = "foo.txt"
%openfile bar = "bar.txt"
%selectfile foo
This line is in foo.
%selectfile STDOUT
Line has been output to foo.
%selectfile bar
This line is in bar.
%selectfile NULL_FILE
This line will not show up anywhere.
%selectfile STDOUT
About to close bar.
%closefile bar
%closefile foo
```

You can switch between buffers to display status messages. The semantics of the three directives `%openfile`, `%selectfile`, and `%closefile` are given in “Target Language Compiler Directives” on page 19-2.

Variable Types

The absence of explicit type declarations for variables is another feature of TLC. See “Target Language Compiler” for more information on the implicit data types of variables.

Records

One of the constructs most relevant to generating code from the *model*.rtw file is a record. A record is similar to a structure in C or a record in Pascal. The syntax of a record declaration is

```
%createrecord recVar { ...
   field1        value1  ...
   field2        value2 ...
      ...
  fieldN        valueN ...
}
```

where `recVar` is the name of the record being declared, `fieldi` is a string, and `valuei` is the corresponding TLC value.

Records can have nested records, or subrecords, within them. The *model*.rtw file is essentially one large record, named `CompiledModel`, containing levels of subrecords.

Unlike MATLAB, TLC requires that you explicitly load function definitions not located in the same target file. In MATLAB, the line `A = myfunc(B)` causes MATLAB to automatically search for and load a MATLAB file or MEX-file named `myfunc`. TLC requires that you specifically include the file that defines the function using the `%addincludepath` directive.

TLC provides a `%with` directive that facilitates using records. See “Target Language Compiler Directives” on page 19-2.

Note The format and structure of the *model*.rtw file are subject to change from one release of the code generator to another.

A record read in from a file is changeable, like other records that you declare in a program. The record `CompiledModel` is modified many times during code generation. `CompiledModel` is the global record in the *model*.rtw file. It contains variables used for code generation, such as `NumNonvirtSubsystems`, `NumBlocks`. It is also appended during code generation with many new variables, options, and subrecords.

Functions such as `LibGetFormattedBlockPath` are provided in TLC libraries located in *matlabroot*/rtw/c/tlc/lib/*.tlc (open). For a complete list of available functions, refer to TLC Function Library Reference on “Target Language Compiler”.

Assign Values to Fields of Records

To assign a value to a field of a record, you must use a qualified variable expression. A qualified variable expression references a variable in one of the following forms:

- An identifier
- A qualified variable followed by “`.`” followed by an identifier, such as

  ```
  var[2].b
  ```
- A qualified variable followed by a bracketed expression such as

  ```
  var[expr]
  ```

Record Aliases

In TLC, it is possible to create what is called an alias to a record. Aliases are similar to pointers to structures in C. You can create multiple aliases to a single record. Modifications to the aliased record are visible to every place that holds an alias.

The following code fragment illustrates the use of aliases:

```
%createrecord foo { field 1 }
%createrecord a { }
%createrecord b { }
%createrecord c { }

%addtorecord a foo foo
%addtorecord b foo foo
%addtorecord c foo { field 1 }

%% notice we are not changing field through a or b.
%assign foo.field  = 2

ISALIAS(a.foo) = %<ISALIAS(a.foo)>
ISALIAS(b.foo) = %<ISALIAS(b.foo)>
ISALIAS(c.foo) = %<ISALIAS(c.foo)>

a.foo.field = %<a.foo.field>
b.foo.field = %<b.foo.field>
c.foo.field = %<c.foo.field>
%% note that c.foo.field is unchanged
```

Saving this script as `record_alias.tlc` and invoking it with

```
tlc -v record_alias.tlc
```

produces the output

```
ISALIAS(a.foo) = 1
ISALIAS(b.foo) = 1
ISALIAS(c.foo) = 0

a.foo.field = 2
b.foo.field = 2
c.foo.field = 1
```

When inside a function, it is possible to create an alias to a locally created record that is within the function. If the alias is returned from the function, it remains valid even after exiting the function, as in the following example:

```
%function func(value) Output
  %createrecord foo { field value }
  %createrecord a { foo foo }
ISALIAS(a.foo) = %<ISALIAS(a.foo)>
   %return a.foo
 %endfunction

 %assign x = func(2)
 ISALIAS(x) = %<ISALIAS(x)>
 x = %<x>
 x.field = %<x.field>
```

Saving this script as `alias_func.tlc` and invoking it with

```
tlc -v alias_func.tlc
```

produces the output

```
ISALIAS(a.foo) = 1
ISALIAS(x) = 1
x = { field 2 }
x.field = 2
```

As long as there is some reference to a record through an alias, that record is not deleted. This allows records to be used as return values from functions.

See Also

Related Examples

- “Target Language Compiler Directives” on page 19-2
- “Data Handling with TLC” on page 17-15

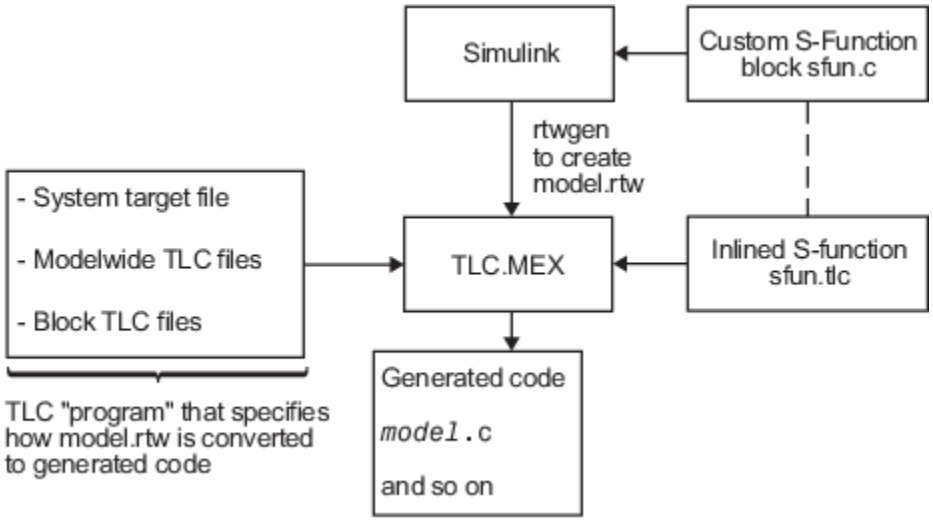
TLC Files

In this section...

"TLC Program" on page 17-11
"Available Target Files" on page 17-11
"Target File Usage" on page 17-12
"System Target Files" on page 17-13
"Compatibility of TLC Files with Generated Code Data Types" on page 17-13

TLC Program

Target Language Compiler (TLC) works with the Simulink software to generate code.



A TLC program is a collection of ASCII files called scripts. Because TLC is an interpreted language, there are no object files. The single target file that calls (with the `%include` directive) other target files used for the program is called the entry point.

Available Target Files

TLC interprets the set of target files to transform the partial representation of the Simulink model (*`model`*`.rtw`) into target-specific code.

Target files provide you with the flexibility to customize the code generated by the compiler. For example, if you use the available system target files, you produce generic C or C++ code from your Simulink model. This executable code is not platform-specific.

Note Do not customize TLC files even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results. Customize only the TLC files that you create.

The parameters in the target files are read from the *model*.rtw file and looked up by using block scoping rules. You can define additional parameters within the target files by using the `%assign` statement.

Use target language directives to write target files. “Target Language Compiler Directives” on page 19-2 provide complete descriptions of the block scope rules and the target language directives.

“model.rtw File and Scopes” on page 18-2 describes `model.rtw` file, which is useful for creating and modifying target files.

In the context of code generation, there are two types of target files:

- System target files

  System target files determine the overall framework of code generation. They determine when blocks are executed, how data is logged, and so on.
- Block target files

  Each block has a target file that determines what code should be generated for the block. The code can vary depending on the exact parameters of the block or the types of connections to it (e.g., wide vs. scalar input). For more information, see “Block Target File Methods” on page 21-21

Model-Wide Target Files and System Target Files

You use model-wide target files on a model-wide basis. Model-wide target files provide basic information to TLC, which transforms the *model*.rtw file into target-specific code.

The system target file is the entry point for TLC. It is analogous to the `main()` routine of a C program. System target files oversee the entire code generation process. For example, the system target file `grt.tlc` sets up some variables for `codegenentry.tlc`, which is the entry point into the system target files. For a complete list of available system target files, see “Compare System Target File Support Across Products”.

Target File Usage

Use the target files to:

- Inline an S-function

  Inlining an S-function means writing a block target file that instructs TLC how to generate code for that S-Function block. The compiler can generate code for noninlined C MEX S-functions. If you inline a C MEX S-function, the compiler can generate more efficient code. Noninlined C MEX S-functions execute by using the S-function application program interface (API) and can be inefficient. You can inline a MATLAB file or Fortran S-function. TLC can generate code for the S-function in both cases.
- Customize the code generated for all models

  You might want to instrument the generated code for profiling or make other changes to overall code generation for all models. To accomplish such changes, modify some of the system target files.

System Target Files

The entire code generation process starts with the single system target file that you specify in the Configuration Parameters dialog box, on the **Code Generation** pane. Click the **Browse** button to activate the system target file browser for this purpose. A close examination of a system target file reveals how code generation occurs. This listing is a listing of the non-comment lines in `grt.tlc`, the target file to generate code for a generic real-time executable.

```
%selectfile NULL_FILE
%assign TargetType = "RT"
%assign Language   = "C"
%assign MatFileLogging = 1
%include "codegenentry.tlc"
```

The three variables, `Language`, `TargetType`, and `MatFileLogging`, are global TLC variables that other functions use. Code generation is then initiated by the call to `codegenentry.tlc`, the main entry point for code generation.

If you want to modify overall code generation, you must change the system target file. After the initial setup, instead of calling `codegenentry.tlc`, you must call your own TLC files. This code shows an example system target file called `mygrt.tlc`.

```
%% Set up variables, etc.
%% Load my library functions
%% Note that mylib.tlc should %include funclib.tlc at the
%% beginning.
%include "mylib.tlc"
%include "commonsetup.tlc"

%% Next, you can include TLC files that you need for
%% preprocessing information about the model and to fill in
%% hooks. The following is an example of including a single
%% TLC file that contains custom hooks.
%include "myhooks.tlc"

%% Finally, call the code generator.
%include "commonentry.tlc"
```

Generated code is placed in a model or subsystem function. The relevant generated function names and their execution order are described in “Execution of Code Generated from a Model” and “Configure Generated C Function Interface for Model Entry-Point Functions”. During code generation, functions from each of the block target files are executed and the generated code is placed in model or subsystem functions.

Compatibility of TLC Files with Generated Code Data Types

When the `DataTypeReplacement` configuration parameter value is `'CDataTypesFixedWidth'`, the generated code does not use Simulink Coder data types, for example, `uint8_T`, `real_T`, and `boolean_T`. Instead, the generated code uses data types from the C99 language standard, for example, `uint8_t`, `double`, and `bool`. If a TLC file contains hard-coded instances of Simulink Coder data types, you can use the `coder.updateTlcForLanguageStandardTypes` function to update the file. The function modifies the specified TLC file by replacing hard-coded instances of Simulink Coder data types with a corresponding API call.

During code generation, the updated TLC code emits:

- C99 language standard names if the `DataTypeReplacement` value is `'CDataTypesFixedWidth'`
- Simulink Coder data type names if the `DataTypeReplacement` value `'CoderTypedefs'`.

For more information, see:

- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “Update TLC File for Compatibility with DataTypeReplacement=CDataTypesFixedWidth Code Generation”

See Also

Related Examples

- “Build Process” on page 17-2
- “Data Handling with TLC” on page 17-15

Data Handling with TLC

In this section...

"Matrix Parameters" on page 17-15
"Code Generator Matrix Parameters" on page 17-15

Matrix Parameters

MATLAB, Simulink, and the code generator use column-major ordering for array storage (1-D, 2-D, ...). The next element of an array in memory is accessed by incrementing the first index of the array. For example, these element pairs are stored sequentially in memory: `A(i)` and `A(i+1)`, `B(i,j)` and `B(i+1,j)`, `C(i,j,k)` and `C(i+1,j,k)`. For more information on the internal representation of MATLAB data, see "MATLAB Data".

Code Generator Matrix Parameters

Simulink and code generator internal data storage formatting differs from MATLAB internal data storage formatting only in the storage of complex number arrays. In MATLAB, the real and imaginary parts are stored in separate arrays. In the Simulink and code generator products they are stored in an "interleaved" format, where the numbers in memory alternate real, imaginary, real, imaginary, and so forth. This convention allows efficient implementations of small signals on Simulink lines and for Mux blocks and other "virtual" signal manipulation blocks (that is, they do not actively copy their inputs, merely the references to them).

The compiled model file, *`model`*`.rtw`, represents matrices as strings in MATLAB syntax, with no implied storage format. This is so you can copy the string out of an `.rtw` file and paste it into MATLAB code and have it recognized by MATLAB.

TLC declares Simulink block matrix parameters as scalar or 1-D array variables

```
real_T scalar;
real_T mat[ nRows * nCols ];
```

where `real_T` can be an arbitrary data type supported by Simulink, and match the variable type given in the model file.

For example, the 3-by-3 matrix in the Look-Up Table (2-D) block

```
1   2   3
4   5   6
7   8   9
```

is stored in *`model`*`.rtw` as

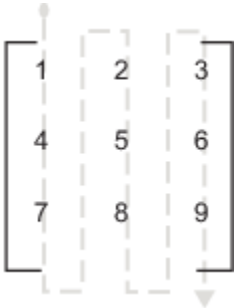
```
Parameter {
      Name                "OutputValues"
      Value                Matrix(3,3)
[[1.0, 2.0, 3.0]; [4.0, 5.0, 6.0]; [7.0, 8.0, 9.0];]
   String                 "t"
   StringType                "Variable"
   ASTNode {
        IsNonTerminal                      0
       Op                      SL_NOT_INLINED
```

```
        ModelParameterIdx                   3
   }
}
```

and results in this definition in *model*.h

```
typedef struct Parameters_tag {
  real_T s1_Look_Up_Table_2_D_Table[9];
                                  /* Variable:s1_Look_Up_Table_2_D_Table
                                    * External Mode Tunable:yes
                                    * Referenced by block:
                                    * <S1>/Look-Up Table (2-D
                          */

  [ ... other parameter definitions ... ]

} Parameters;
```

The *model*.h file declares the actual storage for the matrix parameter and you can see that the format is column-major. That is, read down the columns, then across the rows.




```
Parameters model_P = {
  /* 3 x 3 matrix s1_Look_Up_Table_2_D_Table */
  { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 },
  [ ... other parameter declarations ...]
};
```

TLC accesses matrix parameters via `LibBlockMatrixParameter` and `LibBlockMatrixParameterAddr`, where

`LibBlockMatrixParameter(OutputValues, "", "", 0, "", "", 1)` returns "*model*_P.s1_Look_Up_Table_2_D_Table[*nRows*]" (automatically optimized from "[0+*nRows**1]") and

`LibBlockMatrixParameterAddr(OutputValues, "", "", 0, "", "", 1)` returns "&*model*_P.s1_Look_Up_Table_2_D_Table[*nRows*]" for both inlined and noninlined block TLC code.

Matrix parameters are like other TLC parameters. Only those parameters explicitly accessed by a TLC library function during code generation are placed in the parameters structure. So, following the example, `s1_Look_Up_Table_2_D_Table` is not declared unless `LibBlockParameter` or `LibBlockParameterAddr` explicitly access it.

See Also

Related Examples

- “Code Generation of Matrices and Arrays”

model.rtw File and Authoring S-Functions and Data Objects

- “model.rtw File and Scopes” on page 18-2
- “Data Object Information in model.rtw File” on page 18-5
- “Data References in the model.rtw File” on page 18-9
- “Exception to Using the Library Functions that Access model.rtw” on page 18-11

model.rtw File and Scopes

The code generation software creates a *model*.rtw file from your Simulink model. A *model*.rtw file is a partial representation of a model generated by the build process for use by the Target Language Compiler. It describes blocks, inputs, outputs, parameters, states, storage, and other model components and properties from the corresponding model file.

The generated *model*.rtw file is input to the Target Language Compiler. If you select **Retain .rtw file** from the **Configuration Parameters** > **Code Generation** pane, after building a model, you can view the *model*.rtw file that was generated.

A *model*.rtw file is implemented as an ASCII file of parameter-value pairs stored in a hierarchy of records. A parameter name/parameter value pair is specified as

```
ParameterName  value
```

where ParameterName (also called an identifier) is the name of the TLC identifier and value is a string, scalar, vector, or matrix. For example, in the parameter name/parameter value pair

```
     NumDataOutputPorts 1
```

NumDataOutputPorts is the identifier and 1 is its value.

A record is specified as

```
RecordName {
      .
      .
      .
}
```

A record contains parameter name/parameter value pairs and/or subrecords. For example, this record contains one parameter name/parameter value pair:

```
DataStores {
    NumDataStores        0
}
```

Note The structure of the *model*.rtw file is very likely to change between releases, which is a compelling reason to limit your access to *model*.rtw to the library functions documented under TLC Function Library Reference: “Target Language Compiler”. For additional information, see “Exception to Using the Library Functions that Access model.rtw” on page 18-11.

Scopes in the model.rtw File

Each record creates a new scope. The *model*.rtw file uses curly braces { and } to open and close records (or scopes). Using scopes, you can access values within the *model*.rtw file.

The scope in this example begins with CompiledModel. Use periods (.) to access values within particular scopes. The format of *model*.rtw is

```
CompiledModel {
  Name   "modelname"           -- Example of a parameter-value
  ...                             pair (record field).
```

```
  System {                      -- There is one system for each
                                   nonvirtual subsystem.
    Block {                     -- Block records for each
      Type     "S-Function"       nonvirtual block in the
                                   system.
      Name     "<S3>/S-Function"
      ...
      Parameter {
       Name "P1"
       Value Matrix(1,2) [[1, 2];]
    }
    ...
    Block {
    }
  }
  ...
  System {                      -- The last system is for the
                                   root of your model.
}
```

For example, to access `Name` within `CompiledModel`, you would use

```
CompiledModel.Name
```

Multiple records of the same name form a list where the index of the first record starts at 0. To access the above S-function block record, you would use

```
CompiledModel.System[0].Block[0]
```

To access the name field of this block, you would use

```
CompiledModel.System[0].Block[0].Name
```

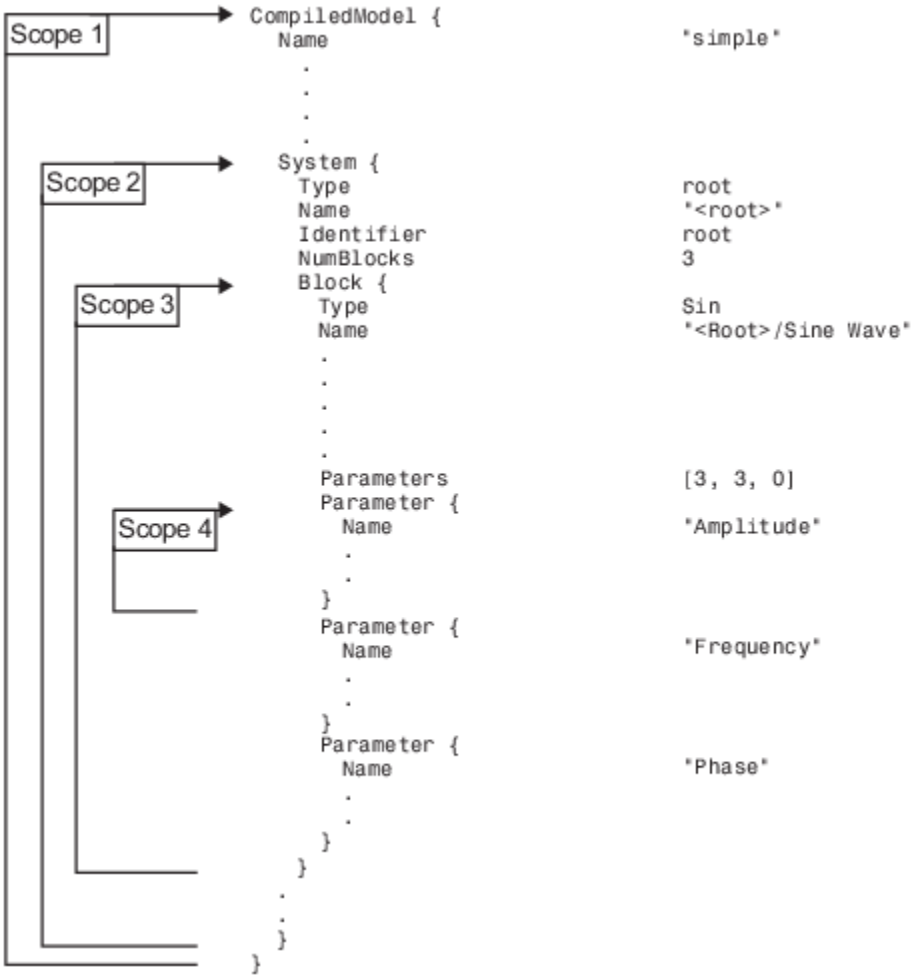
To simplify this process, you can use the `%with` directive, which changes the current scope. For example:

```
%with CompiledModel.System[0].Block[0]
%assign blockName = Name
%endwith
```

`blockName` will have the value `"<S3>/S-Function"`.

When inlining S-function blocks, your S-function block record is scoped as though the above `%with` directive was done. In an inlined `.tlc` file, you should access fields without a fully qualified path.

The following code shows a more detailed scoping example where the `Block` record has several parameter-value pairs (`Type`, `Name`, `Identifier`, and so on), and three subrecords, each called `Parameter`. `Block` is a subrecord of `System`, which is a subrecord of `CompiledModel`. Note that the parameter names in this file changes from release to release.



See Also

Related Examples

- “Data Object Information in model.rtw File” on page 18-5
- “Data References in the model.rtw File” on page 18-9

Data Object Information in model.rtw File

You can obtain meta information about the generated code by accessing the *model*.rtw file in the build folder.

During the build process, the code generator writes information about Simulink signal and parameter data objects to the *model*.rtw file. An Object record with CoderInfo property information is written for each parameter or signal that meets certain conditions. These conditions are described in “Object Records for Parameters” on page 18-5 and “Object Records for Signals” on page 18-6.

The Object records contain the information corresponding to the associated data object. To access Object records, you must write Target Language Compiler code (see “Access Data Object Information by Using Target Language Compiler (TLC) Code” on page 18-7).

For some data, defining storage classes can be helpful. For more information, see “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23. Embedded Coder is required.

Note These Object record examples are generated from the example model rtwdemo_configrpinterface. The parameters and signals are individually configured by using the ExportedGlobal storage class.

Object Records for Parameters

An Object record that has CoderInfo property information is included in the ModelParameters section of the *model*.rtw file for each parameter that meets these conditions:

- The parameter resolves to a Simulink.Parameter data object or to a parameter data object that comes from a class derived from the Simulink.Parameter class.
- The parameter symbol is preserved in the generated code. The symbol is preserved when the CoderInfo.StorageClass property of the data object is not set to Auto or if you set the storage class on the individual parameter by using the Code Mappings editors or the coder.mapping.api.CodeMapping object.

This example shows part of an Object record for a parameter.

```
ModelParameters {
  NumParameters     10
  ...
    Parameter {
      Identifier          "K1"
      LogicalSrc          P7
      WorkspaceVarName          "K1"
      Protected              no
      Tunable              yes
      StorageClass          "ExportedGlobal"
      Value              [2]
      OriginalDataTypeIdx     2
      CGTypeIdx              36
      ContainerCGTypeIdx      42
      IsPublic              1
      ReferencedBy          Matrix(1,4)
[[1, 0, 4, 2];]
      GraphicalRef          Matrix(1,2)
[[0, 10];]
      GraphicalSource          [-1, -1]
```

```
      OwnerSysIdx          [1, -1]
      HasObject              1
      Object {
        Package            Simulink
        Class            Parameter
        ObjectProperties {
          Value              2.0
          CoderInfo {
            Object {
            Package              Simulink
            Class              CoderInfo
            ObjectProperties {
              HasCoderInfo        1
              StorageClass        "ExportedGlobal"
              TypeQualifier        ""
              Alias            ""
              Alignment        -1
              IsCSCPackageOverridden    0
              CSCPackageName        "Simulink"
              ParameterOrSignal    "Parameter"
              CustomStorageClass    "Default"
              CustomAttributes {
                Object {
                  Package            SimulinkCSC
                  Class            AttribClass_Simulink_Default
                  ObjectProperties {
             }
         }
          }
         }
        }
       }
  ...
}
```

Object Records for Signals

An `Object` record that has `CoderInfo` property information is included in either the `ExternalOutputs`, `ExternalInputs`, or `BlockOutputs` section of the *`model`*`.rtw` file for each signal (including root-level Inport and Outport blocks) whose symbol is preserved in the generated code. The symbol is preserved when the signal uses a storage class other than `Auto` and is individually mapped.

An `Object` record that has `CoderInfo` property information is included in the *`model`*`.rtw` file for each signal that preserves the signal symbol in the generated code and meets these conditions:

- The `CoderInfo.StorageClass` property of the data object is not set to `Auto`.
- The data object is individually mapped to a predefined storage class by using the Code Mappings editor or the `coder.mapping.api.CodeMapping` object.

If you configure the signal to be an unstructured global variable in the generated code, its validity and uniqueness are enforced and its symbol is preserved.

This example shows part of an `Object` record for a root-level Outport block.

```
ExternalOutputs {
  ...
  NumExternalOutputs        1
  ...
    ExternalOutput {
      ArgSrc              Y0
      Block              [1,3]
      BlockName              "<Root>/Out1"
      GrSrc              [0, 4]
      Identifier          "output"
```

```
      StorageClass          "ExportedGlobal"
      HasObject              1
      Object {
        Package            Simulink
        Class            Signal
        ObjectProperties {
          CoderInfo {
            Object {
              Package              Simulink
              Class              CoderInfo
              ObjectProperties {
                HasCoderInfo        1
                StorageClass        "ExportedGlobal"
                TypeQualifier        ""
                Alias            "output"
                Alignment        -1
                IsCSCPackageOverridden    0
                CSCPackageName        "Simulink"
                ParameterOrSignal    "Signal"
                CustomStorageClass    "Default"
                CustomAttributes {
                  Object {
                    Package            SimulinkCSC
                    Class            AttribClass_Simulink_Default
                    ObjectProperties {
                    }
                  }
                }
              }
            }
          }
  ...
}
```

Access Data Object Information by Using Target Language Compiler (TLC) Code

Here, sample code shows how to access data object information from the *model*.rtw file by using TLC code.

Access Parameter Object Records

This code fragment iterates over `Parameter` structures in the `ModelParameters` section of the *model*.rtw file. The code extracts information from parameter `Object` records.

```
%with CompiledModel.ModelParameters
  %foreach modelParamIdx = NumParameters
    %assign thisModelParam = Parameter[modelParamIdx]
    %assign paramName = thisModelParam.Identifier
    %if EXISTS("thisModelParam.Object.ObjectProperties")
      %with thisModelParam.Object.ObjectProperties
        %assign valueInObject = Value
        %with CoderInfo.Object.ObjectProperties
          %assign storageClassInObject  = StorageClass
        %endwith
        %% ***********************************
        %% Access user-defined properties here
        %% ***********************************
        %if EXISTS("MY_PROPERTY_NAME")
          %assign userDefinedPropertyName = MY_PROPERTY_NAME
        %endif
        %% ***********************************
      %endwith
    %endif
  %endforeach
%endwith
```

Access Signal Object Records

This code fragment iterates over `ExternalBlockOutput` structures in the `BlockOutputs` section of the *`model`*`.rtw` file. The code extracts information from signal `Object` records.

```
%with CompiledModel.BlockOutputs
  %foreach blockOutputIdx = NumExternalBlockOutputs
    %assign thisBlockOutput = ExternalBlockOutput[blockOutputIdx]
    %assign signalName = thisBlockOutput.Identifier
    %if EXISTS("thisBlockOutput.Object.ObjectProperties")
      %with thisBlockOutput.Object.ObjectProperties
        %with CoderInfo.Object.ObjectProperties
          %assign storageClassInObject  = StorageClass
        %endwith \
        %% ************************************\
        %% Access user-defined properties here\
        %% ************************************
        %if EXISTS("MY_PROPERTY_NAME")
          %assign userDefinedPropertyName = MY_PROPERTY_NAME
        %endif
        %% ************************************
      %endwith
    %endif
  %endforeach
%endwith
```

See Also

Related Examples

- “Data References in the model.rtw File” on page 18-9
- “model.rtw File and Scopes” on page 18-2

Data References in the model.rtw File

Data Reference Overview

Some records in a *model*.rtw file, such as those corresponding to parameters and constant block I/O, can have extremely large data value vectors embedded in them. Such a vector can cause significant memory overhead during code generation because the values must be maintained as text in memory during this process.

To avoid such overhead, by default the Simulink software does not write out the entire data value vector into *model*.rtw. Instead, it writes a key called a data reference that can be used during code generation to access the data directly from Simulink. If the data is not mutated during code generation, it is efficiently streamed to disk when the actual code containing the data values is written out.

A data reference has the format SLData(*index*), where *index* is a numeric value that tells Simulink which data is being referenced. TLC directives such as GENERATE_FORMATTED_VALUE store data references in unexpanded format in memory. When the generated code is written out to disk, the data values expand to the actual values. To access the data vector of the model parameter written to the ModelParameters record in the *model*.rtw, use the LibGetParameterValue function.

Control Data Reference Threshold

By default, if you do not specify a value for the model configuration parameter RTWDataReferencesMinSize, the code generator assigns the value -1 to the parameter. In this case, the code generator writes a data reference of a model parameter to the *model*.rtw file in place of a data vector. This optimization reduces the file read/write operations, which improves the code generation time. When you specify a positive value for the RTWDataReferencesMinSize parameter, the code generator uses the value as the threshold size based on which it writes data references to the *model*.rtw file in place of a data vector.

To change the maximum length of a vector that can appear literally in the file, use:

```
set_param(0, 'RTWDataReferencesMinSize', maxlen)
```

Simulink replaces a vector as long or longer than maxlen with a data reference when it creates *model*.rtw. Specify maxlen as an integer or as inf. Specifying inf disables data references. The complete value set of every vector, however long, then appears literally in *model*.rtw and occupies text memory during code generation.

Setting an explicit *maxlen* affects only the current MATLAB session. To set the value across sessions, include a set_param command in your startup.m file, or automate execution of the command when MATLAB launches.

Expand Data References

You can explicitly expand a data reference by using the GENERATE_FORMATTED_VALUE built-in function with the optional third expand argument. Commands such as FEVAL may cause a data reference to be expanded to the full form.

Avoid Data Reference Expansion

Either turning off data references completely or expanding select parameters in TLC can cause significant text memory overhead during the code generation process. During most common code generation tasks, it is unnecessary to have the expanded data vector in memory and pay the price of the additional overhead. Avoid expanded data vectors unless no alternative exists.

Restart Code Generation

A `model.rtw` file that contains data references cannot be used in isolation to restart a custom code generation process. The data references within it become stale once the code generation process is completed. Attempting to start a code generation process using only this file may result in unpredictable behavior and memory segmentation faults.

See Also

Related Examples

- “Data Object Information in model.rtw File” on page 18-5
- “Exception to Using the Library Functions that Access model.rtw” on page 18-11

Exception to Using the Library Functions that Access model.rtw

There are several library functions that provide access to block inputs, outputs, parameters, sample times, and other information. It is recommended that you use these library functions to access many of the parameter name/parameter values pairs in the block record, as opposed to accessing the parameter name/parameter value pairs directly from your block TLC code. For more information about using these functions (recommended method for accessing *model*.rtw), see “Target Language Compiler Library Functions Overview” on page 22-2.

An exception to using these functions is when you access parameter settings for a block. Parameter settings can be written out using the `mdlRTW` function of a C MEX S-function. They can contain data in the form of strings, scalar values, vectors, and matrices. They can be used to pass fixed values and information that is used to alter the generated code for a block or directly as values in the resulting code of a block.

Example Exception to Using the Library Functions

The following example demonstrates accessing parameter settings for a block using the `mdlRTW` function of a C MEX S-function. For more details on using parameter settings, see “Inlining S-Functions” on page 15-8.

mdlRTW Function in C MEX S-Function Code

```
static void mdlRTW(SimStruct *S)
{
    if (!ssWriteRTWParamSettings( S, 1, SSWRITE_VALUE_QSTR, "Operator", "AND"))
    {
        ssSetErrorStatus(S,"Error writing parameter data to .rtw file");
        return;
    }
}
```

Resulting Block Record in model.rtw File

```
Block {
      Type    "S-Function"
      Name    "<Root>/S-Function"

      ...

      SFcnParamSettings {
        Operator    "AND"
      }
    }
```

TLC Code to Access the Parameter Settings

```
%function Outputs(block, system) Output
  %%
  %% Select Operator
  %switch(SFcnParamSettings.Operator)
    %case "AND"
      %assign LogicOp        = "&"
      %break
    ...
  %endswitch
%endfunction
```

Caution Against Directly Accessing Record Fields

When functions in the block target file are called, they are passed to the block and system records for this instance as arguments. The first argument, `block`, is in scope, which means that variable names inside this instance's block record are accessible by name. For example:

```
%assign fast = SFcnParamSetting.Fast
```

Block target files could generate code for a given block by directly using the fields in the Block record for the block. This process is *not* recommended, for two reasons:

- The contents of the *`model`*`.rtw` file can change from release to release. This can cause block TLC files that access the *`model`*`.rtw` file directly to stop working.
- TLC library functions are provided that substantially reduce the amount of TLC code for implementing a block while handling the various configurations (widths, data types, etc.) a block might have. These library functions are provided by the system target files to provide access to inputs, outputs, parameters, and so on. Using these functions in a block TLC script makes it flexible enough to generate code for multiple instances or configurations of the block, as well as across releases. Exceptions to this do occur, however, such as when you want to directly access a field in the block's record. This happens with parameter settings, as discussed in "TLC Code to Access the Parameter Settings" on page 18-11.

See Also

Related Examples

- "Access Memory in Generated Code Using Global Data Map"
- "Data References in the model.rtw File" on page 18-9

Directives and Built-In Functions

You control how code is generated from models largely through writing or modifying scripts that apply TLC directives and built-in functions. Use the following sections as your primary reference to the syntax and format of target language constructs, as well as the MATLAB `tlc` command itself.

- “Target Language Compiler Directives” on page 19-2
- “Command-Line Arguments” on page 19-49

Target Language Compiler Directives

You control how code is generated from models largely through writing or modifying scripts that apply TLC directives and built-in functions. Use the following sections as your primary reference to the syntax and format of target language constructs, as well as the MATLAB `tlc` command itself.

A target language directive must be the first non-blank character on a line and begins with the `%` character. Lines beginning with `%%` are TLC comments, and are *not* passed to the output stream. Lines beginning with `/*` are C comments, and *are* passed to the output stream.

Syntax

A target language file consists of a series of statements of either form:

- `[text | %<expression>]*`

  The literal text is passed to the output stream unmodified, and expressions enclosed in `%< >` are evaluated before being written to output (stripped of `%< >`).
- `%keyword [argument1, argument2, ...]`

  The `%keyword` represents one of the directives of Target Language Compiler, and `[argument1, argument2, ...]` represents expressions that define required parameters. For example, the statement

  ```
  %assign sysNumber = sysIdx + 1
  ```

  uses the `%assign` directive to define or change the value of the `sysNumber` parameter.

Directives

The rest of this section shows the complete set of Target Language Compiler directives, and describes each directive in detail.

%% text

Single-line comment where `text` is the comment.

/% text%/

Single (or multiline) comment where `text` is the comment.

%matlab

Calls a MATLAB function that does not return a result. For example, `%matlab disp(2.718)`.

%<expr>

Target language expressions that are evaluated. For example, if you have a TLC variable that was created via `%assign varName = "foo"`, then `%<varName>` would expand to `foo`. Expressions can also be function calls, as in `%<FcnName(param1,param2)>`. On directive lines, TLC expressions need not be placed within the `%<>` syntax. Doing so causes a double evaluation. For example, `%if %<x> == 3` is processed by creating a hidden variable for the evaluated value of the variable x. The

`%if` statement then evaluates this hidden variable and compares it against 3. The efficient way to do this operation is to write `%if x == 3`. In MATLAB notation, this would equate to writing if `eval('x') == 3` as opposed to if `x = 3`. The exception to this is during an `%assign` for format control, as in

```
 %assign str = "value is: %<var>"
```

Note: Nested evaluation expressions (e.g., `%<foo(%<expr>)>`) are not supported.

There is not a speed penalty for evaluations inside strings, such as

```
%assign x = "%<expr>"
```

Avoid evaluations outside strings, such as the following example.

```
%assign x = %<expr>
```

%if expr%elseif expr%else%endif

Conditional inclusion, where the constant expression *`expr`* must evaluate to an integer. For example, the following code checks whether a parameter, `k`, has the numeric value 0.0 by executing a TLC library function to check for equality.

```
%if ISEQUAL(k, 0.0)
  <text and directives to be processed if k is 0.0>
%endif
```

In this and other directives, you do not have to expand variables or expressions using the `%<expr>` notation unless `expr` appears within a string. For example,

`%if ISEQUAL(idx, "my_idx%<i>")`, where `idx` and `i` are both strings.

As in other languages, logical evaluations do short-circuit (are halted when the result is known).

%switch expr %case expr %break %default %break %endswitch

The `%switch` directive is similar to the C language `switch` statement. The expression `expr` should be of a type that can be compared for equality using the `== operator`. If the `%break` is not included after a `%case` statement, then it will fall through to the next statement.

%with %endwith

`%with recordName` is a scoping operator. Use it to bring the named record into the current scope, to remain until the matching `%endwith` is encountered (`%with` directives can be nested as desired).

Note that on the left side of `%assign` statements contained within a `%with` / `%endwith` block, references to fields of records must be fully qualified (see “Assign Values to Fields of Records” on page 17-8), as in the following example.

```
%with CompiledModel
     %assign oldName = name
     %assign CompiledModel.name = "newname"
     %endwith
```

%setcommandswitch string

Changes the value of a command-line switch as specified by the argument string. Only the following switches are supported:

```
v, m, p, O, d, r, I, a
```

The following example sets the verbosity level to 1.

```
%setcommandswitch "-v1"
```

See also “Command-Line Arguments” on page 19-49.

%assert expr

Tests a value of a Boolean expression. If the expression evaluates to false, TLC issues an error message, a stack trace and exit; otherwise, the execution continues normally. To enable the evaluation of assertions outside the code generator environment, use the command-line option `-da`. When building from within the code generator, this flag is ignored, as it is superseded by the **Enable TLC assertion** check box on the **TLC process** section of the **Code Generation > Debug** pane. To control assertion handling from the MATLAB Command Window, use

```
set_param(model, 'TLCAssertion', 'on|off')
```

to set this flag on or off. Default is Off. To see the current setting, use

```
get_param(model, 'TLCAssertion')
```

%error %warning %trace %exit

Flow control directives:

`%error` *`tokens`*

The *`tokens`* are expanded and displayed.

`%warning` *`tokens`*

The *`tokens`* are expanded and displayed.

`%trace` *`tokens`*

The *`tokens`* are expanded and displayed only when the verbose output command-line option `-v` or `-v1` is specified.

`%exit` *`tokens`*

The *`tokens`* are expanded, displayed, and TLC exits.

When reporting errors, use the following command if the error is produced by an incorrect configuration that the user needs to fix in the model.

```
%exit Error Message
```

If you are adding assert code (that is, code that should never be reached), use

```
%setcommandswitch "-v1" %% force TLC stack trace
%exit Assert message
```

%assign

Creates identifiers (variables). The general form is

```
%assign [::]variable = expression
```

The `::` specifies that the variable being created is a global variable; otherwise, it is a local variable in the current scope (i.e., a local variable in the function).

If you need to format the variable, say, within a string based upon other TLC variables, then you should perform a double evaluation, as in

```
%assign nameInfo = "The name of this is %<Name>"
```

or alternately

```
%assign nameInfo = "The name of this is " + Name
```

To assign a value to a field of a record, you must use a qualified variable expression. See “Assign Values to Fields of Records” on page 17-8.

%createrecord

Creates records in memory. This command accepts a list of one or more record specifications (e.g., `{ foo 27 }`). Each record specification contains a list of zero or more name-value pairs (e.g., `foo 27`) that become the members of the record being created. The values themselves can be record specifications, as the following illustrates.

```
%createrecord NEW_RECORD { foo 1 ; SUB_RECORD {foo 2} }
%assign x = NEW_RECORD.foo                 /* x = 1 */
%assign y = NEW_RECORD.SUB_RECORD.foo      /* y = 2 */
```

If more than one record specification follows a given record name, the set of record specifications constitutes an array of records.

```
%createrecord RECORD_ARRAY { foo 1 } ...
                                   { foo 2 } ...
                                   { bar 3 }
%assign x = RECORD_ARRAY[1].foo            /* x = 2 */
%assign y = RECORD_ARRAY[2].bar            /* y = 3 */
```

Note that you can create and index arrays of subrecords by specifying `%createrecord` with identically named subrecords, as follows:

```
%createrecord RECORD_ARRAY { SUB_RECORD { foo 1 } ...
                             SUB_RECORD { foo 2 } ...
                             SUB_RECORD { foo 3 } }
%assign x = RECORD_ARRAY.SUB_RECORD[1].foo  /* x = 2 */
%assign y = RECORD_ARRAY.SUB_RECORD[2].foo  /* y = 3 */
```

If the scope resolution operator (`::`) is the first token after the `%createrecord` token, the record is created in the global scope.

Note You should not create a record array by using `%createrecord` within a loop.

%addtorecord

Adds fields to an existing record. The new fields can be name-value pairs or aliases to already existing records.

```
%addtorecord OLD_RECORD foo 1
```

If the new field being added is a record, then `%addtorecord` makes an alias to that record instead of a deep copy. To make a deep copy, use `%copyrecord`.

```
%createrecord NEW_RECORD { foo 1 }
%addtorecord OLD_RECORD NEW_RECORD_ALIAS NEW_RECORD
```

%mergerecord

Adds (or merges) one or more records into another. The first record will contain the results of the merge of the first record plus the contents of the other records specified by the command. The contents of the second (and subsequent) records are deep copied into the first (i.e., they are not references).

```
%mergerecord OLD_RECORD NEW_RECORD
```

If duplicate fields exist in the records being merged, the original record's fields are not overwritten.

%copyrecord

Makes a deep copy of an existing record. It creates a new record in a similar fashion to `%createrecord` except the components of the record are deep copied from the existing record. Aliases are replaced by copies.

```
%copyrecord NEW_RECORD OLD_RECORD
```

%realformat

Specifies how to format real variables. To format in exponential notation with 16 digits of precision, use

```
%realformat "EXPONENTIAL"
```

To format without loss of precision and minimal number of characters, use

```
%realformat "CONCISE"
```

When inlining S-functions, the format is set to `concise`. You can switch to `exponential`, but should switch it back to `concise` when done.

%language

This must appear before the first `GENERATE` or `GENERATE_TYPE` function call. This specifies the name of the language as a string, which is being generated as in `%language "C"`. Generally, this is added to your system target file.

The only valid value is `C` which enables support for `C` and `C++` code generation as specified by the configuration parameter `TargetLang` (see Language for more information).

%implements

Placed within the `.tlc` file for a specific record type, when mapped via `%generatefile`. The syntax is `%implements "Type" "Language"`. When inlining an S-function in C or C++, this should be the first noncomment line in the file, as in

```
%implements "s_function_name" "C"
```

The next noncomment lines will be `%function` directives specifying the functionality of the S-function.

See the `%language` and `GENERATE` function descriptions for further information.

%generatefile

Provides a mapping between a record `Type` and functions contained in a file. Each record can have functions of the same name but different contents mapped to it (i.e., polymorphism). Generally, this is used to map a `Block` record `Type` to the `.tlc` file that implements the functionality of the block, as in

```
%generatefile "Sin" "sin_wave.tlc"
```

%filescope

Limits the scope of variables to the file in which they are defined. A `%filescope` directive anywhere in a file declares that variables in the file are visible only within that file. Note that this limitation also applies to files inserted, via the `%include` directive, into the file containing the `%filescope` directive.

You should not use the `%filescope` directive within functions or `GENERATE` functions.

`%filescope` is useful in conserving memory. Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This frees memory allocated to such variables. By contrast, global variables persist in memory throughout execution of the program.

%include

Use `%include "file.tlc"`to insert the specified target file at the current point.

The `%include` directives behave as if they were in a global context. For example,

```
%addincludepath "./sub1"
%addincludepath "./sub2"
```

in a `.tlc` file enables either subfolder to be referenced implicitly:

```
%include "file_in_sub1.tlc"
%include "file_in_sub2.tlc"
```

Use forward slashes for folder names, as they work on both UNIX and PC systems. However, if you do use back slashes in PC folder names, be sure to escape them, e.g., `"C:\\mytlc"`. Alternatively, you can express a PC folder name as a literal using the L format specifier, as in `L"C:\mytlc"`.

%addincludepath

Use `%addincludepath "folder"` to add additional paths to be searched. Multiple `%addincludepath` directives can appear. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.

Using `%addincludepath` directives establishes a global context. For example,

```
%addincludepath "./sub1"
%addincludepath "./sub2"
```

in a `.tlc` file enables either subfolder to be referenced implicitly:

```
%include "file_in_sub1.tlc"
%include "file_in_sub2.tlc"
```

Use forward slashes for folder names, as they work on both UNIX and PC systems. However, if you do use back slashes in PC folder names, be sure to escape them, e.g., `"C:\\mytlc"`. Alternatively, you can express a PC folder name as a literal using the L format specifier, as in `L"C:\mytlc"`.

%roll %endroll

Multiple inclusion plus intrinsic loop rolling based upon a specified threshold. This directive can be used by most Simulink blocks that have the concept of an overall block width that is usually the width of the signal passing through the block.

This example of the `%roll` directive is for a gain operation, `y=u*k`:

```
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign rollVars = ["U", "Y", "P"]
  %roll sigIdx = RollRegions, lcv = RollThreshold, block,...
        "Roller", rollVars
    %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
    %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
    %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
      %<y> = %<u> * %<k>;
  %endroll
%endfunction
```

The `%roll` directive is similar to `%foreach`, except that it iterates the identifier (`sigIdx` in this example) over roll regions. Roll regions are computed by looking at the input signals and generating regions where the inputs are contiguous. For blocks, the variable `RollRegions` is automatically computed and placed in the `Block` record. An example of a roll regions vector is `[0:19, 20:39]`, where there are two contiguous ranges of signals passing through the block. The first is `0:19` and the second is `20:39`. Each roll region is either placed in a loop body (e.g., the C language `for` statement) or inlined, depending upon whether or not the length of the region is less than the roll threshold.

Each time through the `%roll` loop, `sigIdx` is an integer for the start of the current roll region or an offset relative to the overall block width when the current roll region is less than the roll threshold. The TLC global variable `RollThreshold` is the general model-wide value used to decide when to place a given roll region in a loop. When the decision is made to place a given region in a loop, the loop control variable is a valid identifier (e.g., `"i"`); otherwise it is `""`.

The `block` parameter is the current block that is being rolled. The `"Roller"` parameter specifies the name for internal `GENERATE_TYPE` calls made by `%roll`. The default `%roll` handler is `"Roller"`, which is responsible for setting up the default block loop rolling structures (e.g., a C `for` loop).

The `rollVars` (roll variables) are passed to `"Roller"` functions to create roll structures. The defined loop variables relative to a block are

`"U"`

The inputs to the block. It assumes you use `LibBlockInputSignal(portIdx, "", lcv, sigIdx)` to access each input, where `portIdx` starts at 0 for the first input port.

`"u`*`i`*`"`

Similar to `"U"`, except only for specific input, *`i`*. The `"u"` must be lowercase or it will be interpreted as `"U"` above.

"Y"

The outputs of the block. It assumes you use `LibBlockOutputSignal(portIdx, "", lcv, sigIdx)` to access each output, where `portIdx` starts at 0 for the first output port.

"y*i*"

Similar to "Y", except only for specific output, *i*. The "y" must be lowercase or it will be interpreted as "Y" above.

"P"

The parameters of the block. It assumes you use `LibBlockParameter(name, "", lcv, sigIdx)` to access them.

"<param>/*name*"

Similar to "P", except specific for a specific *name*.

rwork

The RWork vectors of the block. It assumes you use `LibBlockRWork(name, "", lcv, sigIdx)` to access them.

"<rwork>/*name*"

Similar to RWork, except for a specific *name*.

dwork

The DWork vectors of the block. It assumes you use `LibBlockDWork(name, "", lcv, sigIdx)` to access them.

"<dwork>/*name*"

Similar to DWork, except for a specific *name*.

iwork

The IWork vectors of the block. It assumes you use `LibBlockIWork(name, "", lcv, sigIdx)` to access them.

"<iwork>/*name*"

Similar to IWork, except for a specific *name*.

pwork

The PWork vectors of the block. It assumes you use `LibBlockPWork(name, "", lcv, sigIdx)` to access them.

"<pwork>/*name*"

Similar to PWork, except for a specific *name*.

"Mode"

The mode vector. It assumes you use `LibBlockMode("",lcv,sigIdx)` to access it.

"PZC"

Previous zero-crossing state. It assumes you use `LibPrevZCState("",lcv, sigIdx)` to access it.

To roll your own vector based upon the block's roll regions, you need to walk a pointer to your vector. Assuming your vector is pointed to by the first PWork, called name,

```
datatype *buf = (datatype*)%<LibBlockPWork(name,"","",0)
%roll sigIdx = RollRegions, lcv = RollThreshold, block, ...
      "Roller", rollVars
  *buf++ = whatever;
%endroll
```

Note: In the above example, `sigIdx` and `lcv` are local to the body of the loop.

%breakpoint

Sets a breakpoint for the TLC debugger. See "%breakpoint Directive" on page 20-5.

%function %return %endfunction

A function that returns a value is defined as

```
%function name(optional-arguments)
    %return value
  %endfunction
```

A void function does not produce output and is not required to return a value. It is defined as

```
%function name(optional-arguments) void
  %endfunction
```

A function that produces outputs to the current stream and is not required to return a value is defined as

```
%function name(optional-arguments) Output
  %endfunction
```

For block target files, you can add to your inlined `.tlc` file the following functions that are called by the model-wide target files during code generation.

`%function` **`BlockInstanceSetup`**`(block,system) void`

Called for each instance of the block within the model.

`%function` **`BlockTypeSetup`**`(block,system) void`

Called once for each block type that exists in the model.

`%function` **`Enable`**`(block,system) Output`

Use this if the block is placed within an enabled subsystem and has to take specific actions when the subsystem is enabled. Place within a subsystem enable routine.

`%function` **`Disable`**`(block,system) Output`

Use this if the block is placed within a disabled subsystem and has to take specific actions when the subsystem is disabled. Place within a subsystem disable routine.

`%function` **`Start`**`(block,system) Output`

Include this function if your block has startup initialization code that needs to be placed within `MdlStart`.

`%function` **`InitializeConditions`**`(block,system) Output`

Use this function if your block has state that needs to be initialized at the start of execution and when an enabled subsystem resets states. Place in `MdlStart` and/or subsystem initialization routines.

`%function` **`Outputs`**`(block,system) Output`

The primary function of your block. Place in `MdlOutputs`.

`%function` **`Update`**`(block,system) Output`

Use this function if your block has actions to be performed once per simulation loop, such as updating discrete states. Place in `MdlUpdate`.

`%function` **`Derivatives`**`(block,system) Output`

Use this function if your block has derivatives.

`%function` **`ZeroCrossings`**`(block,system) Output`

Use this function if your block does zero-crossing detection and has actions to be performed in `MdlZeroCrossings`.

`%function` **`Terminate`**`(block,system) Output`

Use this function if your block has actions that need to be in `MdlTerminate`.

%foreach %endforeach

Multiple inclusion that iterates from `0` to the `upperLimit-1` constant integer expression. Each time through the loop, the `loopIdentifier`, (e.g., `x`) is assigned the current iteration value.

```
%foreach loopIdentifier = upperLimit
   %break -- use this to exit the loop
   %continue -- use this to skip the following code and
                continue to the next iteration
%endforeach
```

Note: The `upperLimit` expression is cast to a TLC integer value. The `loopIdentifier` is local to the loop body.

%for

Multiple inclusion directive with syntax

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the `%for` directive is identical to the `%foreach` statement. The `%break` and `%continue` directives act the same as they do in the `%foreach` directive. `const-exp2` is a Boolean expression that indicates whether the loop should be rolled (see `%roll` above).

If `const-exp2` evaluates to `TRUE`, `ident2` is assigned the value of `const-exp3`. Otherwise, `ident2` is assigned an empty string.

Note: `ident1` and `ident2` above are local to the loop body.

%openfile %selectfile %closefile

These are used to manage the files that are created. The syntax is

```
%openfile streamId="filename.ext" mode  {open for writing}
%selectfile streamId                    {select an open file}
%closefile streamId                     {close an open file}
```

Note that the "`filename.ext`" is optional. If a filename is not specified, a variable (string buffer) named `streamId` is created containing the output. The mode argument is optional. If specified, it can be "`a`" for appending or "`w`" for writing.

Note that the special string `streamIdNULL_FILE` specifies no output. The special string `streamIdSTDOUT` specifies output to the terminal.

To create a buffer of text, use

```
%openfile buffer
text to be placed in the 'buffer' variable.
%closefile buffer
```

Now `buffer` contains the expanded text specified between the `%openfile` and `%closefile` directives.

%generate

`%generate blk fn` is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`)`.

`%generate blk fn type` is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`,`*`type`*`)`.

See “GENERATE and GENERATE_TYPE Functions” on page 19-24.

%undef

`%undef` *`var`* removes the variable *`var`* from scope. If *`var`* is a field in a record, `%undef` removes that field from the record. If *`var`* is a record array, `%undef` removes the entire record array.

Comments

You can place comments anywhere within a target file. To include comments, use the `/%...%/` or `%%` directives. For example:

```
/%
    Abstract:    Return the field with [width], if field is wide
%/
```

or

```
%endfunction %% Outputs function
```

Use the `%%` construct for line-based comments. Characters from `%%` to the end of the line become a comment.

Non-directive lines, that is, lines that do not have `%` as their first non-blank character, are copied into the output buffer verbatim. For example,

```
/* Initialize sysNumber */
int sysNumber = 3;
```

copies both lines to the output buffer.

To include comments on lines that do not begin with the `%` character, you can use the `/%...%/` or `%%` comment directives. In these cases, the comments are not copied to the output buffer.

Note If a non-directive line appears within a function, it is not copied to the output buffer unless the function is an output function or you specifically select an output file using the `%selectfile` directive. For more information about functions, see “Target Language Functions” on page 19-46.

Line Continuation

You can use the C language \ character or the MATLAB sequence `...` to continue a line. If a directive is too long to fit conveniently on one line, this allows you to split the directive onto multiple lines. For example:

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block,\
      "Roller", rollVars
```

or

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block,...
      "Roller", rollVars
```

Note Use \ to suppress line feeds to the output and the ellipsis `...` to indicate line continuation. Note that \ and the ellipsis `...` cannot be used inside strings.

Target Language Value Types

This table shows the types of values you can use within the context of expressions in your target language files. Expressions in the Target Language Compiler must use these types.

Value Type String	Example	Description
`"Boolean"`	`1==1`	Result of a comparison or other Boolean operator. The result will be `TLC_TRUE` or `TLC_FALSE`.
`"Complex"`	`3.0+5.0i`	64-bit double-precision complex number (`double` on the target machine).
`"Complex32"`	`3.0F+5.0Fi`	32-bit single-precision complex number (`float` on the target machine).
`"File"`	`%openfile x`	String buffer opened with `%openfile`.
`"File"`	`%openfile x = "out.c"`	File opened with `%openfile`.
`"Function"`	`%function foo...`	User-defined function and `TLC_FALSE` otherwise.
`"Gaussian"`	`3+5i`	32-bit integer imaginary number (`int` on the target machine).
`"Identifier"`	`abc`	Identifier values can appear only within the *`model`*`.rtw` file and cannot appear in expressions (within the context of an expression, identifiers are interpreted as values). To compare against an identifier value, use a string; the identifier will be converted to a string.
`"Matrix"`	`Matrix (3,2) [ [ 1, 2]; [3 , 4]; [ 5, 6] ]`	Matrices are simply lists of vectors. The individual elements of the matrix do not need to be the same type, and can be any supported type except vectors or matrices. The `Matrix (3,2)` text in the example is optional.
`"Number"`	`15`	Integer number (`int` on the target machine).

Value Type String	Example	Description
"Range"	[1:5]	Range of integers between 1 and 5, inclusive.
"Real"	3.14159	Floating-point number (double on the target machine), including exponential notation.
"Real32"	3.14159F	32-bit single-precision floating-point number (float on the target machine).
"Scope"	Block { ... }	Block record.
"Special"	FILE_EXISTS	Special built-in function, such as FILE_EXISTS.
"String"	"Hello, World"	ASCII character strings. In all contexts, two strings in a row are concatenated to form the final value, as in "Hello, " "World", which is combined to form "Hello, World". These strings include the ANSI C standard escape sequences such as \n, \r, \t, etc. Use of line continuation characters (i.e., \ and ...) inside strings is illegal.
"Subsystem"	<sub1>	Subsystem identifier. Within the context of an expansion, be careful to escape the delimiters on a subsystem identifier, as in %<x == <sub\>>.
"Unsigned"	15U	32-bit unsigned integer (unsigned int on the target machine).
"Unsigned Gaussian"	3U+5Ui	32-bit complex unsigned integer (unsigned int on the target machine).
"Vector"	[1, 2] or BR Vector(2) [1, 2]	Vectors are lists of values. The individual elements of a vector do not need to be the same type, and can be of any supported type except vectors or matrices.

Target Language Expressions

You can include an expression of the form %<*expression*> in a target file. The Target Language Compiler replaces %<*expression*> with a calculated replacement value based upon the type of the variables within the %<> operator. Integer constant expressions are folded and replaced with the resultant value; string constants are concatenated (e.g., two strings in a row "a" "b", are replaced with "ab").

```
%<expression>                /* Evaluates the expression.
  * Operators include most standard C
  * operations on scalars. Array indexing
  * is required for certain parameters that
  * are block-scoped within the .rtw file.*/
```

Within the context of an expression, each identifier must evaluate to an identifier or function argument currently in scope. You can use the %< > directive on a line to perform text substitution. To include the > character within a replacement, you must escape it with a \ character. For example:

```
%<x \> 1 ? "ABC" : "123">
```

Operators that need the > character to be escaped are the following:

Operator	Description	Example
>	greater than	`y = %<x \> 2>;`
>=	greater than or equal to	`y = %<x \>= 3>;`
>>	right shift	`y = %<x \>\> 4>;`

The table Target Language Expressions lists the operators that are allowed in expressions. In this table, expressions are listed in order from highest to lowest precedence. The horizontal lines distinguish the order of operations.

As in C expressions, conditional operators are short-circuited. If the expression includes a function call with effects, the effects are noticed as if the entire expression was not fully evaluated. For example:

```
%if EXISTS(foo) && foo == 3
```

If the first term of the expression evaluates to a Boolean false (i.e., `foo` does not exist), the second term (`foo == 3`) is not evaluated.

In the upcoming table, note that *`numeric`* is one of the following:

- `Boolean`
- `Number`
- `Unsigned`
- `Real`
- `Real32`
- `Complex`
- `Complex32`
- `Gaussian`
- `UnsignedGaussian`

Also, note that *`integral`* is one of the following:

- `Number`
- `Unsigned`
- `Boolean`

See “TLC Data Promotions” on page 19-18 for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

Target Language Expressions

Expression	Definition
`constant`	Constant parameter value. The value can be a vector or matrix.
`variable-name`	Valid in-scope variable name, including the local function scope, if any, and the global scope.
`::variable-name`	Used within a function to indicate that the function scope is ignored when the variable is looked up. This accesses the global scope.
`expr[expr]`	Index into an array parameter. Array indices range from `0` to `N-1`. This syntax is used to index into vectors, matrices, and repeated scope variables.
`expr([expr[,expr]...])`	Function call or macro expansion. The expression outside the parentheses is the function/macro name; the expressions inside are the arguments to the function or macro. **Note:** Macros are text-based; they cannot be used within the same expression as other operators.
`expr . expr`	The first expression must have a valid scope; the second expression is a parameter name within that scope.
`(expr)`	Use `()` to override the precedence of operations.
`!expr`	Logical negation (generates `TLC_TRUE` or `TLC_FALSE`). The argument must be numeric or Boolean.
`-expr`	Unary minus negates the expression. The argument must be numeric.
`+expr`	No effect; the operand must be numeric.
`~expr`	Bit-wise negation of the operand. The argument must be an integer.
`expr * expr`	Multiplies the two expressions; the operands must be numeric.
`expr / expr`	Divides the two expressions; the operands must be numeric.
`expr % expr`	Takes the integer modulo of the expressions; the operands must be integers.

Expression	Definition
`expr + expr`	Works on numeric types, strings, vectors, matrices, and records as follows: **Numeric types:** Add the two expressions; the operands must be numeric. **Strings:** The strings are concatenated. **Vectors:** If the first argument is a vector and the second is a scalar, the scalar is appended to the vector. **Matrices:** If the first argument is a matrix and the second is a vector of the same column width as the matrix, the vector is appended as another row in the matrix. **Records:** If the first argument is a record, the second argument is added as a parameter identifier (with its current value). Note that the addition operator is associative.
`expr - expr`	Subtracts the two expressions; the operands must be numeric.
`expr << expr`	Left-shifts the left operand by an amount equal to the right operand; the arguments must be integers.
`expr >> expr`	Right-shifts the left operand by an amount equal to the right operand; the arguments must be integers.
`expr > expr`	Tests whether the first expression is greater than the second expression; the arguments must be numeric.
`expr < expr`	Tests whether the first expression is less than the second expression; the arguments must be numeric.
`expr >= expr`	Tests whether the first expression is greater than or equal to the second expression; the arguments must be numeric.
`expr <= expr`	Tests whether the first expression is less than or equal to the second expression; the arguments must be numeric.
`expr == expr`	Tests whether the two expressions are equal.
`expr != expr`	Tests whether the two expressions are not equal.
`expr & expr`	Performs the bit-wise AND of the two arguments; the arguments must be integers.
`expr ^ expr`	Performs the bit-wise XOR of the two arguments; the arguments must be integers.
`expr | expr`	Performs the bit-wise OR of the two arguments; the arguments must be integers.
`expr && expr`	Performs the logical AND of the two arguments and returns `TLC_TRUE` or `TLC_FALSE`. This can be used on either numeric or Boolean arguments.

Expression	Definition
`expr || expr`	Performs the logical OR of the two arguments and returns `TLC_TRUE` or `TLC_FALSE`. This can be used on either numeric or Boolean arguments.
`expr ? expr : expr`	Tests the first expression for `TLC_TRUE`. If true, the first expression is returned; otherwise, the second expression is returned.
`expr , expr`	Returns the value of the second expression.

Note Relational operators (`<, =<, >, >=, !=, ==`) can be used with nonfinite values.

You do not have to place expressions in the `%< >eval` format when they appear on directive lines. Doing so causes a double evaluation.

TLC Data Promotions

When the Target Language Compiler operates on mixed types of expressions, it promotes the results to the common types indicated in the following table.

The table uses the following abbreviations:

`B`	Boolean
`N`	Number
`U`	Unsigned
`F`	Real32
`D`	Real
`G`	Gaussian
`UG`	UnsignedGaussian
`C32`	Complex32
`C`	Complex

The top row (in bold) and first column (in bold) show the types of expression used in the operation. The intersection of the row and column shows the resulting type of expressions.

For example, if the operation involves a Boolean expression (`B`) and an unsigned expression (`U`), the result will be an unsigned expression (`U`).

Data Types Resulting from Expressions of Mixed Type

	B	**N**	**U**	**F**	**D**	**G**	**UG**	**C32**	**C**
B	B	N	U	F	D	G	UG	C32	C
N	N	N	U	F	D	G	UG	C32	C
U	U	U	U	F	D	UG	UG	C32	C
F	F	F	F	F	D	C32	C32	C32	C
D	D	D	D	D	D	C	C	C	C
G	G	G	UG	C32	C	G	UG	C32	C
UG	UG	UG	UG	C32	C	UG	UG	C32	C
C32	C32	C32	C32	C32	C	C32	C32	C32	C
C	C	C	C	C	C	C	C	C	C

Formatting

By default, the Target Language Compiler outputs floating-point numbers in exponential notation with 16 digits of precision. To override the default, use the directive

```
%realformat string
```

If *`string`* is `"EXPONENTIAL"`, the standard exponential notation with 16 digits of precision is used. If *`string`* is `"CONCISE"`, the compiler uses internal heuristics to output the values in a more readable form while maintaining accuracy. The `%realformat` directive sets the default format for `Real` number output to the selected style for the remainder of processing or until it encounters another `%realformat` directive.

Conditional Inclusion

The conditional inclusion directives are

```
%if constant-expression
%else
%elseif constant-expression
%endif

%switch constant-expression
%case constant-expression
%break
%default
%endswitch
```

%if

The `constant-expression` must evaluate to an integer expression. It controls the inclusion of the following lines until it encounters an `%else`, `%elseif`, or `%endif` directive. If the `constant-expression` evaluates to `0`, the lines following the directive are not included. If the `constant-expression` evaluates to an integer value other than `0`, the lines following the `%if` directive are included until the `%endif`, `%elseif`, or `%else` directive.

When the compiler encounters an `%elseif` directive, and no prior `%if` or `%elseif` directive has evaluated to nonzero, the compiler evaluates the expression. If the value is 0, the lines following the

`%elseif` directive are not included. If the value is nonzero, the lines following the `%elseif` directive are included until the subsequent `%else`, `%elseif`, or `%endif` directive.

The `%else` directive begins the inclusion of source text if the previous `%elseif` statements or the original `%if` statement evaluates to `0`; otherwise, it prevents the inclusion of subsequent lines up to and including the following `%endif`.

The `constant-expression` can contain any expression specified in "Target Language Expressions" on page 19-14.

%switch

The `%switch` statement evaluates the constant expression and compares it to expressions appearing on `%case` selectors. If a match is found, the body of the `%case` is included; otherwise the `%default` is included.

`%case ... %default` bodies flow together, as in C, and `%break` must be used to exit the switch statement. `%break` exits the nearest enclosing `%switch`, `%foreach`, or `%for` loop in which it appears. For example,

```
%switch(type)
%case x
    /* Matches variable x. */
    /* Note: Any valid TLC type is allowed. */
%case "Sin"
    /* Matches Sin or falls through from case x. */
   %break
    /* Exits the switch. */
%case "gain"
    /* Matches gain. */
   %break
%default
    /* Does not match x, "Sin," or "gain." */
%endswitch
```

In general, this is a more readable form for the `%if/%elseif/%else` construction.

Multiple Inclusion

%foreach

The syntax of the `%foreach` multiple inclusion directive is

```
%foreach identifier = constant-expression
   %break
   %continue
%endforeach
```

The `constant-expression` must evaluate to an integer expression, which then determines the number of times to execute the `foreach` loop. The `identifier` increments from `0` to one less than the specified number. Within the `foreach` loop, you can use `x`, where `x` is the identifier, to access the identifier variable. `%break` and `%continue` are optional directives that you can include in the `%foreach` directive:

- Use `%break` to exit the nearest enclosing `%for`, `%foreach`, or `%switch` statement.

- Use `%continue` to begin the next iteration of a loop.

%for

Note The `%for` directive is functional, but it is not recommended. Instead, use `%roll`, which provides the same capability in a more open way. The code generator does not use the `%for` construct.

The syntax of the `%for` multiple inclusion directive is

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the `%for` directive is identical to the `%foreach` statement in that it causes a loop to execute from `0` to `N-1` times over the body of the loop. In the normal case, it includes only the lines between `%body` and `%endbody`, and the lines between the `%for` and `%body`, and ignores the lines between `%endbody` and `%endfor`.

The `%break` and `%continue` directives act the same as they do in the `%foreach` directive.

`const-exp2` is a Boolean expression that indicates whether the loop should be rolled. If `const-exp2` is true, `ident2` receives the value of `const-exp3`; otherwise it receives the null string. When the loop is rolled, the lines between the `%for` and the `%endfor` are included in the output exactly one time. `ident2` specifies the identifier to be used for testing whether the loop was rolled within the body. For example,

```
%for Index = <NumNonVirtualSubsystems>3, rollvar="i"

    {
        int i;

        for (i=0; i< %<NumNonVirtualSubsystems>; i++)
        {
          %body
    x[%<rollvar>] = system_name[%<rollvar>];
          %endbody

        }
    }
%endfor
```

If the number of nonvirtual subsystems (`NumNonVirtualSubsystems`) is greater than or equal to 3, the loop is rolled, causing the code within the loop to be generated exactly once. In this case, `Index = 0`.

If the loop is not rolled, the text before and after the body of the loop is ignored and the body is generated `NumNonVirtualSubsystems` times.

This mechanism gives each individual loop control over whether or not it should be rolled.

%roll

The syntax of the `%roll` multiple inclusion directive is

```
%roll ident1 = roll-vector-exp, ident2 = threshold-exp, ...
               block-exp [, type-string [,exp-list] ]
  %break
  %continue
%endroll
```

This statement uses the `roll-vector-exp` to expand the body of the `%roll` statement multiple times as in the `%foreach` statement. If a range is provided in the `roll-vector-exp`and that range is larger than the `threshold-exp` expression, the loop will roll. When a loop rolls, the body of the loop is expanded once and the identifier (`ident2`) provided for the threshold expression is set to the name of the loop control variable. If no range is larger than the specified rolling threshold, this statement is identical to the `%foreach` statement. For example,

```
%roll Idx = [ 1 2 3:5, 6, 7:10 ], lcv = 10, ablock
%endroll
```

In this case, the body of the `%roll` statement expands 10 times, as in the `%foreach` statement, because there are no regions greater than or equal to 10. `Idx` counts from 1 to 10, and `lcv` is set to the null string, `""`.

When the Target Language Compiler determines that a given block will roll, it performs a `GENERATE_TYPE` function call to output the various pieces of the loop (other than the body). The default type used is `Roller`; you can override this type with a string that you specify. Extra arguments passed to the `%roll` statement are provided as arguments to these special-purpose functions. The called function is one of these four functions:

RollHeader(block, ...)

This function is called once on the first section of this roll vector that will actually roll. It should return a string that is assigned to the `lcv` within the body of the `%roll` statement.

LoopHeader(block, StartIdx, Niterations, Nrolled, ...)

This function is called once for each section that will roll prior to the body of the `%roll` statement.

LoopTrailer(block, Startidx, Niterations, Nrolled, ...)

This function is called once for each section that will roll after the body of the `%roll` statement.

RollTrailer(block, ...)

This function is called once at the end of the `%roll` statement if any of the ranges caused loop rolling.

These functions should output language-specific declarations, loop code, and so on as required to generate code for the loop.

An example of a `Roller.tlc` file is

```
%implements Roller "C"
%function RollHeader(block) Output
    {
        int i;
```

```
    %return ("i")
%endfunction
%function LoopHeader(block,StartIdx,Niterations,Nrolled) Output
     for (i = %<StartIdx>; i < %<Niterations+StartIdx>; i++)
    {
%endfunction
%function LoopTrailer(block,StartIdx,Niterations,Nrolled) Output
    }
%endfunction
%function RollTrailer(block) Output
    }
%endfunction
```

Note The Target Language Compiler function library provided with the code generator has the capability to extract references to the block I/O and other code generator vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, `LibDeclareRollVars`, `LibBlockMatrixParameter`, `LibBlockParameterAddr`, `LibBlockContinuousState`, and `LibBlockDiscreteState`. (See the function reference pages in “Input Signal Functions” on page 22-7, “Output Signal Functions” on page 22-17, “Parameter Functions” on page 22-23, and “Block State and Work Vector Functions” on page 22-29.) This library also includes a default implementation of `Roller.tlc` as a “flat” roller.

Extending the former example to a loop that rolls,

```
%language "C"
%assign ablock = BLOCK { Name "Hi" }
%roll Idx = [ 1:20, 21, 22, 23:25, 26:46], lcv = 10, ablock
    Block[%< lcv == "" ? Idx : lcv>] *= 3.0;
%endroll
```

This Target Language Compiler code produces this output:

```
{
    int             i;
    for (i = 1; i < 21; i++)
    {
        Block[i] *= 3.0;
    }
    Block[21] *= 3.0;
    Block[22] *= 3.0;
    Block[23] *= 3.0;
    Block[24] *= 3.0;
    Block[25] *= 3.0;
    for (i = 26; i < 47; i++)
    {
        Block[i] *= 3.0;
    }
}
```

Object-Oriented Facility for Generating Target Code

The Target Language Compiler provides a simple object-oriented facility. The language directives are

```
%language string
%generatefile
%implements
```

This facility was designed specifically for customizing the code for Simulink blocks, but can be used for other purposes as well.

%language

The `%language` directive specifies the target language being generated. It is required as a consistency check to verify the implementation files found for the language being generated. The `%language` directive must appear prior to the first `GENERATE` or `GENERATE_TYPE` built-in function call. `%language` specifies the language as a string. For example:

```
%language "C"
```

Simulink blocks have a `Type` parameter. This parameter is a string that specifies the type of the block, for example `"Sin"` or `"Gain"`. The object-oriented facility uses this type to search the path for a file that implements the block. By default the name of the file is the `Type` of the block with `.tlc` appended, so for example, if the `Type` is `"Sin"` the Compiler would search for `"Sin.tlc"` along the path. You can override this default filename using the `%generatefile` directive to specify the filename that you want to use to replace the default filename. For example,

```
%generatefile "Sin" "sin_wave.tlc"
```

The files that implement the block-specific code must contain an `%implements` directive indicating both the type and the language being implemented. The Target Language Compiler will produce an error if the `%implements` directive does not match as expected. For example,

```
%implements "Sin" "Pascal"
```

causes an error if the initial language choice was C.

You can use a single file to implement more than one target language by specifying the desired languages in a vector. For example,

```
%implements "Sin" "C"
```

Finally, you can implement several types using the wildcard (*) for the type field:

```
%implements * "C"
```

Note The use of the wildcard (*) is not recommended because it relaxes error checking for the `%implements` directive.

GENERATE and GENERATE_TYPE Functions

The Target Language Compiler has two built-in functions that dispatch object-oriented calls, `GENERATE` and `GENERATE_TYPE`. You can call a function appearing in an implementation file (from outside the specified file) only by using the `GENERATE` and `GENERATE_TYPE` special functions.

GENERATE

The `GENERATE` function takes two or more input arguments. The first argument must be a valid scope and the second a string containing the name of the function to call. The `GENERATE` function passes

the first block argument and any additional arguments specified to the function being called. The return argument is the value, if any, returned from the function being called. Note that the compiler automatically "scopes" or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line. (See "Variable Scoping" on page 19-39.) This scope is removed when the function returns.

GENERATE_TYPE

The `GENERATE_TYPE` function takes three or more input arguments. It handles the first two arguments identically to the `GENERATE` function call. The third argument is the type; the type specified in the Simulink block is ignored. This facility is used to handle S-function code generation by the build process. That is, the block type is `S-function`, but the Target Language Compiler generates it as the specific S-function specified by `GENERATE_TYPE`. For example,

```
GENERATE_TYPE(block, "Output", "dp_read")
```

specifies that S-function `block` is of type `dp_read`.

The `block` argument and any additional arguments are passed to the function being called. Like the `GENERATE` built-in function, the compiler automatically scopes the first argument before the `GENERATE_TYPE` function is entered and then removes the scope on return.

Within the file containing `%implements`, function calls are looked up first within the file and then in the global scope. This makes it possible to have hidden helper functions used exclusively by the current object.

Note It is not an error for the `GENERATE` and `GENERATE_TYPE` directives to find no matching functions. This is to prevent requiring empty specifications for all aspects of block code generation. Use the `GENERATE_FUNCTION_EXISTS` or `GENERATE_TYPE_FUNCTION_EXISTS` directives to determine whether the specified function actually exists.

Output File Control

The structure of the output file control construct is

```
%openfile string optional-equal-string optional-mode
%closefile id
%selectfile id
```

%openfile

The `%openfile` directive opens a file or buffer for writing; the required string variable becomes a variable of type `file`. For example,

```
%openfile x               /% Opens and selects x for writing. %/
%openfile out = "out.h"   /% Opens "out.h" for writing. %/
```

%selectfile

The `%selectfile` directive selects the file specified by the variable as the current output stream. Output goes to that file until another file is selected using `%selectfile`. For example,

```
%selectfile x               /% Select file x for output. %/
```

%closefile

The `%closefile` directive closes the specified file or buffer. If the closed entity is the currently selected output stream, `%closefile` invokes `%selectfile` to reselect the previously selected output stream.

There are two possible cases that `%closefile` must handle:

- If the stream is a file, the associated variable is removed as if by `%undef`.
- If the stream is a buffer, the associated variable receives the text that has been output to the stream. For example,

```
%assign x = ""                 /% Creates an empty string. %/
%openfile x
"hello, world"
%closefile x         /% x = "hello, world\n"%/
```

If desired, you can append to an output file or string by using the optional mode, `a`, as in

```
%openfile "foo.c", "a"             %% Opens foo.c for appending.
```

Input File Control

The input file control directives are

```
%include string
%addincludepath string
```

%include

The `%include` directive searches the path for the target file specified by `string` and includes the contents of the file inline at the point where the `%include` statement appears.

%addincludepath

The `%addincludepath` directive adds an additional include path to be searched when the Target Language Compiler references `%include` or block target files. The syntax is

```
%addincludepath string
```

The `string` can be an absolute path or an explicit relative path. For example, to specify an absolute path, use

```
%addincludepath "C:\\folder1\\folder2"     (PC)
%addincludepath "/folder1/folder2"         (UNIX)
```

To specify a relative path, the path must explicitly start with `..` For example,

```
%addincludepath ".\\folder2"                   (PC)
%addincludepath "./folder2"                    (UNIX)
```

Note that for PC, the backslashes must be escaped (doubled).

When an explicit relative path is specified, the folder that is added to the Target Language Compiler search path is created by concatenating the location of the target file that contains the `%addincludepath` directive and the explicit relative path.

The Target Language Compiler searches the folders in the following order for target or include files:

1. The current folder.
2. Include paths specified in `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.
3. Include paths specified at the command line via `-I` . The compiler evaluates multiple `-I` options from *right to left*.

Typically, `%addincludepath` directives should be specified in your system target file. Multiple `%addincludepath` directives will add multiple paths to the Target Language Compiler search path.

Note The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

Asserts, Errors, Warnings, and Debug Messages

The related assert, error, warning, and debug message directives are

```
%assert expression
%error tokens
%warning tokens
%trace tokens
%exit tokens
```

These directives produce error, warning, or trace messages whenever a target file detects an error condition, or tracing is desired. The tokens following the directive on a line become part of the generated error or warning message.

The Target Language Compiler places messages generated by `%trace` onto `stderr` if and only if you specify the verbose mode switch (`-v`) to the Target Language Compiler. See “Command-Line Arguments” on page 19-49 for additional information about switches.

The `%assert` directive evaluates the expression and produces a stack trace if the expression evaluates to a Boolean `false`.

Note In order for `%assert` directives to be evaluated, **Enable TLC assertion** must be selected in the **TLC process** section of the **Code Generation** > **Debug** pane. The default action is for `%assert` directives not to be evaluated.

The `%exit` directive reports an error and stops further compilation.

Built-In Functions and Values

The following table lists the built-in functions and values that are added to the list of parameters that appear in the *`model`*`.rtw` file. These Target Language Compiler functions and values are defined in uppercase so that they are visually distinct from other parameters in the *`model`*`.rtw` file, and, by convention, from user-defined parameters.

TLC Built-In Functions and Values

Built-In Function Name	Expansion
`CAST(expr, expr)`	The first expression must be a string that corresponds to one of the type names in the table "Target Language Value Types" on page 19-13, and the second expression will be cast to that type. A typical use might be to cast a variable to a real format as in `CAST("Real", variable-name)` An example of this is in working with parameter values for S-functions. To use them within C or C++ code, you need to typecast them to `real` so that a value such as `1` will be formatted as `1.0` (see also `%realformat`).
`CONTAINS(expr1, expr2)`	Returns `TLC_TRUE` if `expr1` contains `expr2`, and `TLC_FALSE` otherwise. `expr1` and `expr2` must be strings. For example, `CONTAINS("I walk up, they walked up, we are walking up.", "walk(\\w*) up")` returns `TLC_TRUE`. This is not equivalent to MATLAB function `contains`.
`EXISTS(var)`	If the `var` identifier is not currently in scope, the result is `TLC_FALSE`. If the identifier is in scope, the result is `TLC_TRUE`. `var` can be a single identifier or an expression involving the `.` and `[]` operators.

`FEVAL(matlab-command, TLC-expressions)`	Performs an evaluation in MATLAB. For example,
	
	`%assign result = FEVAL("sin",3.14159)`
	
	The `%matlab` directive can be used to call a MATLAB function that does not return a result. For example,
	
	`%matlab disp(2.718)`
	
	Note: If the MATLAB function returns more than one value, TLC receives the first value only.

`FIELDNAMES(record)`	Returns an array of strings containing the record field names associated with the record. Because it returns a sorted list of strings, the function is `O(n*log(n))`.
`FILE_EXISTS(expr)`	`expr` must be a string. If a file by the name `expr` does not exist on the path, the result is `TLC_FALSE`. If a file by that name exists on the path, the result is `TLC_TRUE`.
`FORMAT(`*`realvalue, format`*`)`	The first expression is a `Real` value to format. The second expression is either `EXPONENTIAL` or `CONCISE`. Outputs the `Real` value in the designated format, where `EXPONENTIAL` uses exponential notation with 16 digits of precision, and `CONCISE` outputs the number in a more readable format while maintaining numerical accuracy.
`FULLFILE(expr,...)`	The function accepts the folder or file names and returns the full file specification. Using this function improves the efficiency of the TLC code when compared to FEVAL function calls to MATLAB function `fullfile`.

Built-In Function Name	Expansion
`GETFIELD(record, "fieldname")`	Returns the contents of the specified field name, if the field name is associated with the record. The function uses hash lookup, and therefore executes in constant time.
`GENERATE(record, function-name, ...)`	Executes function calls mapped to a specific record type (i.e., block record functions). For example, use this to execute the functions in the `.tlc` files for built-in blocks. Not that TLC automatically “scopes” or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line.
`GENERATE_FILENAME(type)`	For the specified record type, does a `.tlc` file exist? Use this to see if the `GENERATE_TYPE` call will succeed.

`GENERATE_FORMATTED_VALUE (expr, string, expand)`	Returns a potentially multiline string that can be used to declare the value(s) of `expr` in the current target language. The second argument is a string that is used as the variable name in a descriptive comment on the first line of the return string. If the second argument is the empty string, `""`, then no descriptive comment is put into the output string. The third argument is a Boolean that when `TRUE` causes `expr` to be expanded into raw text before being output. `expand = TRUE` uses much more memory than the default (`FALSE`); set `expand = TRUE` only if the parameter text needs to be processed for some reason before being written to disk.
	
	For example,
	
	`static const unsigned char contents[] =`
	
	`%assign value = GENERATE_FORMATTED_VALUE(SFcnParamSettings.CONTENTS, "", TLC_FALSE)`
	
	`;`
	
	yields this C code:
	
	`static const unsigned char contents[] = { 0U, 1U, 2U, 3U, 4U };`

`GENERATE_FUNCTION_EXISTS (record, function-name)`	Determines whether a given block function exists. The first expression is the same as the first argument to `GENERATE`, namely a block scoped variable containing a `Type`. The second expression is a string that should match the function name.
`GENERATE_TYPE (record, function-name, type, ...)`	Similar to `GENERATE`, except that `type` overrides the `Type` field of the record. Use this when executing functions mapped to specific S-function block records based upon the S-function name (i.e., the name becomes the type).
`GENERATE_TYPE_FUNCTION_EXISTS (record, function-name, type)`	Same as `GENERATE_FUNCTION_EXISTS` except that it overrides the `Type` built into the record.

Built-In Function Name	Expansion
GET_COMMAND_SWITCH	Returns the values of command-line switches. Only the following switches are supported: v, m, p, O, d, dr, r, I, a See also “Command-Line Arguments” on page 19-49.
IDNUM(expr)	expr must be a string. The result is a vector where the first element is a leading string, if any, and the second element is a number appearing at the end of the input string. For example, IDNUM("ABC123") yields ["ABC", 123]
IMAG(expr)	Returns the imaginary part of a complex number.
INT8MAX	127
INT8MIN	-128
INT16MAX	32767
INT16MIN	-32768
INT32MAX	2147483647
INT32MIN	-2147483648
INTMIN	Minimum integer value on host machine.
INTMAX	Maximum integer value on host machine.
ISALIAS(record)	Returns TLC_TRUE if the record is a reference (symbolic link) to a different record, and TLC_FALSE otherwise.
ISEQUAL(expr1, expr2)	Where the data types of both expressions are numeric: returns TLC_TRUE if the first expression contains the same value as the second expression; returns TLC_FALSE otherwise. Where the data type of either expression is nonnumeric (e.g., string or record): returns TLC_TRUE if and only if both expressions have the same data type and contain the same value; returns TLC_FALSE otherwise.
ISEMPTY(expr)	Returns TLC_TRUE if the expression contains an empty string, vector, or record, and TLC_FALSE otherwise.
ISFIELD(record, "fieldname")	Returns TLC_TRUE if the field name is associated with the record, and TLC_FALSE otherwise.
ISINF(expr)	Returns TLC_TRUE if the value of the expression is inf, and TLC_FALSE otherwise.
ISNAN(expr)	Returns TLC_TRUE if the value of the expression is NAN, and TLC_FALSE otherwise.
ISFINITE(expr)	Returns TLC_TRUE if the value of the expression is not +/- inf or NAN, and TLC_FALSE otherwise.

Built-In Function Name	Expansion
ISSLPRMREF(param.value)	Returns a Boolean value indicating whether its argument is a reference to a Simulink parameter or not. This function supports parameter sharing with Simulink; using it can save memory and time during code generation. For example, %if !ISSLPRMREF(param.Value) assign param.Value = CAST("Real", param.Value) %endif
ISSUBSTRING(expr1, expr2)	Returns TLC_TRUE if the expr2 is a substring of expr1, and TLC_FALSE otherwise. expr1 and expr2 must be a string.
NULL_FILE	A predefined file for no output that you can use as an argument to %selectfile to prevent output.
NUMTLCFILES	The number of target files used thus far in expansion.
OUTPUT_LINES	Returns the number of lines that have been written to the currently selected file or buffer. Does not work for STDOUT or NULL_FILE.
REAL(expr)	Returns the real part of a complex number.
REGEXP_MATCH(expr1, expr2)	Returns the substrings in expr1 that match the pattern expr2. expr1 and expr2 must be strings. For example, REGEXP_MATCH(“I walk up, they walked up, we are walking up.”, “walk(\\w*) up”) returns [“walk up”, “walked up”, “walking up”].
REGEXPREP(expr1, expr2, expr3)	Returns a new string that replaces instances of the substring expr2 in string expr1 with the substring expr3. expr1, expr2 and expr3 must be strings. This function supports tokens in replacement string. For example, REGEXPREP(“I walk up, they walked up, we are walking up.”, “walk(\\w*) up”, “ascend$1”) returns “I ascend, they ascended, we are ascending.”.
REMOVEFIELD(record, "fieldname")	Removes the specified field from the contents of the record. Returns TLC_TRUE if the field was removed; otherwise returns TLC_FALSE.
ROLL_ITERATIONS()	Returns the number of times the current roll regions are looping or NULL if not inside a %roll construct.
SETFIELD(record, "fieldname", value)	Sets the contents of the field name associated with the record. Returns TLC_TRUE if the field was added; otherwise returns TLC_FALSE.
SIZE(expr[,expr])	Calculates the size of the first expression and generates a two-element row vector. If the second operand is specified, it is used as an integer index into this row vector; otherwise the entire row vector is returned. SIZE(x) applied to a scalar returns [1 1]. SIZE(x) applied to a scope returns the number of repeated entries of that scope type. For example, SIZE(Block) returns [1,<number of blocks>]
SPRINTF(format,var,...)	Formats the data in variable var (and in any additional variable arguments) under control of the specified format string, and returns a string variable containing the values. Operates like the C library sprintf(), except that output is the return value rather than contained in an argument to sprintf.

Built-In Function Name	Expansion
STDOUT	A predefined file for stdout output. You can use this as an argument to %selectfile to force output to stdout.
STRING(expr)	Expands the expression into a string; the characters \, \n, and " are escaped by preceding them with \ (backslash). The ANSI escape sequences are translated into string form. If %<> is in the expression, it is escaped so that the enclosed string is not subject to further TLC interpretation.
STRINGOF(expr)	Accepts a vector of UTF-16 values and returns a UTF-8 encoded string that the function constructs by treating each value as a single code point (UTF-16 surrogate pairs are not supported). Use this function primarily for S-function string parameters.
STRNCMP(expr1, expr2, value)	Returns TLC_TRUE if the expr1 is same as the expr2 to the number of characters specified in value, and TLC_FALSE otherwise. expr1 and expr2 must be a string, and value must be a number.
STRNREP(expr1, expr2, expr3, value)	Accepts the string expr1 and returns a new string that replaces all instances of the substring expr2 present in expr1 with the string expr3 for the number of instances specified in value.
STRREP(expr1, expr2, expr3)	Accepts the string expr1 and returns a new string that replaces all instances of the substring expr2 present in expr1 with the string expr3.
SYSNAME(expr)	Looks for specially formatted strings of the form <x>/y and returns x and y as a two-element string vector. This is used to resolve subsystem names. For example, %<sysname("<sub>/Gain")> returns ["sub","Gain"] In Block records, the name of the block is written <*sys*/blockname>, where *sys* is S# or Root. You can obtain the full path name by calling LibGetBlockPath(block); this will include newlines and other troublesome characters that cause display difficulties. To obtain a full path name suitable for one-line comments but not identical to the Simulink path name, use LibGetFormattedBlockPath(block).
TLCFILES	Returns a vector containing the names of the target files included thus far in the expansion. Absolute paths are used. See also NUMTLCFILES.
TLC_FALSE	Boolean constant that equals a negative evaluated Boolean expression.
TLC_TRUE	Boolean constant that equals a positive evaluated Boolean expression.
TLC_TIME	Date and time of compilation.
TLC_VERSION	Version and date of the Target Language Compiler.

Built-In Function Name	Expansion
`TYPE(expr)`	Evaluates `expr` and determines the result type. The result of this function is a string that corresponds to the type of the given expression. See the **Value Type String** column in the table "Target Language Value Types" on page 19-13 for possible values.
`UINT8MAX`	`255U`
`UINT16MAX`	`65535U`
`UINT32MAX`	`4294967295U`
`UINTMAX`	Maximum unsigned integer value on host machine.
`WHITE_SPACE(expr)`	Accepts a string and returns 1 if the string contains only white-space characters (, `\t`, `\n`, `\r`); returns 0 otherwise.
`WILL_ROLL(expr1, expr2)`	The first expression is a roll vector and the second expression is a roll threshold. This function returns `true` if the vector contains a range that will roll.

FEVAL Function

The `FEVAL` built-in function calls MATLAB file functions and MEX-functions. The structure is

```
%assign result = FEVAL( matlab-function-name, rhs1, rhs2, ...
    rhs3, ... );
```

Note

- Only a single left-side argument is allowed when you use `FEVAL`.
- If your MATLAB function evaluation leads to an error, the TLC compiler does not terminate but continues with the execution. The FEVAL directive returns an empty value to the TLC.
- For string operations, it is recommended to use string TLC directives instead of FEVAL functions.

This table shows the conversions that are made when you use `FEVAL`.

MATLAB Conversions

TLC Type	MATLAB Type
`"Boolean"`	`Boolean` (scalar or matrix)
`"Number"`	`Double` (scalar or matrix)
`"Real"`	`Double` (scalar or matrix)
`"Real32"`	`Double` (scalar or matrix)
`"Unsigned"`	`Double` (scalar or matrix)
`"String"`	`String`
`"Vector"`	If the vector is homogeneous, it is converted to a MATLAB vector. If the vector is heterogeneous, it is converted to a MATLAB cell array.
`"Gaussian"`	`Complex` (scalar or matrix)
`"UnsignedGaussian"`	`Complex` (scalar or matrix)
`"Complex"`	`Complex` (scalar or matrix)
`"Complex32"`	`Complex` (scalar or matrix)
`"Identifier"`	`String`
`"Subsystem"`	`String`
`"Range"`	Expanded vector of `Doubles`
`"Idrange"`	Expanded vector of `Doubles`
`"Matrix"`	If the matrix is homogeneous, it is converted to a MATLAB matrix. If the matrix is heterogeneous, it is converted to a MATLAB cell array. (Cell arrays can be nested.)
`"Scope"` or `"Record"`	Structure with elements
`Scope` or `Record` alias	String containing fully qualified alias name
`Scope` or `Record` array	Cell array of structures
Other type not listed above	Conversion not supported

When values are returned from MATLAB, they are converted as shown in this table. Note that conversion of matrices with more than two dimensions is not supported, nor is conversion or downcast of 64-bit integer values.

More Conversions

MATLAB Type	TLC Type
`String`	`String`
`Vector of Strings`	Vector of Strings
`Boolean (scalar or matrix)`	`Boolean` (scalar or matrix)
`INT8,INT16,INT32 (scalar or matrix)`	`Number` (scalar or matrix)
`INT64`	Not supported
`UINT64`	Not supported
`Complex INT8,INT16,INT32 (scalar or matrix)`	Gaussian (scalar or matrix)
`UINT8,UINT16,UINT32 (scalar or matrix)`	`Unsigned` (scalar or matrix)
`Complex UINT8,UINT16,UINT32 (scalar or matrix)`	`UnsignedGaussian` (scalar or matrix)
`Single precision`	`Real32` (scalar or matrix)
`Complex single precision`	`Complex32` (scalar or matrix)
`Double precision`	`Real` (scalar or matrix)
`Complex double precision`	`Complex` (scalar or matrix)
`Sparse matrix`	Expanded to matrix of `Doubles`
`Cell array of structures`	Record array
`Cell array of non-structures`	Vector or matrix of types converted from the types of the elements
`Cell array of structures and non-structures`	Conversion not supported
`Structure`	`Record`
`Object`	Conversion not supported

Other value types are not currently supported.

As an example, this statement uses the `FEVAL` built-in function to call MATLAB to take the sine of the input argument.

```
%assign result = FEVAL( "sin", 3.14159 )
```

Variables (identifiers) can take on the following constant values. Note the suffix on the value .

Constant Form	TLC Type
`1.0`	`"Real"`
`1.0[F/f]`	`"Real32"`
`1`	`"Number"`
`1[U|u]`	`"Unsigned"`
1.0i	`"Complex"`

Constant Form	TLC Type
`1[Ui|ui]`	`"UnsignedGaussian"`
`1i`	`"Gaussian"`
`1.0[Fi|fi]`	`"Complex32"`

Note The suffix controls the Target Language Compiler type obtained from the constant.

This table shows Target Language Compiler constants and their equivalent MATLAB values.

TLC Constants	Equivalent MATLAB Value
`rtInf`, `Inf`, `inf`	`+inf`
`rtMinusInf`	`-inf`
`rtNan`, `NaN`, `nan`	`nan`
`rtInfi`, `Infi`, `infi`	`inf*i`
`rtMinusInfi`	`-inf*i`
`rtNaNi`, `NaNi`, `nani`	`nan*i`

TLC Reserved Constants

For double-precision values, the following are defined for infinite and not-a-number IEEE® values:

```
rtInf, inf, rtMinusInf, -inf, rtNaN, nan
```

For single-precision values, these constants apply:

```
rtInfF, InfF, rtMinusInfF, rtNaNF, NaNF
```

Their corresponding versions when complex are:

```
rtInfi, infi, rtMinusInfi, -infi, rtNaNi (for doubles)
rtInfFi, InfFi, rtMinusInfFi, rtNaNFi, NaNFi (for singles)
```

For integer values, the following are defined:

```
INT8MIN, INT8MAX, INT16MIN, INT16MAX, INT32MIN, INT32MAX,
UINT8MAX, UINT16MAX, UINT32MAX, INTMAX, INTMIN, UINTMAX
```

Identifier Definition

To define or change identifiers (TLC variables), use the directive

```
%assign [::]expression = constant-expression
```

This directive introduces new identifiers (variables) or changes the values of existing ones. The left side can be a qualified reference to a variable using the `.` and `[]` operators, or it can be a single element of a vector or matrix. In the case of the matrix, only the single element is changed by the assignment.

The `%assign` directive inserts new identifiers into the local function scope, file function scope, generate file scope, or into the global scope. Identifiers introduced into the function scope are not

available within functions being called, and are removed upon return from the function. Identifiers inserted into the global scope are persistent. You can change existing identifiers by completely respecifying them. The constant expressions can include legal identifiers from the `.rtw` files. You can use `%undef` to delete identifiers in the same way that you use it to remove macros.

Within the scope of a function, variable assignments create new local variables unless you use the `::` scope resolution operator. In this example, the assignment creates a variable `foo`, local to the function, that will disappear when the function exits.

```
%function ...
...
%assign foo = 3
...
%endfunction
```

Note that `foo` is created even if a global `foo` already exists.

To create or change values in the global scope, you must use the scope resolution operator (`::`) to disambiguate, as in

```
%function ...
%assign foo = 3
%assign ::foo = foo
...
%endfunction
```

The scope resolution operator forces the compiler to assign the value of the local variable `foo`, 3, to the global variable `foo`.

Note It is an error to change a value from the code generator file without qualifying it with the scope. This example does not generate an error:

```
    %assign CompiledModel.name = "newname"    %% No error
```

This example generates an error:

```
%with CompiledModel
  %assign name = "newname"       %% Error     %endwith
```

Creating Records

Use the `%createrecord` directive to build new records in the current scope. For example, if you want to create a new record called `Rec` that contains two items (e.g., `Name "Name"` and `Type "t"`), use

```
%createrecord Rec {  Name "Name"; Type "t" }
```

Adding Records

Use the `%addtorecord` directive to add new records to existing records. For example, if you have a record called `Rec1` that contains a record called `Rec2`, and you want to add an additional `Rec2` to it, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
```

This figure shows the result of adding the record to the existing one.

```
Rec1 {
  Rec2 {
    Name"Name0"          } Existing Record
    Type"t0"
  }
  Rec2 {
    Name"Name1"          } New Record
    Type"t1"
  }
    .
    .
}
```

If you want to access the new record, you can use

```
%assign myname = Rec1.Rec2[1].Name
```

In this same example, if you want to add two records to the existing record, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
%addtorecord Rec1 Rec2 {  Name "Name2"; Type "t2" }
```

This produces

```
Rec1 {
  Rec2 {
    Name"Name0"          } Existing Record
    Type"t0"
  }
  Rec2 {
    Name"Name1"          } First New Record
    Type"t1"
  }
  Rec2 {
    Name"Name2"          } Second New Record
    Type"t2"
  }
    .
    .
}
```

Adding Parameters to an Existing Record

You can use the `%assign` directive to add a new parameter to an existing record. For example,

```
%addtorecord Block[Idx] N 500 /% Adds N with value 500 to Block %/
%assign myn = Block[Idx].N    /% Gets the value 500 %/
```

adds a new parameter, `N`, at the end of an existing block with the name and current value of an existing variable, as shown in this figure. It returns the block value.

```
Block {
  .
  .
  .
  N   500          ———} New Parameter
}
```

Variable Scoping

This section discusses how the Target Language Compiler resolves references to variables (including records).

Scope, in this document, has two related meanings. First, scope is an attribute of a variable that defines its visibility and persistence. For example, a variable defined within the body of a function is visible only within that function, and it persists only as long as that function is executing. Such a variable has function (or local) scope. Each TLC variable has one (and only one) of the scopes described in “Scopes” on page 19-39.

The term scope also refers to a collection, or pool, of variables that have the same scope. At a given point in the execution of a TLC program, several scopes can exist. For example, during execution of a function, a function scope (the pool of variables local to the function) exists. In all cases, a global scope (the pool of global variables) also exists.

To resolve variable references, TLC maintains a search list of current scopes and searches them in a well-defined sequence. The search sequence is described in “How TLC Resolves Variable References” on page 19-43.

Dynamic scoping refers to the process by which TLC creates and deallocates variables and the scopes in which they exist. For example, variables in a function scope exist only while the defining function executes.

Scopes

The following sections describe the possible scopes that a TLC variable can have.

Global Scope

By default, TLC variables have global scope. Global variables are visible to, and can be referenced by, code anywhere in a TLC program. Global variables persist throughout the execution of the TLC program. Global variables are said to belong to the global pool.

Note that the `CompiledModel` record of the *`model`*`.rtw` file has global scope. Therefore, you can access this structure from your TLC functions or files.

You can use the scope resolution operator (`::`) to explicitly reference or create global variables from within a function. See “The Scope Resolution Operator” on page 19-42 for examples.

Note that you can use the `%undef` directive to free memory used by global variables.

File Scope

Variables with file scope are visible only within the file in which they are created. To limit the scope of variables in this way, use the `%filescope` directive anywhere in the defining file.

In the following code fragment, the variables `fs1` and `fs2` have file scope. Note that the `%filescope` directive does not have to be positioned before the statements that create the variables.

```
%assign fs1 = 1
%filescope
%assign fs2 = 3
```

Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This lets you free memory allocated to such variables.

Function (Local) Scope

Variables defined within the body of a function have function scope. That is, they are visible within and local to the defining function. For example, in the following code fragment, the variable `localv` is local to the function `foo`. The variable `x` is global.

```
%assign x = 3

%function foo(arg)
   %assign localv = 1
   %return x + localv
%endfunction
```

A local variable can have the same name as a global variable. To refer, within a function, to identically named local and global variables, you must use the scope resolution operator (`::`) to disambiguate the variable references. See “The Scope Resolution Operator” on page 19-42 for examples.

Note Functions themselves (as opposed to the variables defined within functions) have global scope. There is one exception: functions defined in generate scope are local to that scope. See “Generate Scope” on page 19-40.

%with Scope

The `%with` directive adds a new scope, referred to as a *`%with scope`*, to the current list of scopes. This directive makes it easier to refer to block-scoped variables.

The structure of the `%with` directive is

```
%with expression
%endwith
```

For example, the directive

```
%with CompiledModel.System[sysidx]
   ...
%endwith
```

adds the `CompiledModel.System[sysidx]` scope to the search list. This scope is searched before anything else. You can then refer to the system name simply by

```
Name
```

instead of

```
CompiledModel.System[sysidx].Name
```

Generate Scope

Generate scope is a special scope used by certain built-in functions that are designed to support code generation. These functions dispatch function calls that are mapped to a specific record type. This capability supports a type of polymorphism in which different record types are associated with functions (analogous to methods) of the same name. Typically, this feature is used to map `Block` records to functions that implement the functionality of different block types.

Functions that employ generate scope include GENERATE, GENERATE_TYPE, GENERATE_FUNCTION_EXISTS, and GENERATE_TYPE_FUNCTION_EXISTS. See “GENERATE and GENERATE_TYPE Functions” on page 19-24. This section discusses generate scope using the GENERATE built-in function as an example.

The syntax of the GENERATE function is

```
GENERATE(blk,fn)
```

The first argument (blk) to GENERATE is a valid record name. The second argument (fn) is the name of a function to be dispatched. When a function is dispatched through a GENERATE call, TLC automatically adds blk to the list of scopes that is searched when variable references are resolved. Thus the record (blk) is visible to the dispatched function as if an implicit %with <blk>... %endwith directive existed in the dispatched function.

In this context, the record named blk is said to be in generate scope.

Three TLC files, illustrating the use of generate scope, are listed below. The file polymorph.tlc creates two records representing two hypothetical block types, MyBlock and YourBlock. Each record type has an associated function named aFunc. The block-specific implementations of aFunc are contained in the files MyBlock.tlc and YourBlock.tlc.

Using GENERATE calls, polymorph.tlc dispatches to a function for each block type. Notice that the aFunc implementations can refer to the fields of MyBlock and YourBlock, because these records are in generate scope.

- The following listing shows polymorph.tlc:

  ```
  %% polymorph.tlc

  %language "C"

  %%create records used as scopes within dispatched functions

  %createrecord MyRecord { Type "MyBlock"; data 123 }
  %createrecord YourRecord { Type "YourBlock"; theStuff 666 }

  %% dispatch the functions thru the GENERATE call.

  %% dispatch to MyBlock implementation
  %<GENERATE(MyRecord, "aFunc")>

  %% dispatch to YourBlock implementation
  %<GENERATE(YourRecord, "aFunc")>

  %% end of polymorph.tlc
  ```

- The following listing shows MyBlock.tlc:

  ```
  %%MyBlock.tlc

  %implements "MyBlock" "C"

  %% aFunc is invoked thru a GENERATE call in polymorph.tlc.
  %% MyRecord is in generate scope in this function.
  %% Therefore, fields of MyRecord can be referenced without
  %% qualification
  ```

```
%function aFunc(r) Output
%selectfile STDOUT
The value of MyRecord.data is: %<data>
%closefile STDOUT
%endfunction

%%end of MyBlock.tlc
```

- The following listing shows `YourBlock.tlc`:

```
%%YourBlock.tlc

%implements "YourBlock" "C"

%% aFunc is invoked thru a GENERATE call in polymorph.tlc.
%% YourRecord is in generate scope in this function.
%% Therefore, fields of YourRecord can be referenced without
%% qualification

%function aFunc(r) Output
%selectfile STDOUT
The value of YourRecord.theStuff is: %<theStuff>
%closefile STDOUT
%endfunction

%%end of YourBlock.tlc
```

The invocation and output of `polymorph.tlc`, as displayed in the MATLAB Command Window, are shown below:

```
tlc -v polymorph.tlc

The value of MyRecord.data is: 123
The value of YourRecord.theStuff is: 666
```

Note Functions defined in generate scope are local to that scope. This is an exception to the general rule that functions have global scope. In the above example, for instance, neither of the `aFunc` implementations has global scope.

The Scope Resolution Operator

The scope resolution operator (`::`) is used to indicate that the global scope should be searched when a TLC function looks up a variable reference. The scope resolution operator is often used to change the value of global variables (or even create global variables) from within functions.

By using the scope resolution operator, you can resolve ambiguities that arise when a function references identically named local and global variables. In the following example, a global variable `foo` is created. In addition, the function `myfunc` creates and initializes a local variable named `foo`. The function `myfunc` explicitly references the global variable `foo` by using the scope resolution operator.

```
%assign foo = 3   %% this variable has global scope
.
.
.
```

```
%function myfunc(arg)
   %assign foo = 3   %% this variable has local scope
   %assign ::foo = arg   %% this changes the global variable foo
%endfunction
```

You can also use the scope resolution operator within a function to create global variables. The following function creates and initializes a global variable:

```
%function sideffect(arg)
   %assign ::theglobal = arg   %% this creates a global variable
%endfunction
```

How TLC Resolves Variable References

This section discusses how the Target Language Compiler searches the existing scopes to resolve variable references.

Global Scope

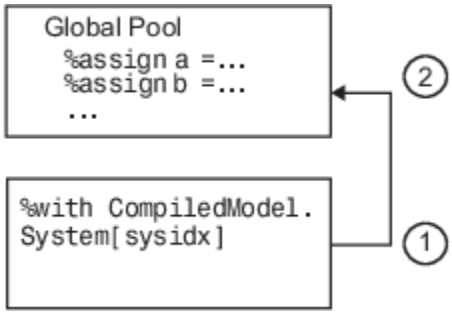
In the simplest case, the Target Language Compiler resolves a variable reference by searching the global pool (including the `CompiledModel` structure).

%with Scope

You can modify the search list and search sequence by using the `%with` directive. For example, when you add the following construct,

```
%with CompiledModel.System[sysidx]
   ...
%endwith
```

the `System[sysidx]` scope is added to the search list. This scope is searched first, as shown by this picture.



This technique makes it simpler to access embedded definitions. Using the `%with` construct (as in the previous example), you can refer to the system name simply by

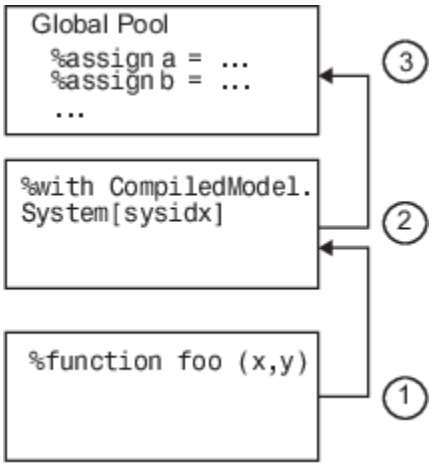
```
Name
```

instead of

```
CompiledModel.System[sysidx].Name
```

Function Scope

A function has its own scope. That scope is added to the previously described search list, as shown in this diagram.

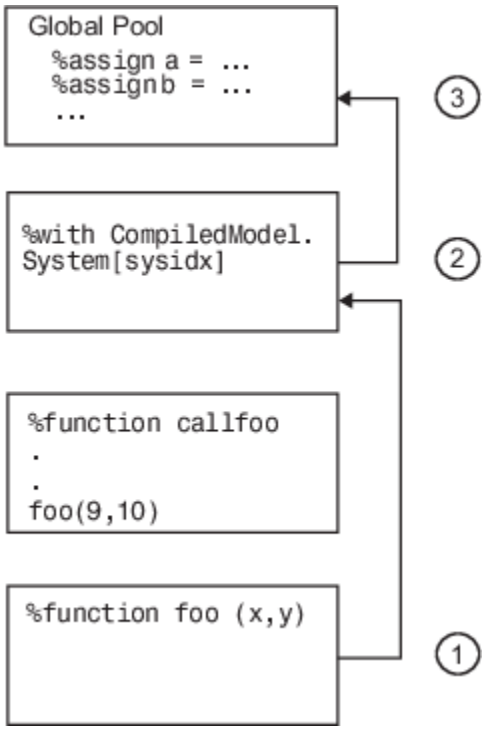


For example, in the following code fragment,

```
% with CompiledModel.System[sysidx]
...
   %assign a=foo(x,y)
...
%endwith
...
%function foo(a,b)
...
   assign myvar=Name
...
%endfunction
...
%<foo(1,2)>
```

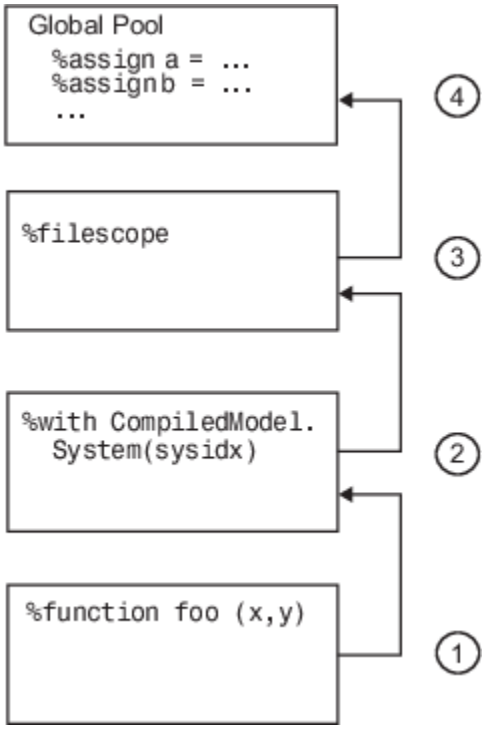
If `Name` is not defined in `foo`, the assignment uses the value of `Name` from the previous scope, `CompiledModel.System[SysIdx].Name`.

In a nested function, only the innermost function scope is searched, together with the enclosing `%with` and global scopes, as shown in the following diagram:



File Scope

File scopes are searched before the global scope, as shown in the following diagram.



The rule for nested file scopes is similar to that for nested function scopes. In the case of nested file scopes, only the innermost nested file scope is searched.

Target Language Functions

The target language function construct is

```
%function identifier ( optional-arguments ) [Output | void]
%return
%endfunction
```

Functions in the target language are recursive and have their own local variable space. Target language functions do not produce output unless they are output functions or explicitly use the `%openfile`, `%selectfile`, and `%closefile` directives.

A function optionally returns a value with the `%return` directive. The returned value can be a type defined in the table at “Target Language Value Types” on page 19-13.

In this example, a function, `name`, returns `x` if `x` and `y` are equal, or returns `z` if `x` and `y` are not equal:

```
%function name(x,y,z) void

%if x == y
   %return x
%else
   %return z
%endif

%endfunction
```

Function calls can appear in contexts where variables are allowed.

The `%with` statements that are in effect when a function is called are available to the function. Calls to other functions do not include the local scope of the function, but do include `%with` statements appearing within the function.

Assignments to variables within a function create new local variables and cannot change the value of global variables unless you use the scope resolution operator (`::`).

By default, a function returns a value and does not produce output. You can override this behavior by specifying the `Output` and `void` modifiers on the function declaration line, as in

```
%function foo() Output
...
%endfunction
```

In this case, the function continues to produce output to the currently open file, and is not required to return a value. You can use the `void` modifier to indicate that the function does not return a value and should not produce output, as in

```
%function foo() void
...
%endfunction
```

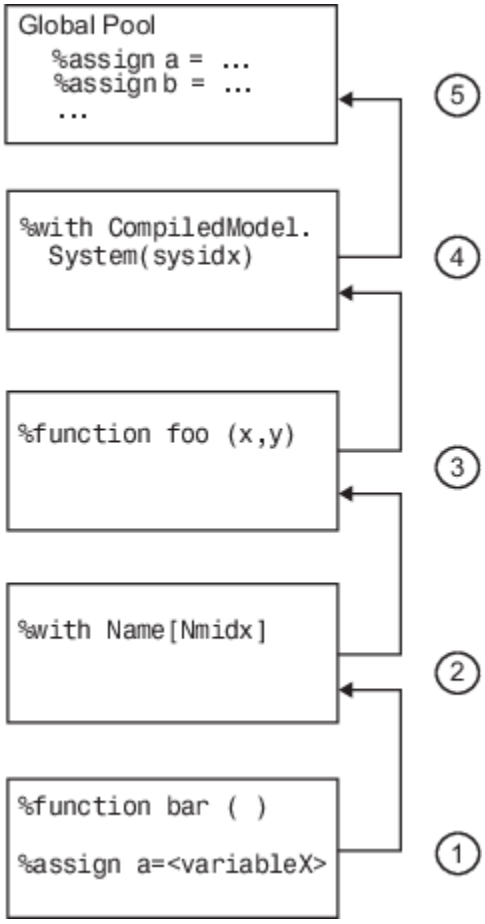
Variable Scoping Within Functions

Within a function, the left-hand member of an `%assign` statement defaults to creating a local variable. A new entry is created in the function's block within the scope chain; it does not affect the other entries. An example appears in "Function Scope" on page 19-43.

You can override this default behavior by using `%assign` with the scope resolution operator (`::`).

When you introduce new scopes within a function, using `%with`, these new scopes are used during nested function calls, but the local scope for the function is not searched.

If a `%with` is included within a function, its associated scope is carried with nested function calls, as shown in the next figure.



%return

The `%return` statement closes all `%with` statements appearing within the current function. In this example, the `%with` statement is automatically closed when the `%return` statement is encountered, removing the scope from the list of searched scopes:

```
%function foo(s)
    %with s
        %return(name)
    %endwith
%endfunction
```

The `%return` statement does not require a value. You can use `%return` to return from a function without a return value.

See Also

Related Examples

- “Command-Line Arguments” on page 19-49

Command-Line Arguments

Target Language Compiler Switches

To call the Target Language Compiler, use

```
tlc [switch1 expr1 switch2 expr2 ...] filename.tlc
```

This table lists the switches you can use with the Target Language Compiler. Order does not makes a difference. Note that if you specify a switch more than once, the last one takes precedence.

Target Language Compiler Switches

Switch	Meaning
`-r filename`	Reads a database file (such as an `.rtw` file). Repeat this option multiple times to load multiple database files into the Target Language Compiler. Omit this option for target language programs that do not depend on the database.
`-v[number]`	Sets the internal verbosity level to *number*. Omitting this option sets the verbosity level to `1`.
`-Ipath`	Adds the specified folder to the list of paths to be searched for TLC files.
`-Opath`	Specifies that the output produced should be placed in the designated folder, including files opened with `%openfile` and `%closefile`, and `.log` files created in debug mode. To place files in the current folder, use `-O` (use the capital letter O, not zero).
`-m[number]`	The *number* specifies the maximum number of errors to report . Without `-m`, the default is to report the first five errors. If the *number* argument is omitted on this option, 1 is assumed.
`-x0`	Parse TLC file only (do not execute).
`-lint`	Performs some simple checks for performance and obsolete features.
`-p[number]`	Prints a dot (`.`) indicating progress for every *number* of TLC primitive operations executed.
`-d[a|c|f|n|o]`	Invokes the TLC's debug mode. `-da` makes TLC execute `%assert` directives. However, when using the build process, this flag is ignored, because it is superseded by the **Enable TLC assertion** check box in the **TLC process** section of the **Code Generation** > **Debug** pane. `-dc` invokes the TLC command-line debugger. `-df filename` invokes the TLC debugger and runs the debugger script specified by *filename*. A debugger script is a text file containing valid debugger commands. TLC searches only the current working folder for the script file. `-dn` causes TLC to produce log files indicating which lines were and were not reached during compilation. `-do` disables the TLC debugging behavior.
`-dr`	Checks for cyclic records (records that reference each other, a source of memory leaks).
`-a[ident]=expr`	Specifies an initial value, *expr*, for the identifier, *ident*, for some parameters; equivalent to the `%assign` command.

Switch	Meaning
`-shadow[0|1]`	Enables a warning when an identifier-value pair of a record overwrites a local variable. The warning is disabled by default. `-shadow0` disables the warning. `-shadow1` enables the warning.

As an example, the command line

```
tlc -r myModel.rtw -v grt.tlc
```

specifies that `myModel.rtw` should be read and used to process `grt.tlc` in verbose mode.

Filenames and Search Paths

Target files have the `.tlc` extension. By default, block-level files have the same name as the `Type` of the block in which they appear. You can override the search path for target files with your own local versions. The Target Language Compiler finds target files along this path. If you specify additional search paths with the `-I` switch of the `tlc` command or via the `%addincludepath` directive, the search order is:

1. Current folder.
2. Include paths specified in `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.
3. Include paths specified at the command line via `-I` . The compiler evaluates multiple `-I` options from *right to left*.

Note The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

See Also

Related Examples

- “Target Language Compiler Directives” on page 19-2

Debugging TLC Files

The Target Language Compiler debugger is a command-line debugger that enables you to identify problems in executing TLC code. The following sections describe the facilities provided and provide examples of use.

- “Using the TLC Debugger” on page 20-2
- “TLC Coverage” on page 20-6
- “TLC Profiler” on page 20-9

Using the TLC Debugger

The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can execute TLC code line-by-line, analyze and/or change variables in a specified block scope, and view the TLC call stack. The TLC debugger has a command-line interface that provides commands similar to standard debugging tools such as `dbx` or `gdb`.

Tips for Debugging TLC Code

Here are a few tips that will help you to debug your TLC code:

1. To see the full TLC call stack, place the following statement in your TLC code before the line that is pointed to by the error message. This will be helpful in narrowing down your problem.

   ```
   %setcommandswitch "-v1"
   ```

2. To trace the value of a variable in a function, place the following statement in your TLC file:

   ```
   %trace This is in my function %<variable>
   ```

   Your message will appear when the Target Language Compiler is run with the `-v` command switch, but not otherwise. You can use `%warning` instead of `%trace` to print variables, but you will need to remove or comment out such lines after you are through debugging.

3. Use the TLC coverage log files to identify parts of your code that have not been reached.

Invoking the Debugger

Use the TLC debugger to identify bugs and potential problems in your TLC files. To use the TLC debugger:

1. On the Configuration Parameters dialog box, select Retain .rtw file. This prevents the *`model`*`.rtw` file from being deleted after code generation.
2. Select Start TLC debugger when generating code to invoke the TLC debugger when starting the code generation process.

   Selecting **Start TLC debugger when generating code** is equivalent to specifying the TLC option `-dc` on the **Code Generation** pane of the Configuration Parameters dialog box.

3. Apply changes and start code generation by pressing **Ctrl+B**. This stops at the first line of executed TLC code, breaks into the TLC command-line debugger, and displays the following prompt:

   ```
   TLC_DEBUG>
   ```

You can now set breakpoints, explore the contents of code generator files, and explore variables in your TLC file using `print`, `which`, or `whos`.

An alternative way to invoke the TLC debugger is from the MATLAB prompt. (This assumes you retained the *`model`*`.rtw` file in the project folder.) To avoid making mistakes, copy the `tlc` command output of the build process to the MATLAB Command Window, and issue it after appending `-dc` to that command line.

A complete list of command-line switches for the TLC debugger is available in the table “Target Language Compiler Switches” on page 19-49.

TLC Debugger Command Summary

The table TLC Debugger Commands summarizes the TLC debugger commands.

To obtain more detailed help on individual commands, use the syntax

```
help command
```

from within the TLC debugger, as in this example:

```
TLC-DEBUG> help clear
```

You can abbreviate a TLC debugger command to its shortest unique form. For example,

```
TLC-DEBUG> break warning
```

can be abbreviated to

```
TLC-DEBUG> br warning
```

To view a complete list of TLC debugger commands, type `help` at the `TLC-DEBUG>` prompt.

TLC Debugger Commands

Command	Description
`assign variable=value`	Change a variable in the running program.
`break ["filename":]line|error|warning|trace|function`	Set a breakpoint. See also “%breakpoint Directive” on page 20-5.
`clear [breakpoint#|all]`	Remove a breakpoint.
`condition [breakpoint#] [expression]`	Attach a condition to a breakpoint.
`continue ["filename":]line|function`	Continue from a breakpoint.
`disable [breakpoint#]`	Disable a breakpoint.
`down [n]`	Move down the stack.
`enable [breakpoint#]`	Enable a breakpoint.
`finish`	Break after completing the current function.
`help [command]`	Obtain help for a command.
`ignore [breakpoint#]count`	Set the ignore count of a breakpoint.
`iostack`	Display contents of I/O stack.
`list start[,end]`	List lines from the file from start to end.
`loadstate "filename"`	Load debugger breakpoint state from a file.
`next`	Single step without going into functions.
`print expression`	Print the value of a TLC expression. To print a record, you must specify a fully qualified scope such as `CompiledModel.System[0].Block[0]`.
`quit`	Quit the TLC debugger. You can also exit the debugger by typing **Ctrl+C** at the prompt.
`run "filename"`	Run a batch file of command-line debugger commands.
`savestate "filename"`	Save debugger breakpoint state to a file.
`status`	Display a list of active breakpoints.
`step`	Step into.
`stop ["filename":]line|error|warning|trace|function`	Set a breakpoint (same as `break`).
`tbreak ["filename":]line|function`	Set a temporary breakpoint.
`thread [n]`	Change the active thread to thread `#n` (0 is the main program’s thread number).
`threads`	List the currently active TLC execution threads.
`tstop ["filename":]line|function`	Set a temporary breakpoint.
`up [n]`	Move up the stack.
`where`	Show the currently active execution chains.
`which name`	Look up the name and display what scope it comes from.
`whos [::|expression]`	List the variables in the given scope.

%breakpoint Directive

As an alternative to the `break` command, you can embed breakpoints at locations in a TLC file by adding the directive

```
%breakpoint
```

Usage Notes

When using `break` or `stop`, use

- `error` to break or stop on error
- `warn` to break or stop on warning
- `trace` to break or stop on trace

For example, if you need to break on error, use

```
TLC_DEBUG> break error
```

When using `clear`, get the status of breakpoints using `status` and clear specific breakpoints. For example

```
TLC-DEBUG> break "foo.tlc":46
TLC-DEBUG> break "foo.tlc":25
TLC-DEBUG> status
Breakpoints:
[1] break File: foo.tlc Line: 46
[2] break File: foo.tlc Line: 25
TLC-DEBUG> clear 2
```

In this example, `clear 2` clears the second breakpoint.

See Also

Related Examples

- “Debug Your TLC Code” on page 16-26
- “TLC Code Coverage to Aid Debugging” on page 16-32
- “TLC Coverage” on page 20-6

TLC Coverage

Using the TLC Coverage Option

The example in “Using the TLC Debugger” on page 20-2 used the debugger to detect a problem in one section of the TLC file. Because a test model may not cover all possible cases, there is a technique that traces the untested cases, the TLC coverage option.

The TLC coverage option provides an easier way to ascertain that the different code parts (not paths) in your code are exercised. To specify TLC coverage tracking, on the Configuration Parameters dialog box, select Start TLC coverage when generating code.

When you initiate TLC coverage, the Target Language Compiler produces a `.log` file for every target file (`*.tlc`) used. These `.log` files are placed in project folder created for the model. Each `.log` file contains usage (count) information regarding how many times it encounters each line during execution. Each line begins with the number of times it is encountered, followed by a colon, followed by the code.

Example .log File

Here is a log file that results from generating code for the example model `sfcndemo_sdotproduct`, located in the folder *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures` (open). This model inlines the `sdotproduct` S-function in TLC. The TLC file that implements the S-function is located in the folder *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c` (open). The `.log` file for `sdotproduct.tlc` is `sdotproduct.log`, which is placed in your build folder. The contents of `sdotproduct.log` are similar to:

```
Source: E:\matlab\toolbox\simulink\simdemos\simfeatures\tlc_c\sdotproduct.tlc
 0: %%
 0: %% File : sdotproduct.tlc generated from sdotproduct.ttlc revision 1.6
 0: %%
 0: %% Abstract:
 0: %%      Dot product block target file.
 1:
 1: %implements sdotproduct "C"
 1:
 0: %% Function: FcnThriftedComplexMultiply
========================================
 0: %% Abstract:
 0: %%      This function multiplies two numbers in the complex plane. If any of
 0: %%      the input arguments is only real, then the complex part is passed in
 0: %%      as "".
 0: %%
 1: %function FcnThriftedComplexConjMultiply(ar,ai,br,bi,cr,ci,op) void
 2:   %openfile buffer
 0:   %%
 0:   %% Compute Cr = Ar * Br + Ai * Bi
 0:   %%
 2:   %assign rhsStr = "%<ar> * %<br>"
 2:   %if !LibIsEqual(ai, "") && !LibIsEqual(bi, "")
 0:     %assign rhsStr = rhsStr + " + %<ai> * %<bi>"
 0:   %endif
 2:   %<cr> %<op> %<rhsStr>;
 0:   %%
 0:   %% Compute Ci = Ar * Bi - Ai * Br
 0:   %%
 2:   %if !LibIsEqual(ci, "")
 0:     %assign rhsStr = "0.0"
 0:     %if !LibIsEqual(bi, "")
 0:       %assign rhsStr = "%<ar> * %<bi>"
 0:     %endif
```

```
 0:     %if !LibIsEqual(ai, "")
 0:       %assign rhsStr = rhsStr + " - %<ai> * %<br>"
 0:     %endif
 0:     %<ci> %<op> %<rhsStr>;
 0:   %endif
 0:   %%
 2:   %closefile buffer
 2:   %return buffer
 0: %endfunction %% FcnThriftedComplexMultiply
 1:
 1:
 0: %% Function: Outputs
============================================================
 0: %% Abstract:
 0: %%      Y = U0' * U1, where U0' is the complex conjugate transpose of U0
 0: %%
 1: %function Outputs(block, system) Output
 1:   %assign sfcnName = ParamSettings.FunctionName
 1:   /* %<Type> Block (%<sfcnName>): %<LibParentMaskBlockName(block)> */
 0:   %%
 1:   %assign u0re = LibBlockInputSignal(0, "", "", "%<tRealPart>0")
 1:   %assign u0im = LibBlockInputSignal(0, "", "", "%<tImagPart>0")
 1:   %assign u1re = LibBlockInputSignal(1, "", "", "%<tRealPart>0")
 1:   %assign u1im = LibBlockInputSignal(1, "", "", "%<tImagPart>0")
 0:   %%
 1:   %assign yre = LibBlockOutputSignal(0, "", "", "%<tRealPart>0")
 1:   %assign yim = LibBlockOutputSignal(0, "", "", "%<tImagPart>0")
 0:   %%
 0:   %% Need to declare a temporary variable for u1re when the output is
 0:   %% being overwritten and u0im is nonzero
 1:   %assign outputOverWritesInput = ...
 0:     ((LibBlockInputSignalBufferDstPort(0) == 0) || ...
 0:      (LibBlockInputSignalBufferDstPort(1) == 0)) &&   ...
 0:     (LibBlockInputSignalIsComplex(0) && LibBlockInputSignalIsComplex(1))
 0:   %%
 1:   %if outputOverWritesInput
 0:     {
 0:       %assign dtName = LibBlockOutputSignalDataTypeName(0, tRealPart)
 0:       %<dtName> tmpVar;
 0:     \
 0:     %assign tmpVar = "tmpVar"
 0:   %else
 1:     %assign tmpVar = yre
 0:   %endif
 0:   %%
 1:   %<FcnThriftedComplexConjMultiply(u0re, u0im, u1re, u1im, tmpVar, yim, "=")>\
 0:   %%
 1:   %assign rollVars    = ["U", "Y"]
 1:   %assign rollRegion  = LibGetRollRegions1(RollRegions)
 0:   %%
 1:   %if LibIsEqual(rollRegion, [])
 0:     %if outputOverWritesInput
 0:       %<yre> = tmpVar;
 0:     %endif
 0:   %else
 0:     %% Continue with dot product for nonscalar case
 1:     %roll idx = rollRegion, lcv = RollThreshold, block, "Roller", rollVars
 1:       %assign u0re = LibBlockInputSignal(0,"",lcv,"%<tRealPart>%<idx>")
 1:       %assign u0im = LibBlockInputSignal(0,"",lcv,"%<tImagPart>%<idx>")
 1:       %assign u1re = LibBlockInputSignal(1,"",lcv,"%<tRealPart>%<idx>")
 1:       %assign u1im = LibBlockInputSignal(1,"",lcv,"%<tImagPart>%<idx>")
 0:       %%
 1:       %assign yre = LibBlockOutputSignal(0,"",lcv,"%<tRealPart>%<idx>")
 1:       %assign yim = LibBlockOutputSignal(0,"",lcv,"%<tImagPart>%<idx>")
 0:       %%
 1:       %<FcnThriftedComplexConjMultiply(u0re, u0im, u1re, u1im, yre, yim, "+=")>\
 0:     %endroll
 0:   %endif
 1:   %if outputOverWritesInput
 0:     }
 0:   %endif
 1:
```

```
0: %endfunction
1:
0: %% [EOF] sdotproduct.tlc
```

Analyzing the Results

This structure makes it easy to identify branches not taken and to develop new tests that can exercise unused portions of the target files.

Looking at the `sdotproduct.log` file, you can see that the code has not been used to assign default values to parameters (e.g., the first part of the code for function `FcnThriftedComplexConjMultiply`). Using this log as a reference and creating models that exercise unexecuted lines, you can make sure that your code is more robust.

See Also

Related Examples

- “Debug Your TLC Code” on page 16-26
- “TLC Code Coverage to Aid Debugging” on page 16-32
- “TLC Profiler” on page 20-9

TLC Profiler

Using the Profiler

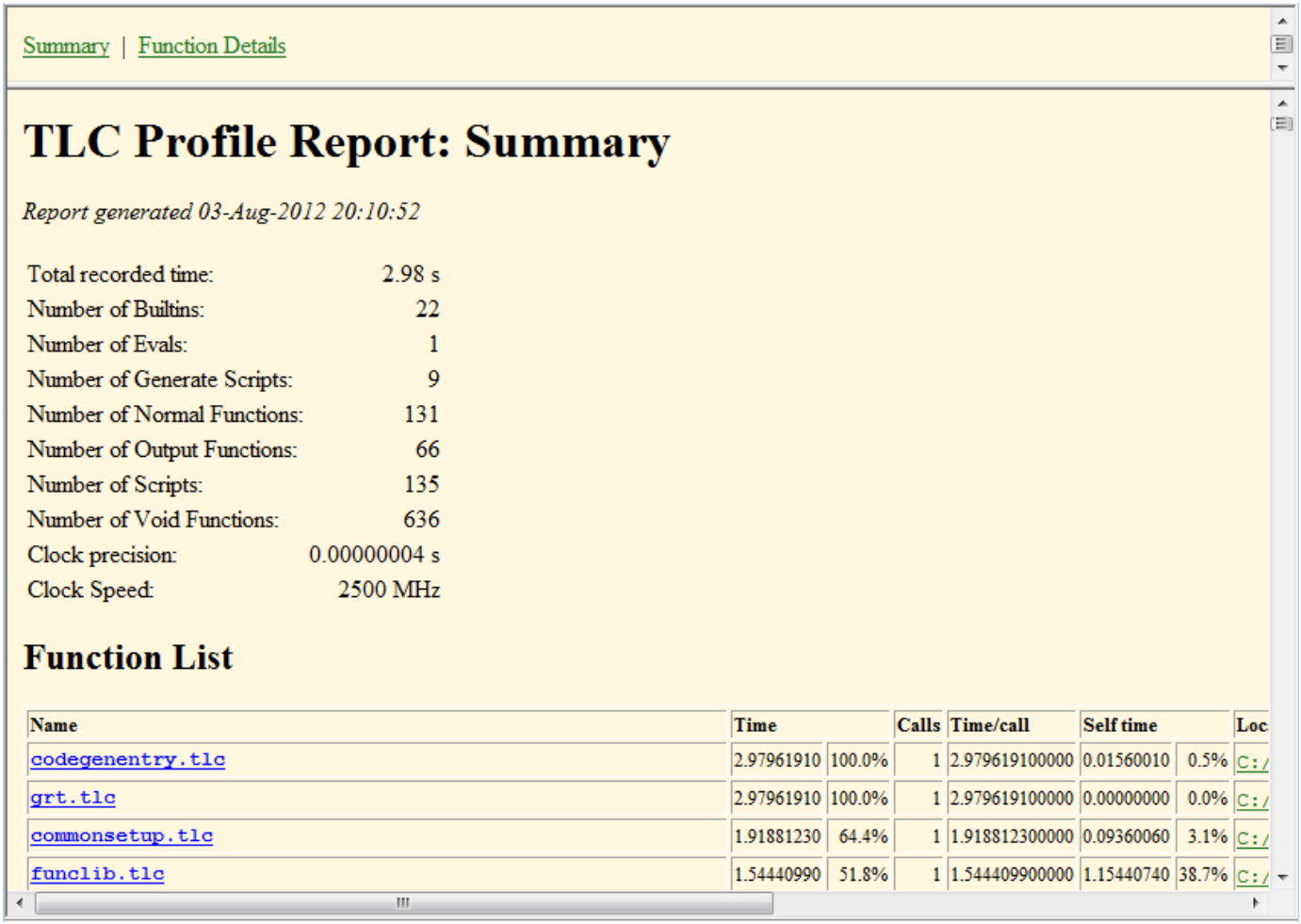
The TLC profiler collects timing statistics for TLC code. It collects execution time for functions, scripts, macros, and built-in functions. These results become the basis of HTML reports that are identical in format to MATLAB profiler reports. By analyzing the report, you can identify bottlenecks in your code that make code generation take longer.

On the Configuration Parameters dialog box, select Profile TLC. Apply your changes and press **Ctrl +B**.

At the end of the TLC process, the build process creates the HTML summary and related files.

Analyzing the Report

The profile report is generated into the build folder. To open the report, change folder (`cd`) to the build folder and open the file *`model`*`.html`, opening it in a browser window. Here is a sample of a TLC profiling report:



Summary | Function Details

TLC Profile Report: Summary

Report generated 03-Aug-2012 20:10:52

Total recorded time:	2.98 s
Number of Builtins:	22
Number of Evals:	1
Number of Generate Scripts:	9
Number of Normal Functions:	131
Number of Output Functions:	66
Number of Scripts:	135
Number of Void Functions:	636
Clock precision:	0.00000004 s
Clock Speed:	2500 MHz

Function List

Name	Time		Calls	Time/call	Self time		Loc
codegenentry.tlc	2.97961910	100.0%	1	2.979619100000	0.01560010	0.5%	C:/
grt.tlc	2.97961910	100.0%	1	2.979619100000	0.00000000	0.0%	C:/
commonsetup.tlc	1.91881230	64.4%	1	1.918812300000	0.09360060	3.1%	C:/
funclib.tlc	1.54440990	51.8%	1	1.544409900000	1.15440740	38.7%	C:/

The created report is fairly self-explanatory. Some points to note are

- Functions are sorted in descending order of their execution time.
- Self-time is the time spent in the function alone, not including the time spent in functions called by the function.
- Functions are hyperlinks that take you to the details related to that specific function.

The profiler report can be helpful when you have inlined S-functions in your model. You can use the profiler to compare time spent in specific user-written or `Lib` functions, and then modify your TLC code accordingly.

Nonexecutable Directives

TLC considers the following directives to be nonexecutable lines. Therefore, these directives are not counted in TLC Profiler reports:

- `%filescope`
- `%else`
- `%endif`
- `%endforeach`
- `%endfor`
- `%endroll`
- `%endwith`
- `%body`
- `%endbody`
- `%endfunction`
- `%endswitch`
- `%default`
- Comment (`%%` or `/% text %/`

Improving Performance

Analyzing the profiler results also gives you an overview of which functions are used more often or are more expensive. Then, you can either improve those functions, or try alternative methods to improve code generation speed. Two points to consider are

- Reduce usage of `EXISTS`. Performing an `EXISTS` on a field is more costly than comparing the field to a value. When possible, create an inert default value for a field. Then, instead of doing an `EXISTS` on the entity, compare it against the default value.
- Reduce the use of one-line functions. One-line functions might be a bottleneck for code generation speed. When readability is not an issue, consider expanding the function.

See Also

Related Examples

- “TLC Coverage” on page 20-6

Inlining S-Functions

To wrap or to inline, that is the question. Once you have decided, the following sections explain how to go about it, using the `timestwo` S-function as a running example. Inlining works almost identically for C, C++, MATLAB file, and Fortran S-functions.

- “Inline S-Functions” on page 21-2
- “Inline C MEX S-Functions” on page 21-8
- “TLC Coding Conventions” on page 21-17
- “Block Target File Methods” on page 21-21
- “Loop Rolling” on page 21-27

Inline S-Functions

Writing S-functions to be included in generated code involves requirements that go beyond writing S-functions used only for simulation. Before you proceed to inline an S-function make sure that it meets requirements and functions as you expect. For more information, see “S-Functions and Code Generation”. If your S-function is multirate, see “Time-Based Scheduling and Code Generation” and “Modeling for Multitasking Execution”, and “Rate Grouping Compliance and Compatibility Issues” on page 71-12.

Inline S-Functions with Block Target Files

When to Inline S-Functions

With C MEX S-functions, non-ERT targets support calling the original C MEX code if the source code (`.c` file) is available when entering the build phase. For S-functions that are in Fortran or MATLAB language, you must inline them to have complete code generation for Simulink models that contain them. Additionally, once you have determined that you will inline an S-function, you must decide to make it either fully inlined or wrapped.

Fully Inlined S-Functions

The block target file for a fully inlined S-function is a self-contained definition of how to inline the block’s functionality directly into the various portions of the generated code — start code, output code, etc. This approach is most beneficial when there are many modes and data types supported for algorithms that are relatively small or when the code size is not significant.

Function-Based or Wrapped Code Generation

When the physical size of the code for a block becomes too large for inlining, the block target file is written to gather inputs, outputs, and parameters, and make a call to a function that you write to perform the block functionality. This has an advantage in generated code size when the code in the function is large or there are many instances of this block in a model. Of course, you should consider the overhead of the function call when weighing the option of fully inlining the block algorithm or generating function calls.

If you choose to go with function-based code generation, two more options need consideration:

- Write the functions once, put them in `.c` files, and have the TLC code’s `BlockTypeSetup` method specify external references to your support functions. Use `LibAddToModelSources` for names of the modules containing the supporting functions. This approach is usually done using one function per file to get the smallest executable possible.
- Write a more sophisticated TLC file. In addition to methods such as `Start` and `Outputs`, conditionally generate customized versions of functions (data types, widths, algorithms, and so on), in separate code generation buffers, to be written to a separate `.c` file. The file should contain only functions used by this model, instead of all possible functions.

Either approach can produce optimal code. The first option can result in hundreds of files if your S-function supports many data types, signal widths, and algorithm choices. The second approach is more difficult to write, but results in a more maintainable code generation library, and the code can be every bit as tight as the first approach.

For further information on wrapping, see “Wrapper Inlined S-Function Example” on page 15-11.

Inline MATLAB File S-Functions

You can inline the functionality of MATLAB file S-functions in the generated code. The process for writing a block target file for a MATLAB file S-function is essentially identical to the process for writing a C MEX S-function.

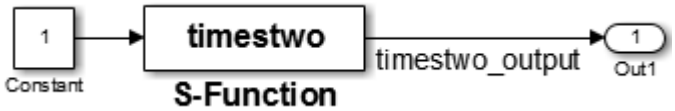
Note While you can fully inline a MATLAB file S-function to improve performance, Simulink Accelerator or the code generator does not include a C or C++ API for the MATLAB Math Library. You therefore cannot call MATLAB Math Library functions from a TLC file.

The following example illustrates the equivalence of C MEX and MATLAB file S-functions for code generation. The S-function MATLAB file `timestwo.m` is equivalent to the C MEX S-function `timestwo`. The TLC file for the C MEX S-function `timestwo` works for the S-function MATLAB file `timestwo.m`. TLC is independent of the type of S-function because TLC requires only the root name of the S-function and not its type. In the case of `timestwo`, one line determines how the code generator implements the TLC file:

```
%implements "timestwo" "C"
```

To try this yourself:

1. Create the following sample model:



2. Copy the file `timestwo.m` from the folder *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures` (open) to a temporary folder.
3. Copy the file `timestwo.tlc` from the folder *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c` (open) to the same temporary folder.
4. In MATLAB, change folder (`cd`) to the temporary folder and make a Simulink model with an S-function block that calls `timestwo.`
5. On the **Signal Attributes** tab of the Inport Block Parameters dialog box, set the **Port dimensions** parameter to 5.

Simulink uses the MATLAB file S-function for simulation because the MATLAB search path finds `timestwo.m` in the current folder before finding the C MEX S-function `timestwo` in the `matlabpath`. Verify which S-function the code generator uses by typing the MATLAB command:

```
which timestwo
```

The answer is the MATLAB file S-function `timestwo.m` in the temporary folder.

In the generated code, the `timestwo.tlc` file inlines the MATLAB file S-function.

```
/* S-Function (timestwo): '<Root>/MATLAB S-Function' */
  /* Multiply input by two */
  {
    int_T i1;
    const real_T *u0 = &times2_B.Gain[0];
    real_T *y0 = &times2_Y.Out1[0];
    for (i1=0; i1 < 5; i1++) {
```

```
      y0[i1] = u0[i1] * 2.0;
    }
  }
```

The output is the product of each input, `u0[i1]` times 2.0. The code generator uses this `Outputs` method from the block target file to generate code:

```
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
  %endroll
%endfunction
```

Alter these temporary copies of the MATLAB file S-function and the TLC file to see how they interact. Start out by just changing the comments in the TLC file and see the changes that appear in the generated code. Then, work up to algorithmic changes.

For more information on inlining C MEX S-Functions, see “Inline C MEX S-Functions” on page 21-8.

Inline Fortran (F-MEX) S-Functions

The capabilities of Fortran MEX S-functions can be fully inlined using a TLC block target file. This interface can be illustrated with a Fortran MEX S-function that implements the `timestwo` function. Here is the sample Fortran S-function code:

```
C
C     FTIMESTWO.FOR
C
C
C     A sample FORTRAN representation of a
C     timestwo S-function.
C     Copyright 1990-2000 The MathWorks, Inc.
C
C=====================================================
C     Function:  SIZES
C
C     Abstract:
C       Set the size vector.
C
C       SIZES returns a vector which determines model
C       characteristics.  This vector contains the
C       sizes of the state vector and other
C       parameters. More precisely,
C       SIZE(1)  number of continuous states
C       SIZE(2)  number of discrete states
C       SIZE(3)  number of outputs
C       SIZE(4)  number of inputs
C       SIZE(5)  number of discontinuous roots in
C                the system
C       SIZE(6)  set to 1 if the system has direct
C                feedthrough of its inputs,
C                otherwise 0
C
```

```
C=====================================================
      SUBROUTINE SIZES(SIZE)
C     .. Array arguments ..
      INTEGER*4        SIZE(*)
C     .. Parameters ..
      INTEGER*4       NSIZES
      PARAMETER       (NSIZES=6)

      SIZE(1) = 0
      SIZE(2) = 0
      SIZE(3) = 1
      SIZE(4) = 1
      SIZE(5) = 0
      SIZE(6) = 1

      RETURN
      END

C
C=====================================================
C     Function:  OUTPUT
C
C     Abstract:
C       Perform output calculations for continuous
C       signals.
C=====================================================
C     .. Parameters ..
      SUBROUTINE OUTPUT(T, X, U, Y)
      REAL*8           T
      REAL*8           X(*), U(*), Y(*)

      Y(1) = U(1) * 2.0

      RETURN
      END

C
C=====================================================
C     Stubs for unused functions.
C=====================================================

      SUBROUTINE INITCOND(X0)
      REAL*8           X0(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DERIVS(T, X, U, DX)
      REAL*8           T, X(*), U(*), DX(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DSTATES(T, X, U, XNEW)
      REAL*8           T, X(*), U(*), XNEW(*)
C --- Nothing to do.
      RETURN
      END
```

```
      SUBROUTINE DOUTPUT(T, X, U, Y)
      REAL*8           T, X(*), U(*), Y(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE TSAMPL(T, X, U, TS, OFFSET)
      REAL*8           T,TS,OFFSET,X(*),U(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE SINGUL(T, X, U, SING)
      REAL*8           T, X(*), U(*), SING(*)
C --- Nothing to do.
      RETURN
      END
```

Copy the preceding code into file `ftimestwo.for` in a convenient working folder.

Putting this into an S-function block in a simple model will illustrate the interface for inlining the S-function. Once your Fortran MEX environment is set up, prepare the code for use by compiling the S-function in a working folder along with the file `simulink.for` from the folder *`matlabroot/simulink/src`* (open). For more information about setting up your Fortran MEX environment, see “Create Level-2 Fortran S-Functions”.

Compile the code with the `mex` command at the MATLAB command line:

```
mex ftimestwo.for simulink.for
```

Now reference this block from a simple model set with a fixed-step solver and the `grt` target.

The TLC code for inlining this block is a modified form of `timestwo.tlc`. In your working folder, create a file named `ftimestwo.tlc` and put this code into it.

```
%implements "ftimestwo" "C"

%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, ...
"Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
  %endroll
%endfunction
```

Now you can generate code for the `ftimestwo` Fortran MEX S-function. The resulting code fragment specific to `ftimestwo` is

```
/* S-Function Block: <Root>/F-MEX S-Function */
  /* Multiply input by two */
  rtB.F_MEX_S_Function = rtB.Gain * 2.0;
```

See Also

Related Examples

- “Write Wrapper S-Function and TLC Files”
- “Write Fully Inlined S-Functions”
- “Write Fully Inlined S-Functions with mdlRTW Routine”

Inline C MEX S-Functions

Inline S-Function Overview

When a Simulink model contains an S-function and a corresponding TLC block target file exists for that S-function, the code generator inlines the S-function. Inlining an S-function can produce more efficient code by eliminating the S-function API layer from the generated code.

For S-functions that can perform a variety of tasks, inlining them gives you the opportunity to generate code only for the current mode of operation set for each instance of the block. As an example of this, if an S-function accepts an arbitrary signal width and loops through each element of the signal, you would want to generate inlined code that has loops when the signal has two or more elements, but generates a simple nonlooped calculation when the signal has just one element.

Level 1 C MEX S-functions (written to an older form of the S-function API) that are not inlined will cause the generated code to make calls to all of these functions even if the routine is empty for the particular S-function.

Function	Purpose
`mdlInitializeSizes`	Initialize the sizes array
`mdlInitializeSampleTimes`	Initialize the sample times array
`mdlInitializeConditions`	Initialize the states
`mdlOutputs`	Compute the outputs
`mdlUpdate`	Update discrete states
`mdlDerivatives`	Compute the derivatives of continuous states
`mdlTerminate`	Clean up when the simulation terminates

Level 2 C MEX S-functions (i.e., those written to the current S-function API) that are not inlined make calls to the above functions, with the following exceptions:

- `mdlInitializeConditions` is called only if `MDL_INITIALIZE_CONDITIONS` is declared with `#define`.
- `mdlStart` is called only if `MDL_START` is declared with `#define`.
- `mdlUpdate` is called only if `MDL_UPDATE` is declared with `#define`.
- `mdlDerivatives` is called only if `MDL_DERIVATIVES` is declared with `#define`.

By inlining an S-function, you can eliminate the calls to these possibly empty functions in the simulation loop. This can greatly improve the efficiency of the generated code.

To inline an S-function called *`sfunc`*`_name`, you create a custom S-function block target file called *`sfunc`*`_name.tlc` and place it in the same folder as the S-function MEX-file. Then, at build time, the target file is executed instead of setting up function calls into the S-function `.c` file. The S-function target file “inlines” the S-function by directing the Target Language Compiler to insert only the statements defined in the target file.

In general, inlining an S-function is especially useful when

- The time required to execute the contents of the S-function is small in comparison to the overhead required to call the S-function.

- Certain S-function routines are empty (e.g., `mdlUpdate`).
- The behavior of the S-function changes between simulation and code generation. For example, device driver I/O S-functions might read from the MATLAB workspace during simulation, but read from an actual hardware address in the generated code.

S-Function Parameters

An S-function can write two different types of parameters into the *`model`*`.rtw` file for Target Language Compiler files to access:

- Parameter settings: These correspond to nontunable parameters (typically set from check boxes and menus on a masked S-function) that are written via the `mdlRTW` method of the S-function using `ssWriteRTWParamSettings`. The S-function TLC implementation file can then directly access the values of these parameter settings from the `SFcnParamSettings` record in the block.
- Tunable parameters: This class of parameters can be accessed when they are registered as run-time parameters within the S-function. Note that such tunable parameters are automatically written out to the *`model`*`.rtw` file. Within the TLC file for the S-function, you can access run-time parameters and their attributes using the `LibBlockParameter` library function and its variants.

For more information on how to create and use run-time parameters, see “Create and Update S-Function Run-Time Parameters”. Also see the example `sfcndemo_runtime` in the S-function examples for how to create and use the two classes of parameters. The example source files, which you can inspect and adapt, are

- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime1.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime1.tlc`
- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime2.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime2.tlc`
- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime3.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime3.tlc`

Sample Code for S-Function

Suppose you have a simple S-function that mimics the Gain block, with one input, one output, and a scalar gain. That is, `y = u * p`. If the Simulink block’s name is `foo` and the name of the Level 2 S-function is `foogain`, the C MEX S-function must contain this code:

```
#define S_FUNCTION_NAME foogain
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#define GAIN mxGetPr(ssGetSFcnParam(S,0))[0]

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates        (S, 0);
  ssSetNumDiscStates        (S, 0);

  if (!ssSetNumInputPorts(S, 1)) return;
  ssSetInputPortWidth            (S, 0, 1);
  ssSetInputPortDirectFeedThrough(S, 0, 1);
```

```
  if (!ssSetNumOutputPorts(S, 1)) return;
  ssSetOutputPortWidth          (S, 0, 1);

  ssSetNumSFcnParams        (S, 1);
  ssSetNumSampleTimes        (S, 0);
  ssSetNumIWork        (S, 0);
  ssSetNumRWork        (S, 0);
  ssSetNumPWork        (S, 0);
}

static void
mdlOutputs(SimStruct *S, int_T tid)
{
  real_T *y = ssGetOutputPortRealSignal(S, 0);
  const InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);

  y[0] = (*u)[0] * GAIN;
}

static void
mdlInitializeSampleTimes(SimStruct *S){}

static void
mdlTerminate(SimStruct *S) {}

#define MDL_RTW  /* Change to #undef to remove function */
#if defined(MDL_RTW)&&(defined(MATLAB_MEX_FILE)||defined(NRT))
static void
mdlRTW (SimStruct *S)
{
  if (!ssWriteRTWParameters(S, 1,SSWRITE_VALUE_VECT,"Gain","",
                            mxGetPr(ssGetSFcnParam(S,0)),1))
  {
    return;
  }
}
#endif

#ifdef  MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

The following two sections show the difference in the generated code for *`model`*`.c` containing noninlined and inlined versions of S-function `foogain`. The model contains no other Simulink blocks.

For more information about these S-function related C library functions, see “Configure C/C++ S-Function Features”. For information about how to generate code, see “Configure Model and Generate Code” and “Approaches for Building Code Generated from Simulink Models”.

Comparison of Noninlined and Inlined Versions of model.c

Without a TLC file to define the S-function specifics, the code generator must call the MEX-file S-function through the S-function API. The following code is the *`model`*`.c` file for the noninlined S-function (i.e., no corresponding TLC file exists).

Noninlined S-Function

```
/*
 * model.c
.
.
.
*/
real_T untitled_RGND = 0.0;             /* real_T ground */
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}
#include "model_reg.h"
/* [EOF] model.c */
```

Inlined S-Function

This code is *model*.c with the `foogain` S-function fully inlined:

```
/*
 * model.c
.
.
.
*/
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}

/* Compute block outputs */
void MdlOutputs(int_T tid)
```

```
  /* S-Function block: <Root>/S-Function */
  /* NOTE: There are no calls to the S-function API in the inlined
     version of model.c. */
  rtB.S_Function = 0.0 * rtP.S_Function_Gain;
}

/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}

/* Terminate function */
void MdlTerminate(void)
{
  /* (no terminate code required) */
}

#include "model_reg.h"

/* [EOF] model.c */
```

If you include this target file for this S-function block, the resulting `model.c` code is

```
rtB.S_Function = 0.0 * rtP.S_Function_Gain;
```

Including a TLC file drastically decreased the code size and increased the execution efficiency of the generated code. These notes highlight some information about the TLC code and the generated output:

- The TLC directive `%implements` is required by block target files, and must be the first executable statement in the block target file. This directive prevents the Target Language Compiler from executing an inappropriate target file for S-function `foogain`.
- The input to `foo` is `rtGROUND` (a Simulink Coder global equal to 0.0) because `foo` is the only block in the model and its input is unconnected.
- Including a TLC file for `foogain` eliminates the need for an S-function registration segment for `foogain`. This significantly reduces code size.
- The TLC code inlines the `gain` parameter when the build process is configured to inline parameter values. For example, if the S-function parameter is specified as 2.5 in the S-function dialog box, the TLC `Outputs` function generates

  ```
  rtB.foo = input * 2.5;
  ```
- Use the `%generatefile` directive if your operating system has a filename size restriction and the name of the S-function is `foosfunction` (that exceeds the limit). In this case, you would include the following statement in the system target file (anywhere prior to a reference to this S-function block target file).

  ```
  %generatefile foosfunction "foosfunc.tlc"
  ```

  This statement tells the Target Language Compiler to open `foosfunc.tlc` instead of `foosfunction.tlc`.

Comparison of Noninlined and Inlined Versions of model_reg.h

Inlining a Level 2 S-function significantly reduces the size of the *model*_reg.h code. Model registration functions are lengthy; much of the code has been eliminated in this example. The code below highlights the difference between the noninlined and inlined versions of *model*_reg.h; inlining eliminates this code:

```
/*
 * model_reg.h
 *
*/
/* Normal model initialization code independent of
      S-functions  */

/* child S-Function registration */
  ssSetNumSFunctions(rtS, 1);

  /* register each child */
  {
    static SimStruct childSFunctions[1];
    static SimStruct *childSFunctionPtrs[1];

    (void)memset((char_T *)&childSFunctions[0], 0,
                 sizeof(childSFunctions));
    ssSetSFunctions(rtS, &childSFunctionPtrs[0]);
    {
      int_T i;

      for(i = 0; i < 1; i++) {
        ssSetSFunction(rtS, i, &childSFunctions[i]);
      }
    }

    /* Level2 S-Function Block: untitled/<Root>/S-Function
       (foogain) */
    {
      extern void foogain(SimStruct *rts);
      SimStruct *rts = ssGetSFunction(rtS, 0);

      /* timing info */
      static time_T sfcnPeriod[1];
      static time_T sfcnOffset[1];
      static int_T sfcnTsMap[1];

      {
        int_T i;

        for(i = 0; i < 1; i++) {
          sfcnPeriod[i] = sfcnOffset[i] = 0.0;
        }
      }
      ssSetSampleTimePtr(rts, &sfcnPeriod[0]);
      ssSetOffsetTimePtr(rts, &sfcnOffset[0]);
      ssSetSampleTimeTaskIDPtr(rts, sfcnTsMap);
      ssSetMdlInfoPtr(rts, ssGetMdlInfoPtr(rtS));

      /* inputs */
      {
```

```
    static struct _ssPortInputs inputPortInfo[1];

    _ssSetNumInputPorts(rts, 1);
    ssSetPortInfoForInputs(rts, &inputPortInfo[0]);

    /* port 0 */
    {
      static real_T const *sfcnUPtrs[1];

      sfcnUPtrs[0] = &untitled_RGND;
      ssSetInputPortWidth(rts, 0, 1);
      ssSetInputPortSignalPtrs(rts, 0,
          (InputPtrsType)&sfcnUPtrs[0]);
    }
  }

  /* outputs */
  {
    static struct _ssPortOutputs outputPortInfo[1];
    _ssSetNumOutputPorts(rts, 1);
    ssSetPortInfoForOutputs(rts, &outputPortInfo[0]);
    ssSetOutputPortWidth(rts, 0, 1);
    ssSetOutputPortSignal(rts, 0, &rtB.S_Function);
  }

  /* path info */
  ssSetModelName(rts, "S-Function");
  ssSetPath(rts, "untitled/S-Function");
  ssSetParentSS(rts, rtS);
  ssSetRootSS(rts, ssGetRootSS(rtS));
  ssSetVersion(rts, SIMSTRUCT_VERSION_LEVEL2);

  /* parameters */
  {
    static mxArray const *sfcnParams[1];

    ssSetSFcnParamsCount(rts, 1);
    ssSetSFcnParamsPtr(rts, &sfcnParams[0]);

    ssSetSFcnParam(rts, 0, &rtP.S_Function_P1Size[0]);
  }

  /* registration */
  foogain(rts);

  sfcnInitializeSizes(rts);
  sfcnInitializeSampleTimes(rts);

  /* adjust sample time */
  ssSetSampleTime(rts, 0, 0.2);
  ssSetOffsetTime(rts, 0, 0.0);
  sfcnTsMap[0] = 0;

  /* Update the InputPortReusable and BufferDstPort flags for
    each input port */
  ssSetInputPortReusable(rts, 0, 0);
  ssSetInputPortBufferDstPort(rts, 0, -1);
```

```
      /* Update the OutputPortReusable flag of each output port */
    }
  }
```

A TLC File to Inline S-Function foogain

To avoid unnecessary calls to the S-function and to generate the minimum code required for the S-function, the following TLC file, `foogain.tlc`, is provided as an example.

```
%implements "foogain" "C"

%function Outputs (block, system) Output
  /* %<Type> block: %<Name> */
  %%
  %assign y = LibBlockOutputSignal (0, "", "", 0)
  %assign u = LibBlockInputSignal (0, "", "", 0)
  %assign p = LibBlockParameter (Gain, "", "", 0)
  %<y> = %<u> * %<p>;
%endfunction
```

Managing Block Instance Data with an Eye Toward Code Generation

Instance data is extra data or working memory that is unique to each instance of a block in a Simulink model. This does not include parameter or state data (which is stored in the model parameter and state vectors, respectively), but rather is used to cache intermediate results or derived representations of parameters and modes. One example of instance data is the buffer used by a transport delay block.

Allocating and using memory on an instance-by-instance basis can be done several ways in a Level 2 S-function: via `ssSetUserData`, work vectors (e.g., `ssSetRWorkValue`, `ssSetIWorkValue`), or data-typed work vectors known as `DWork` vectors. For the smallest effort in writing the S-function and block target file and for automatic conformance to both static and `malloc` instance data on targets such as `grt`, use data-typed work vectors when writing S-functions with instance data.

The advantages are twofold. In the first place, writing the S-function is more straightforward, in that memory allocations and frees are handled for you by Simulink. Secondly, the `DWork` vectors are written to the *`model`*`.rtw` file for you automatically, including the `DWork` name, data type, and size. This makes writing the block target file easier, because you do not have to write TLC code for allocating and freeing the `DWork` memory.

Additionally, if you want to bundle groups of `DWork` vectors into structures for passing to functions, you can populate the structure with pointers to `DWork` arrays in both your S-function `mdlStart` function and the block target file's `Start` method, achieving consistency between the S-function and the generated code's handling of data.

Finally, using a `DWork` makes it straightforward to create a specific version of code (data types, scalar vs. vectorized, etc.) for each block instance that matches the implementation in the S-function. Both implementations use `DWork` in the same way so that the inlined code can be used with the Simulink Accelerator software without changes to the C MEX S-function or the block target file.

Using Inlined Code with the Simulink Accelerator Software

By default, the Simulink Accelerator software calls your C MEX S-function as part of an accelerated model simulation. If you prefer to have the accelerator inline your S-function before running the accelerated model, tell the accelerator to use your block target file to inline the S-function with the

`SS_OPTION_USE_TLC_WITH_ACCELERATOR` flag in the call to `ssSetOptions()` in the `mdlInitializeSizes` function of that S-function.

Note that memory and work vector size and usage must be the same for the TLC generated code and the C MEX S-function, or the Simulink Accelerator software cannot execute the inlined code properly. This is because the C MEX S-function is called to initialize the block and its work vectors, calling the `mdlInitializeSizes`, `mdlInitializeConditions`, `mdlCheckParameters`, `mdlProcessParameters`, and `mdlStart` functions. In the case of constant signal propagation, `mdlOutputs` is called from the C MEX S-function during the initialization phase of model execution.

During the time-stepping phase of accelerated model execution, the code generated by the `Output` and `Update` block TLC methods will execute, plus the `Derivatives` and zero-crossing methods if they exist. The `Start` method of the block target file is not used in generating code for an accelerated model.

See Also

Related Examples

- “Write Noninlined S-Function”
- “Write Fully Inlined S-Functions”
- “Write Fully Inlined S-Functions with mdlRTW Routine”

TLC Coding Conventions

These guidelines can help you apply programming style in each target file consistently.

Begin Identifiers with Uppercase Letters

Identifiers in the file begin with an uppercase letter. For example,

```
NumModelInputs              1
NumModelOutputs             2
NumNonVirtBlocksInModel     42
DirectFeedthrough           yes
NumContStates               10
```

Because a `Name` identifier may be promoted into the parent scope, block records that contain a `Name` identifier should start the name with an uppercase letter. For example, a block might contain

```
Block {
   :
   :
RWork                  [4, 0]
   :
NumRWorkDefines        4
RWorkDefine {
    Name               "TimeStampA"
    Width              1
    StartIndex         0
  }
}
```

Because the `Name` identifier within the `RWorkDefine` record is promoted to `PrevT` in its parent scope, it must start with an uppercase letter. The promotion of the `Name` identifier into the parent block scope is currently done for the `Parameter`, `RWorkDefine`, `IWorkDefine`, and `PWorkDefine` block records.

The Target Language Compiler assignment directive (`%assign`) generates a warning if you assign a value to an “unqualified” code generator identifier. For example,

```
%assign TID = 1
```

produces an error because the `TID` identifier is not qualified by `Block`. However, a “qualified” assignment does not generate a warning. For example,

```
%assign Block.TID = 1
```

does not generate a warning because the assignment contains a qualifier. The Target Language Compiler therefore assumes that the programmer is intentionally modifying an identifier.

Begin Global Variable Assignments with Uppercase Letters

Global TLC variable assignments should start with uppercase letters. A global variable is a variable declared in a system target file (`grt.tlc`, `mdlwide.tlc`, `mdlhdr.tlc`, `mdlbody.tlc`, `mdlreg.tlc`, or `mdlparam.tlc`), or within a function that uses the operator. Global assignments have the same scope as code generator variables. An example of a global TLC variable defined in `mdlwide.tlc` is

```
%assign InlineParameters = 1
```

An example of a global reference in a function is

```
%function foo() void
  %assign ::GlobalIdx = ::GlobalIdx + 1
%endfunction
```

Begin Local Variable Assignments with Lowercase Letters

Local TLC variable assignments should start with lowercase letters. A local TLC variable is a variable assigned inside a function. For example,

```
%assign numBlockStates = ContStates[0]
```

Begin Functions Declared in block.tlc Files with Fcn

When you declare a function inside a `block.tlc` file, it should start with `Fcn`. For example,

```
%function FcnMyBlockFunc(...)
```

Note Functions declared inside a system file are global; functions declared inside a block file are local.

Do Not Hard-Code Variables Defined in commonsetup.tlc

Because the code generator tracks use of variables and generates code based on usage, you should use access routines instead of directly using a variable. For example, you should not use the following in your TLC file:

```
x = %<tInf>;
```

You should use

```
x = %<LibRealNonFinite(inf)>;
```

Similarly, instead of using `%<tTID>`, use `%<LibTID()>`. For a complete list of functions, see TLC Function Library Reference in “Target Language Compiler”.

Simulink Coder global variables start with `rt` and Simulink Coder global functions start with `rt_`.

Avoid naming global variables in modules that start with `rt` or `rt_` because they might conflict with Simulink Coder global variables and functions. These TLC variables are declared in `commonsetup.tlc`.

This convention creates consistent variables throughout the target files. For example, the Gain block contains the following `Outputs` function.

```
%% Function: Outputs ========================
%% Abstract:
%%      Y = U * K
%%
%function Outputs(block, system) Output
 /* %<Type> Block: %<Name> */
 %assign rollVars = ["U", "Y", "P"]
 %roll sidIdx = RollRegions, lcv = RollThreshold, block,...
       "Roller", rollVars
  %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
  %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
  %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
  %<y> = %<u> * %<k>;
 %endroll

%endfunction
```

Note c

Note a

Note e

Notes d,f

Note b

Notes about this TLC code

a. The code section for each block begins with a comment specifying the block type and name.

b. Include a blank line immediately after the end of the function to create consistent spacing between blocks in the output code.

c. Try to stay within 80 columns per line for the function banner. You might set up an 80 column comment line at the top of each function. As an example, see `constant.tlc`.

d. For consistency, use the variables `sysIdx` and `blkIdx` for system index and block index, respectively.

e. Use the variable `rollVars` when using the `%roll` construct.

f. When naming loop control variables, use `sigIdx` and `lcv` when looping over `RollRegions` and `xidx` and `xlcv` when looping over the states.

Example: Output function in gain.tlc

```
%roll sigIdx = RollRegions, lcv = RollThreshold, ...
       block, "Roller", rollVars
```

Example: InitializeConditions function in linblock.tlc

```
%roll xidx = [0:nStates-1], xlcv = RollThreshold,...
       block, "Roller", rollVars
```

Conditional Inclusion in Library Files

The Target Language Compiler function library files are conditionally included with guard code so you can reference them multiple times using `%include` without worrying if they have previously been included. Follow this practice for TLC library files that you create.

The convention is to use a variable with the same name as the base filename, uppercase and with underscores attached at both ends. So, a file named `customlib.tlc` should have the variable `_CUSTOMLIB_` guarding it.

As an example, the main Target Language Compiler function library, `funclib.tlc`, contains this TLC code to prevent multiple inclusion:

```
%if EXISTS("_FUNCLIB_") == 0
%assign _FUNCLIB_ = 1
```

```
  .
  .
  .
%endif  %% _FUNCLIB_
```

Code Defensively

As the code your TLC generates could be used in referenced models in unpredictable contexts, do not assume too much about name spaces. For example, when writing TLC code for a block and adding a `typedef`, guard it with `if/def`, as the following example illustrates:

```
%openfile tmpBuff
    #ifndef RESOLUTION_TYPEDEF

    typedef enum { LO_RES, HI_RES } Resolution;
    typedef struct { Resolution res; int8_T value; } Data;

    #define RESOLUTION_TYPEDEF
    #endif /* RESOLUTION_TYPEDEF */
  %closefile tmpBuff

  %<LibCacheTypedefs(tmpBuff)>;
```

See Also

Related Examples

- “Block Target File Methods” on page 21-21

Block Target File Methods

Block Functions Overview

Each block has a target file that determines what code should be generated for the block. The code can vary depending on the exact parameters of the block or the types of connections to it (e.g., wide vs. scalar input).

Within each block target file, block functions specify the code to be output for the block in the model's or subsystem's `start` function, `output` function, `update` function, and so on.

The functions declared inside each of the block target files are called by the system target files. In these tables, `block` refers to a Simulink block name (e.g., `gain` for the Gain block) and `system` refers to the subsystem in which the block resides. The first table lists the two functions that are used for preprocessing and setup. Neither of these functions outputs generated code.

- "BlockInstanceSetup(block, system)" on page 21-21
- "BlockTypeSetup(block, system)" on page 21-22

The following functions generate executable code that the code generator places appropriately:

- "Enable(block, system)" on page 21-23
- "Disable(block, system)" on page 21-23
- "Start(block, system)" on page 21-23
- "InitializeConditions(block, system)" on page 21-24
- "Outputs(block, system)" on page 21-25
- "Update(block, system)" on page 21-26
- "Derivatives(block, system)" on page 21-26
- "Terminate(block, system)" on page 21-26

In object-oriented programming terms, these functions are polymorphic in nature, because each block target file contains the same functions. The Target Language Compiler dynamically determines at run-time which block function to execute depending on the block's type. That is, the system file only specifies that the `Outputs` function, for example, is to be executed. The particular `Outputs` function is determined by the Target Language Compiler depending on the block's type.

To write a block target file, use these polymorphic block functions combined with the Target Language Compiler library functions. For a complete list of the Target Language Compiler library functions, see TLC Function Library Reference on "Target Language Compiler".

BlockInstanceSetup(block, system)

The `BlockInstanceSetup` function executes for the blocks that have this function defined in their target files in a model. For example, if a model includes 10 From Workspace blocks, then the `BlockInstanceSetup` function in `fromwks.tlc` executes 10 times, once for each From Workspace block instance. Use `BlockInstanceSetup` to generate code for each instance of a given block type.

See TLC Function Library Reference on “Target Language Compiler” for available utility processing functions to call from inside this block function. See the file *matlabroot*/rtw/c/tlc/blocks/lookup2d.tlc for an example of the BlockInstanceSetup function.

Syntax

```
    BlockInstanceSetup(block, system) void
    block = Reference to a Simulink block
    system = Reference to a nonvirtual Simulink subsystem
```

This example uses BlockInstanceSetup:

```
%function BlockInstanceSetup(block, system) void
%if (block.InMask == "yes")
    %assign blockName = LibParentMaskBlockName(block)
  %else
    %assign blockName = LibGetFormattedBlockPath(block)
  %endif
  %if (CodeFormat == "Embedded-C")
    %if !(ParamSettings.ColZeroTechnique == "NormalInterp" && ...
          ParamSettings.RowZeroTechnique == "NormalInterp")
      %selectfile STDOUT

Note: Removing repeated zero values from the X and Y axes will
produce more efficient code for block: %<blockName>.  To locate
this block, type

open_system('%<blockName>')

at the MATLAB command prompt.

      %selectfile NULL_FILE
    %endif
  %endif
%endfunction
```

BlockTypeSetup(block, system)

BlockTypeSetup executes once per block type before code generation begins. That is, if 10 Lookup Table blocks exist in the model, the BlockTypeSetup function in look_up.tlc is called only one time. Use this function to perform general work for multiple blocks of a given type.

See TLC Function Library Reference on “Target Language Compiler” for a list of relevant functions to call from inside this block function. See look_up.tlc for an example of the BlockTypeSetup function.

Syntax

```
    BlockTypeSetup(block, system) void
    block = Reference to a Simulink block
    system = Reference to a nonvirtual Simulink subsystem
```

As an example, given the S-function foo, which requires a #define and two function declarations in the header file, you could define:

```
%function BlockTypeSetup(block, system) void
```

```
  %% Place a #define in the model's header file

  %openfile buffer
    #define A2D_CHANNEL 0
  %closefile buffer

  %<LibCacheDefine(buffer)>

  %% Place function prototypes in the model's header file

  %openfile buffer
    void start_a2d(void);
    void reset_a2d(void);
  %closefile buffer

  %<LibCacheFunctionPrototype(buffer)>
%endfunction
```

The remaining block functions execute once for each block in the model.

Enable(block, system)

The code generator creates `Enable` functions for nonvirtual subsystem whenever a Simulink subsystem contains a block with an `Enable` function. Including the `Enable` function in a block's target file places the block's specific enable code in this subsystem `Enable` function. For example:

```
%% Function: Enable ============================================
%% Abstract:
%% Subsystem Enable code is required only for the discrete form
%% of the Sine Block. Setting the Boolean to TRUE causes the
%% Output function to resync its last values of cos(wt) and
%% sin(wt).
%%
%function Enable(block, system) Output
  %if LibIsDiscrete(TID)
    /* %<Type> Block: %<Name> */
    %<LibBlockIWork(SystemEnable, "", "", 0)> = (int_T) TRUE;

  %endif
%endfunction
```

Disable(block, system)

Nonvirtual subsystem `Disable` functions are created whenever a Simulink subsystem contains a block with a `Disable` function. Including the `Disable` function in a block's target file places the block's specific disable code into this subsystem `Disable` function.

Start(block, system)

Include a `Start` function to place code in the `Start` function. The code inside the `Start` function executes once and only once. Typically, you include a `Start` function to execute code once at the

beginning of the simulation (e.g., initialize values in the work vectors) or code that does not need to be re-executed when the subsystem in which it resides is enabled. See `constant.tlc` for an example of the `Start` function.

```
%% Function: Start ============================================
%% Abstract:
%% Set the output to the constant parameter value if the block
%% output is visible in the model's start function scope, i.e.,
%% it is in the global rtB structure.
%%
%function Start(block, system) Output
  %if  LibBlockOutputSignalIsInBlockIO(0)
    /* %<Type> Block: %<Name> */
    %assign rollVars = ["Y", "P"]
    %roll idx = RollRegions, lcv = RollThreshold, block, ...
      "Roller", rollVars
      %assign yr = LibBlockOutputSignal(0,"", lcv, ...
        "%<tRealPart>%<idx>")
      %assign pr = LibBlockParameter(Value, "", lcv, ...
        "%<tRealPart>%<idx>")
      %<yr> = %<pr>;
      %if LibBlockOutputSignalIsComplex(0)
        %assign yi = LibBlockOutputSignal(0, "", lcv, ...
          "%<tImagPart>%<idx>")
        %assign pi = LibBlockParameter(Value, "", lcv, ...
          "%<tImagPart>%<idx>")
        %<yi> = %<pi>;
      %endif
    %endroll
  %endif
%endfunction %% Start
```

InitializeConditions(block, system)

TLC code that is generated from the block's `InitializeConditions` function appears in one of two places. A nonvirtual subsystem contains an `Initialize` function when it is configured to reset states on enable. In this case, the TLC code generated by this block function is placed in the subsystem `Initialize` function, and the `start` function calls this subsystem `Initialize` function. If, however, the Simulink block resides in the root system or in a nonvirtual subsystem that does not require an `Initialize` function, the code generated from this block function is placed directly (inlined) into the `start` function.

There is a subtle difference between the block functions `Start` and `InitializeConditions`. Typically, you include a `Start` function to execute code that does not need to re-execute when the subsystem in which it resides is enabled. You include an `InitializeConditions` function to execute code that must re-execute when the subsystem in which it resides is enabled. For example:

```
%% Function: InitializeConditions ============================
%%
%% Abstract: Invalidate the stored output and input in
%% rwork[1 2*blockWidth] by setting the time stamp stored
%% in rwork[0]) to rtInf.
%%
%function InitializeConditions(block, system) Output
  /* %<Type> Block: %<Name> */
```

```
  %<LibBlockRWork(PrevT, "", "", 0)> = %<LibRealNonFinite(inf)>;
%endfunction
```

Outputs(block, system)

A block should generally include an `Outputs` function. The TLC code generated by a block's `Outputs` function is placed in one of two places. The code is placed directly in the model's `Outputs` function if the block does not reside in a nonvirtual subsystem, and in a subsystem's `Outputs` function if the block resides in a nonvirtual subsystem. For example:

```
%% Function: Outputs ==========================================
%% Abstract:
%%      Y[i] = fabs(U[i]) if U[i] is real or
%%      Y[i] = sqrt(U[i].re^2 + U[i].im^2) if U[i] is complex.
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  %assign inputIsComplex = LibBlockInputSignalIsComplex(0)
  %assign RT_SQUARE = "RT_SQUARE"
  %%
  %assign rollVars = ["U", "Y"]
  %if inputIsComplex
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %%
      %assign ur = LibBlockInputSignal( 0, "", lcv, ...
        "%<tRealPart>%<sigIdx>")
      %assign ui = LibBlockInputSignal( 0, "", lcv, ...
        "%<tImagPart>%<sigIdx>")
      %%
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
      %<y> = sqrt( %<RT_SQUARE>( %<ur> ) + %<RT_SQUARE>( %<ui> ) );
    %endroll
  %else
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %assign u = LibBlockInputSignal (0, "", lcv, sigIdx)
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
      %<y> = fabs(%<u>);
    %endroll
  %endif
  %endfunction
```

Note Zero-crossing reset code is placed in the `Outputs` function.

If you write TLC code to generate inlined code from an S-function, and if the TLC code contains an `Outputs` function, you must modify the TLC code if all of these conditions are true:

- An output port uses or inherits constant sample time. The output port has a constant value.
- The S-function is a multirate S-function or uses port-based sample times.

In this case, the TLC code must generate code for the constant-valued output port by using the function `OutputsForTID` instead of the function `Outputs`. For more information, see “Specifying Constant Sample Time (Inf) for a Port”.

Update(block, system)

Include an `Update` function if the block has code that needs to be updated at each major time step. Code generated from this function is placed in either the model’s or the subsystem’s `Update` function, depending on whether or not the block resides in a nonvirtual subsystem. For example:

```
%% Function: Update ============================================
%% Abstract:
%%      X[i] = U[i]
%%
%function Update(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign rollVars = ["U", "Xd"]
  %roll idx = RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
    %assign u = LibBlockInputSignal(0, "", lcv, idx)
    %assign x = LibBlockDiscreteState("", lcv, idx)
    %<x> = %<u>;
  %endroll
%endfunction %% Update
```

Derivatives(block, system)

Include a `Derivatives` function when generating code to compute the block’s continuous states. Code generated from this function is placed in either the model’s or the subsystem’s `Derivatives` function, depending on whether or not the block resides in a nonvirtual subsystem.

Terminate(block, system)

Include a `Terminate` function to place code in `MdlTerminate`. User-defined S-function target files can use this function to save data, free memory, reset hardware on the target, and so on. See `tofile.tlc` for an example of the `Terminate` function.

See Also

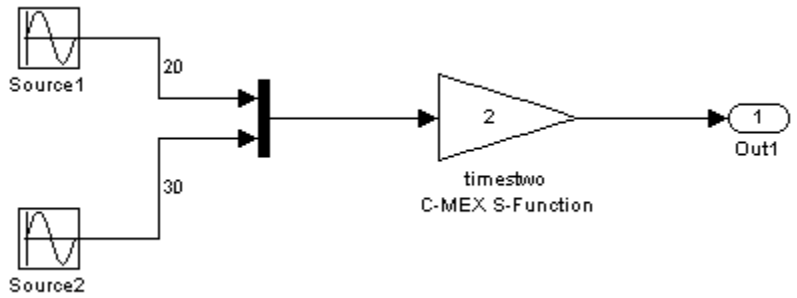
Related Examples

- “Loop Rolling” on page 21-27

Loop Rolling

One of the optimization features of the Target Language Compiler is the intrinsic support for loop rolling. Based on a specified threshold, code generation for looping operations can be unrolled or left as a loop (rolled).

Coupled with loop rolling is the concept of noncontiguous signals. Consider the following model:



The input to the `timestwo` S-function comes from two arrays located at two different memory locations, one for the output of `source1` and one for the output of block `source2`. This is because of an optimization that makes the Mux block virtual, meaning that code is not explicitly generated for the Mux block and thus processor cycles are not spent evaluating it (i.e., it becomes a pure graphical convenience for the block diagram). So this is represented in the *`model`*`.rtw` file in this case as

```
Block {
      Type              "S-Function"
      MaskType              "S-function: timestwo"
      BlockIdx              [0, 0, 2]
      SL_BlockIdx          2
      GrSrc              [0, 1]
      ExprCommentInfo {
    SysIdxList        []
    BlkIdxList        []
    PortIdxList        []
      }
      ExprCommentSrcIdx {
    SysIdx            -1
    BlkIdx            -1
    PortIdx            -1
      }
      Name              "<Root>/timestwo  C-MEX S-Function"
      SLName              "<Root>/timestwo \nC-MEX S-Function"
      Identifier          timestwoCMEXSFunction
      TID              0
      RollRegions          [0:19, 20:49]
      NumDataInputPorts          1
      DataInputPort {
    SignalSrc        [b0@20, b1@30]
    SignalOffset        [0:19, 0:29]
    Width            50
    RollRegions        [0:19, 20:49]
      }
      NumDataOutputPorts      1
```

```
      DataOutputPort {
    SignalSrc        [b2@50]
    SignalOffset        [0:49]
    Width            50
      }
      Connections {
    InputPortContiguous    [no]
    InputPortConnected    [yes]
    OutputPortConnected    [yes]
    OutputPortBeingMerged    [no]
    DirectSrcConn        [no]
    DirectDstConn        [yes]
    DataOutputPort {
      NumConnPoints          1
      ConnPoint {
        SrcSignal            [0, 50]
        DstBlockAndPortEl        [0, 4, 0, 0]
      }
    }
  }
.
.
.
```

From this fragment of the *model*`.rtw` file you can see that the block and input port `RollRegion` entries are not just one number, but two groups of numbers. This denotes two groupings in memory for the input signal. The generated code looks like this:

```
/* S-Function Block: <Root>/timestwo  C-MEX S-Function */
  /* Multiply input by two */
  {
    int_T i1;

    const real_T *u0 = &contig_sample_B.u[0];
    real_T *y0 = contig_sample_B.timestwoCMEXSFunction_m;

    for (i1=0; i1 < 20; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }

    u0 = &contig_sample_B.u_o[0];
    y0 = &contig_sample_B.timestwoCMEXSFunction_m[20];

    for (i1=0; i1 < 30; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }
}
```

Notice that two loops are generated and between them the input signal is redirected from the first base address, `&contig_sample_B.u[0]`, to the second base address of the signals, `&contig_sample_B.u_o[0]`. If you do not want to support this in your S-function or your generated code, you can use

```
ssSetInputPortRequiredContiguous(S, 1);
```

in the `mdlInitializeSizes` function to cause Simulink to implicitly generate code that performs a buffering operation. This option uses both extra memory and CPU cycles at run-time, but might be worth it if your algorithm performance increases enough to offset the overhead of the buffering.

Use the `%roll` directive to generate loops. See also “%roll” on page 19-22 for the reference entry for `%roll`, and “Input Signal Functions” on page 22-7 for a discussion on the behavior of `%roll`.

See Also

Related Examples

- “Target Language Compiler Directives” on page 19-2

TLC Function Library Reference

This chapter provides a set of Target Language Compiler functions that are useful for inlining S-functions. The TLC files contain many other library functions, but you should use only the functions that are documented in these reference pages for development. Undocumented functions can change significantly from release to release. “Obsolete Functions” on page 22-71 includes a table of obsolete functions and their replacements.

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3
- “Input Signal Functions” on page 22-7
- “Output Signal Functions” on page 22-17
- “Parameter Functions” on page 22-23
- “Block State and Work Vector Functions” on page 22-29
- “Block Path and Error Reporting Functions” on page 22-33
- “Code Configuration Functions” on page 22-35
- “Sample Time Functions” on page 22-55
- “Miscellaneous Functions” on page 22-63
- “Advanced Functions” on page 22-73
- “Block Compatibility Functions” on page 22-78

Target Language Compiler Library Functions Overview

There are several library functions that provide access to block inputs, outputs, parameters, sample times, and other information. It is recommended that you use these library functions to access many of the parameter name/parameter values pairs in the block record, as opposed to accessing the parameter name/parameter value pairs directly from your block TLC code.

The library functions simplify block TLC code and provide support for loop rolling, data types, and complex data. The functions also provide a layer to protect against changes that can occur to the contents of the `model.rtw` file.

An exception to using these functions is when you access parameter settings for a block. Parameter settings can be written out using the `mdlRTW` function of a C MEX S-function. They can contain data in the form of strings, scalar values, vectors, and matrices. They can be used to pass fixed values and information that is used to alter the generated code for a block or directly as values in the resulting code of a block. For more information about this exception, see “Exception to Using the Library Functions that Access model.rtw” on page 18-11.

See Also

More About

- “Target Language Compiler Function Conventions” on page 22-3
- “Input Signal Functions” on page 22-7
- “Output Signal Functions” on page 22-17
- “Parameter Functions” on page 22-23
- “Block State and Work Vector Functions” on page 22-29
- “Block Path and Error Reporting Functions” on page 22-33
- “Code Configuration Functions” on page 22-35
- “Sample Time Functions” on page 22-55
- “Miscellaneous Functions” on page 22-63
- “Advanced Functions” on page 22-73

Target Language Compiler Function Conventions

You can find examples using these functions in *matlabroot*/toolbox/simulink/blocks/tlc_c and *matlabroot*/toolbox/simulink/simdemos/simfeatures/tlc_c. The corresponding MEX S-function source code is located in *matlabroot*/simulink/src or *matlabroot*/toolbox/simulink/simdemos/simfeatures/src. MATLAB file S-functions and the MEX-file executables (for example, *sfunction*.mex*) are located in *matlabroot*/toolbox/simulink/blocks or *matlabroot*/toolbox/simulink/simdemos/simfeatures.

Common Function Arguments

Several functions take similar or identical arguments. To simplify the reference pages, some of these arguments are documented in detail here instead of in the reference pages.

Argument	Description
portIdx	Refers to an input or output port index, starting at 0. For example, the first input port of an S-function is 0.
ucv	User control variable. This is an advanced feature that overrides the lcv and sigIdx parameters. When used within an inlined S-function, it should generally be specified as "".
lcv	Loop control variable. This is generally generated by the %roll directive via the second %roll argument (e.g., lcv=RollThreshold) and should be passed directly to the library function. It contains either "", indicating that the current pass through the %roll is being inlined, or it is the name of a loop control variable such as "i", indicating that the current pass through the %roll is being placed in a loop. Outside the %roll directive, this is usually specified as "".

Argument	Description
sigIdx*or*idx	Signal index. Sometimes referred to as the signal element index. When accessing specific elements of an input or output signal directly, the call to the various library routines should have `ucv=""`, `lcv=""`, and `sigIdx` equal to the desired integer signal index starting at 0. For complex signals, `sigIdx` can be an overloaded integer index specifying both whether the real or imaginary part is being accessed and which element. When you access these items inside a `%roll`, use the `sigIdx` generated by the `%roll` directive. Most functions that take a `sigIdx` argument accept it in an overloaded form, where `sigIdx` can be • An integer, e.g., `3`. If the referenced signal is complex, then this refers to the identifier for the complex container. If the referenced signal is not complex, then this refers to the identifier. • An `id-num`, usually of the form (see “Overloading sigIdx” on page 22-4) **a** `"%<tRealPart>%<idx>"` (e.g., `"re3"`). The real part of the signal element. Usually `"%<tRealPart>%<sigIdx>"` when `sigIdx` is generated by the `%roll` directive. **b** `"%<tImagPart>%<idx>"` (e.g., `"im3"`). The imaginary part of the signal element or `""` if the signal is not complex. Usually `"%<tImagPart> %<sigIdx>"` when `sigIdx` is generated by the `%roll` directive. Use the `idx` name when referring to a state or work vector. Functions that accept the three arguments `ucv`, `lcv`, `sigIdx` (or `idx`) are called differently depending upon whether or not they are used within a `%roll` directive. If they are used within a `%roll` directive, `ucv` is generally specified as `""` and, `lcv` and `sigIdx` are the same as those specified in the `%roll` directive. If they are not used within a `%roll` directive, `ucv` and `lcv` are generally specified as `""`, and `sigIdx` specifies the index to access.
`paramIdx`	Parameter index. Sometimes referred to as the parameter element index. The handling of this parameter is very similar to `sigIdx` above: it can be `#`, `re#`, or `im#`.
`stateIdx`	State index. Sometimes referred to as the state vector element index. It must evaluate to an integer where the first element starts at `0`.

Overloading sigIdx

The signal index (`sigIdx` sometimes written as `idx`) can be overloaded when passed to most library functions. Suppose you are interested in element 3 of a signal, and `ucv=""`, `lcv=""`. The following table shows

- Values of `sigIdx`
- Whether the signal being referenced is complex
- What the function that uses `sigIdx` returns
- An example of a returned variable
- Data type of the returned variable

Note that “container” in the following table refers to the object that encapsulates both the real and imaginary parts of the number, e.g., `creal_T`, defined in `tmwtypes.h`.

sigIdx	Complex	Function Returns	Example	Data Type
`"re3"`	Yes	Real part of element 3	`u0[2].re`	`real_T`
`"im3"`	Yes	Imaginary part of element 3	`u0[2].im`	`real_T`
`"3"`	Yes	Complex container of element 3	`u0[2]`	`creal_T`
`3`	Yes	Complex container of element 3	`u0[2]`	`creal_T`
`"re3"`	No	Element 3	`u0[2]`	`real_T`
`"im3"`	No	`""`	N/A	N/A
`"3"`	No	Element 3	`u0[2]`	`real_T`
`3`	No	Element 3	`u0[2]`	`real_T`

Now suppose the following:

1. You are interested in element 3 of a signal.
2. (`ucv = "i"` AND `lcv == ""`) OR (`ucv = ""` AND `lcv = "i"`).

The following table shows values of `idx`, whether the signal is complex, and what the function that uses `idx` returns.

sigIdx	Complex	Function Returns
`"re3"`	Yes	Real part of element `i`
`"im3"`	Yes	Imaginary part of element `ii`
`"3"`	Yes	Complex container of element `i`
`3`	Yes	Complex container of element `i`
`"re3"`	No	Element `i`
`"im3"`	No	`""`
`"3"`	No	Element `i`
`3`	No	Element `i`

Notes

- The vector index is added only for wide signals.
- If `ucv` is not an empty string (`""`), then `ucv` is used instead of `sigIdx` in the above examples and both `lcv` and `sigIdx` are ignored.
- If `ucv` is empty but `lcv` is not empty, then the function returns `"&y%<portIdx>[%<lcv>]"` and `sigIdx` is ignored.
- It is assumed that the roller has declared and initialized the variables accessed inside the roller. The variables accessed inside the roller should be specified using `rollVars` as the argument to the `%roll` directive.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2

Input Signal Functions

LibBlockInputHasSymbolicWidth(portIdx)

Returns 1 (true) if the specified block input port index has symbolic dimensions. This function take the port index as an input argument.

See `LibBlockInputHasSymbolicWidth` in `blkio_api.tlc`.

LibBlockInputPortIndexMode(block, pidx)

Purpose

Determines the index mode of a block's input port.

Arguments

`block` — Block record

`pidx` — Port index

Returns

`""` for a nonindex port, and `"Zero-based"` or `"One-based"` otherwise.

Description

If an input port of a `block` is set as an index port and its indexing base is marked as zero-based or one-based, this information is written into the *`model`*`.rtw` file. `LibBlockInputPortIndexMode` queries the indexing base to branch to different code according to what the input port indexing base is.

Example

```
%if LibBlockInputPortIndexMode(block, pidx) == "Zero-based"
  ...
%elseif LibBlockInputPortIndexMode(block, pidx) == "One-based"
  ...
%else
  ...
%endif
```

See `LibBlockInputPortIndexMode` in `blkiolib.tlc`.

LibBlockInputPortIsContinuousQuantity(portIdx)

Returns whether the input port accepts a co-simulation signal. This function takes the port index as an input argument.

See `LibBlockInputPortIsContinuousQuantity` in `blkio_api.tlc`.

LibBlockInputSignal(portIdx, ucv, lcv, sigIdx)

Based on the input port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and where this input signal is coming from, `LibBlockInputSignal` returns the reference to a block input signal.

The returned string value is a valid `rvalue` (right-side value) for an expression. The block input signal can come from another block, a state vector, or an external input, or it can be a literal constant (e.g., 5.0).

Note Do not use `LibBlockInputSignal` to access the address of an input signal.

Because the returned value can be a literal constant, you should not use `LibBlockInputSignal` to access the address of an input signal. To access the address of an input signal, use `LibBlockInputSignalAddr`. Accessing the address of the signal via `LibBlockInputSignal` can result in a reference to a literal constant (e.g., 5.0).

For example, the following would *not* work.

```
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
x = &%<u>;
```

If `%<u>` refers to an invariant signal with a value of `4.95`, the statement (after being processed by the preprocessor) would be generated as

```
x = &4.95;
```

or, if the input signal sources to ground, the statement could come out as

```
x = &0.0;
```

Neither of these would compile.

Avoid such situations by using `LibBlockInputSignalAddr`.

```
%assign uAddr = LibBlockInputSignalAddr(0, "", lcv, sigIdx)
x = %<uAddr>;
```

The code generator tracks signals and parameters accessed by their addresses and declares them in addressable memory.

Input Arguments

The following table summarizes the input arguments to `LibBlockInputSignal`.

LibBlockInputSignal Arguments

Argument	Description
`portIdx`	Integer specifying the input port index (zero-based). **Note:** For certain built-in blocks, `portIdx` can be a string identifying the port (such as `"enable"` or `"trigger"`).
`ucv`	User control variable. Must be a string, either an indexing expression or `""`.
`lcv`	Loop control variable. Must be a string, either an indexing expression or `""`.

`sigIdx`	Either an integer literal or a string of the form
	
	`%<tRealPart>Integer`
	`%<tImagPart>Integer`
	
	For example, the following signifies the real part of the signal and the imaginary part of the signal starting at 5:
	
	`"%<tRealPart>5"`
	`"%<tImagPart>5"`

General Usage

Uses of `LibBlockInputSignal` fall into the categories described below.

Direct indexing

If `ucv == ""` and `lcv == ""`, `LibBlockInputSignal` returns an indexing expression for the element specified by `sigIdx`.

Loop rolling/unrolling

In this case, `lcv` and `sigIdx` are generated by the `%roll` directive, and `ucv` must be `""`. A nonempty value for `lcv` is allowed only when generated by the `%roll` directive and when using the Roller TLC file (or a user supplied Roller TLC file that conforms to the same variable/signal offset handling). In addition, calls to `LibBlockInputSignal` with `lcv` should occur only when `"U"` or a specific input port (e.g., `"u0"`) is passed to the `%roll` directive via the roll variables argument.

The following example shows a single input/single output port S-function.

```
%assign rollVars  = ["U", "Y", "P"]
%roll sigIdx=RollRegions, lcv=RollThreshold, block, ...
    "Roller", rollVars
  %assign u = LibBlockInputSignal( 0, "", lcv, sigIdx)
  %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
  %assign p = LibBlockParameter(   0, "", lcv, sigIdx)
  %<y> = %<p> * %<u>;
%endroll
```

With the `%roll` directive, `sigIdx` is the starting index of the current roll region and `lcv` is `""` or an indexing variable. The following are examples of valid values:

```
LibBlockInputSignal(0, "", lcv, sigIdx)    rtB.blockname[0]

LibBlockInputSignal(0, "", lcv, sigIdx)    u[i]
```

In the first example, `LibBlockInputSignal` returns `rtB.blockname[2]` when the input port is connected to the output of another block, and

- The loop control variable (`lcv`) generated by the `%roll` directive is empty, indicating that the current roll region is below the roll threshold, and `sigIdx` is `0`.
- The width of the input port is `1`, indicating that this port is being scalar expanded.

  If `sigIdx` is nonzero, then `rtB.blockname[sigIdx]` is returned. For example, if `sigIdx` is `3`, then `rtB.blockname[3]` is returned.

In the second example, `LibBlockInputSignal` returns `u[i]` when the current roll region is above the roll threshold and the input port width is nonscalar (wide). In this case, the Roller TLC file sets up a local variable, `u`, to point to the input signal, and the code in the current `%roll` directive is placed within a `for` loop.

For another example, consider a block with multiple input ports where each port has a width greater than or equal to 1 and at least one port has width equal to 1. The following code sets the output signal to the sum of the squares of the input signals.

```
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = 0;

%assign rollVars = ["U"]
%foreach port = block.NumDataInputPorts - 1
  %roll sigIdx=RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
  %assign u = LibBlockInputSignal(port, "", lcv, sigIdx)
  %<y> += %<u> * %<u>;
  %endroll
%endforeach
```

Because the first parameter of `LibBlockInputSignal` is `0` indexed, you must index the `foreach` loop to start from `0` and end at `NumDataInputPorts-1`.

User Control Variable (ucv) Handling

This is an advanced mode and generally not required by S-function authors.

If `ucv != ""`, `LibBlockInputSignal` returns an `rvalue` for the input signal using the user control variable indexing expression. The control variable indexing expression has the following form:

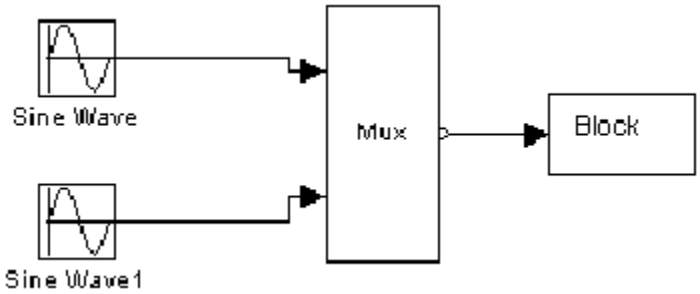
```
rvalue_id[%<ucv>]%<optional_real_or_imag_part>
```

To obtain `rvalue_id`, look at the integer part of `sigIdx`. You must specify `sigIdx` because the input to this block can be discontinuous, meaning that the input can come from several different memory areas (signal sources) and `sigIdx` is used to identify the area of interest for the `ucv`. You can also use `sigIdx` to determine whether the real or imaginary part of a signal is to be accessed.

You can obtain `optional_real_or_imag_part` from the string part of `sigIdx` (i.e., `"re"`, or `"im"`, or `""`).

Note that the value for `lcv` is ignored and `sigIdx` must point to the same element in the input signal to which the `ucv` initially points.

The handling of `ucv` with `LibBlockInputSignal` requires care. Consider a discontinuous input signal feeding an input port as in the following block diagram:



To use `ucv` in a robust manner, you must use the `%roll` directive with a roll threshold of `1` and a Roller TLC file that does not have loop header/trailer setup for this input signal. In addition, you need to use `ROLL_ITERATIONS` to determine the width of the current roll region, as in the following TLC code:

```
{
int i;

%assign rollVars  = [""]
%assign threshold = 1
  %roll sigIdx=RollRegions, lcv=threshold, block, ...
    "FlatRoller", rollVars
  %assign u = LibBlockInputSignal( 0, "i", "", sigIdx)
  %assign y = LibBlockOutputSignal(0, "i+%<sigIdx>", "", sigIdx)
  %assign p = LibBlockParameter( 0, "i+%<sigIdx>", "", sigIdx)
  for (i = 0; i < %<ROLL_ITERATIONS()>; i++) {
    %<y> = %<p> * %<u>;
  }
%endroll
}
```

Note that the `FlatRoller` does not have loop header/trailer setup (`rollVars` is ignored). Its purpose is to walk the `RollRegions` of the block. Alternatively, you can force a contiguous input signal to your block by specifying

```
ssSetInputPortRequiredContiguous(S, port, TRUE)
```

in your S-function.

In this case, the TLC code simplifies to

```
{
%assign u = LibBlockInputSignal( 0, "i", "", 0)
%assign y = LibBlockOutputSignal(0, "i", "", 0)
%assign p = LibBlockParameter(   0, "i", "", 0)

for (i = 0; i < %<LibBlockInputSignalWidth(0)>; i++) {
  %<y> = %<p> * %<u>;
  }
}
```

If you create your own roller and the indexing does not conform to the way the Roller TLC file provided by MathWorks operates, then must to use `ucv` instead of `lcv`.

Handling Input Arguments: ucv, lcv, and sigIdx

Consider the following cases:

Function (Case 1, 2, 3,4)	Example Return Value
`LibBlockInputSignal(0, "i", "", sigIdx)`	`rtB.blockname[i]`
`LibBlockInputSignal(0, "i", "", sigIdx)`	`rtU.signame[i]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`u0[i1]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`rtB.blockname[0]`

The value returned depends on what the input signal is connected to in the block diagram and how the function is invoked (e.g., in a `%roll` or directly). In the above example,

- Cases 1 and 2 occur when an explicit call is made with the `ucv` set to `"i"`.

  Case 1 occurs when `sigIdx` points to the block I/O vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to be starting at offset `5`, then you should specify `sigIdx == 5`.

  Case 2 occurs when `sigIdx` points to the external input vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to start at offset `20`, then you should specify `sigIdx == 20`.
- Cases 3 and 4 receive the same arguments, `lcv` and `sigIdx`; however, they produce different return values.

  Case 3 occurs when `LibBlockInputSignal` is called within a `%roll` directive and the current roll region is being rolled (`lcv != ""`).

  Case 4 occurs when `LibBlockInputSignal` is called within a `%roll` directive and the current roll region is not being rolled (`lcv == ""`).

When called within a`%roll` directive, `LibBlockInputSignal` looks at `ucv`, `lcv`, and `sigIdx`, the current roll region, and the current roll threshold to determine the return value. The variable `ucv` has highest precedence, `lcv` has the next highest precedence, and `sigIdx` has the lowest precedence. That is, if `ucv` is specified, it is used (thus, when called in a `%roll` directive it is usually `""`). If `ucv` is not specified, and if `lcv` and `sigIdx` are specified, the returned value depends on whether or not the current roll region is being placed in a `for` loop or being expanded. If the roll region is being placed in a loop, then `lcv` is used; otherwise, `sigIdx` is used.

A direct call to `LibBlockInputSignal` (inside or outside a `%roll` directive) uses `sigIdx` when `ucv` and `lcv` are specified as `""`.

For an example of `LibBlockInputSignal`, see `sfun_multiport.tlc`.

See also `blkiolib.tlc`.

LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns a string that provides the memory address of the specified block input port signal.

When you need an input signal address, you must use `LibBlockInputSignalAddr` instead of appending an “&” to the string returned by `LibBlockInputSignal`. For example,

`LibBlockInputSignal` can return a literal constant, such as `5` (i.e., an invariant input signal). The code generator tracks when `LibBlockInputSignalAddr` is called on an invariant signal and declares the signal as `const` data (which is addressable), instead of being placed as a literal constant in the generated code (which is not addressable).

Note that the last input argument, `sigIdx`, is not overloaded, which it is in `LibBlockInputSignal`. Hence, if the input signal is complex, the address of the complex container is returned.

Example

To get the address of a wide input signal and pass it to a user function for processing, you could use

```
%assign uAddr = LibBlockInputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn(%<uAddr>);
```

See `LibBlockInputSignalAddr` in `blkiolib.tlc`.

LibBlockInputSignalAliasedThruDataTypeName(portIdx, reim)

Returns the name of the aliased thru data type (e.g., `int_T, ... creal_T`) corresponding to the specified block input port. Specify the `reim` argument as `""` (empty) if you want the complete signal type name.

For example, if `reim == ""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,"")
```

Specify `reim` as `tRealPart` if you want the raw element type name. For example, if `reim == tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See `LibBlockInputSignalAliasedThruDataTypeName` in `blkiolib.tlc`.

LibBlockInputSignalAllowScalarExpandedExpr(block,portIdx)

Allow the input signal to be an expression, even when an output signal is wide. This function takes the block record and the port index as input arguments. Call this function from within the `BlockInstanceSetup` function.

See `LibBlockInputSignalAllowScalarExpandedExpr` in `blkio_api.tlc`.

LibBlockInputSignalASCIIEscapedUnitExpr(portIdx)

Returns the escaped name of the units corresponding to the specified block input port. This function takes the port index as an input argument.

See `LibBlockInputSignalASCIIEscapedUnitExpr` in `blkio_api.tlc`.

LibBlockInputSignalConnected(portIdx)

Returns 1 if the specified input port is connected to a block other than the Ground block and 0 otherwise.

See `LibBlockInputSignalConnected` in`blkio_api.tlc` .

LibBlockInputSignalDataTypeId(portIdx)

Returns the numeric identifier (`id`) corresponding to the data type of the specified block input port.

If the input port signal is complex, `LibBlockInputSignalDataTypeId` returns the data type of the real part (or the imaginary part) of the signal.

See `LibBlockInputSignalDataTypeId` in `blkiolib.tlc`.

LibBlockInputSignalDataTypeName(portIdx, reim)

Returns the name of the data type (e.g., `int_T`, ... `creal_T`) corresponding to the specified block input port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim==""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,"")
```

Specify the `reim` argument as `tRealPart` if you want the raw element type name. For example, if `reim==tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,tRealPart)
```

See `LibBlockInputSignalDataTypeName` in `blkiolib.tlc`.

LibBlockInputSignalDimensions(portIdx)

Returns the dimensions vector of the specified block input port, e.g., `[2,3]`.

See `LibBlockInputSignalDimensions` in `blkio_api.tlc`.

LibBlockInputSignalIsComplex(portIdx)

Returns 1 if the specified block input port is complex, 0 otherwise.

See `LibBlockInputSignalIsComplex` in `blkio_api.tlc`.

LibBlockInputSignalIsExpr(portIdx)

Returns 1 (true) if the input signal is an expression (versus a variable), and 0 (false) otherwise. This function takes the port index as an input argument.

See `LibBlockInputSignalIsExpr` in `blkio_api.tlc`.

LibBlockInputSignalIsFrameData(portIdx)

Returns 1 if the specified block input port is frame based, 0 otherwise.

See `LibBlockInputSignalIsFrameData` in `blkio_api.tlc`.

LibBlockInputSignalIsTrivialExpr(portIdx)

Returns 1 (true) if the input signal is a trivial expression (versus a variable), and 0 (false) otherwise. This function take the port index as an input argument.

See `LibBlockInputSignalIsTrivialExpr` in `blkio_api.tlc`.

LibBlockInputSignalLocalSampleTimeIndex(portIdx)

Returns the local sample time index corresponding to the specified block input port.

See `LibBlockInputSignalLocalSampleTimeIndex` in `blkiolib.tlc`.

LibBlockInputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block input port.

See `LibBlockInputSignalNumDimensions` in `blkio_api.tlc`.

LibBlockInputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block input port.

See `LibBlockInputSignalOffsetTime` in `blkiolib.tlc`.

LibBlockInputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block input port.

See `LibBlockInputSignalSampleTime` in `blkiolib.tlc`.

LibBlockInputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block input port.

See `LibBlockInputSignalSampleTimeIndex` in `blkiolib.tlc`.

LibBlockInputSignalSymbolicDimensions(portIdx)

Returns the number of dimensions of the specified block input port.

See `LibBlockInputSignalSymbolicDimensions(portIdx)` in `blkiolib.tlc`.

LibBlockInputSignalSymbolicWidth(portIdx)

Returns the symbolic width of the specified block input port.

See `LibBlockInputSignalSymbolicWidth(portIdx)` in `blkiolib.tlc`.

LibBlockInputSignalWidth(portIdx)

Returns the width of the specified block input port index.

See `LibBlockInputSignalWidth` in `blkio_api.tlc`.

LibBlockInputSignalUnitExpr(portIdx)

Returns the name of the units corresponding to the specified block input port. This function takes the port index as an input argument.

See `LibBlockInputSignalUnitExpr` in `blkio_api.tlc`.

LibBlockInputSignalUnitId(portIdx)

Returns the numeric identifier (id) corresponding to the units of the specified block input port. This function takes the port index as an input argument.

See `LibBlockInputSignalUnitId` in `blkio_api.tlc`.

See also `LibBlockInputSignalUnitExp` and `LibBlockOutputSignalUnitId` in `blkio_api.tlc`.

LibBlockNumInputPorts(block)

Returns the number of data input ports of a block (excludes control ports).

See `LibBlockNumInputPorts` in `blocklib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Output Signal Functions

LibBlockAssignOutputSignal(portIdx, ucv, lcv, sigIdx, rhs)

Based on the output port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the output signal destination, `LibBlockAssignOutputSignal` assigns a block's output to a specified right hand side value, (`rhs`).

See `LibBlockAssignOutputSignal` in `customstoragelib.tlc`.

LibBlockGetNumFcnCallOutputs(block)

Returns the number of function call outputs for the specified block. This function takes the block record as an input argument.

See `LibBlockGetNumFcnCallOutputs` in `block_api.tlc`.

LibBlockGetNumFcnCallOutputsDsts(block)

Returns the number of function call output destinations for the specified block. This function takes the block record as an input argument.

See `LibBlockGetNumFcnCallOutputDsts` in `block_api.tlc`.

LibBlockNumOutputPorts(block)

Returns the number of data output ports of a block (excludes control and state ports).

See `LibBlockNumOutputPorts` in `blocklib.tlc`.

LibBlockOutputHasSymbolicWidth(portIdx)

Returns 1 (true) if the specified block output port index has symbolic dimensions, and 0 (false) otherwise. This function takes the port index as an input argument.

See `LibBlockOutputHasSymbolicWidth` in `blkio_api.tlc`.

LibBlockOutputPortIndexMode(block, pidx)

Purpose

Determines the index mode of a block's output port.

Description

If a block's output port is set as an index port and its indexing base is marked as zero-based or one-based, this information is written into the *`model`*`.rtw` file. `LibBlockOutputPortIndexMode` queries the indexing base to branch to different code according to what the output port indexing base is.

Example

```
%if LibBlockOutputPortIndexMode(block, idx) == "Zero-based"
  ...
%elseif LibBlockOutputPortIndexMode(block, idx) == "One-based"
  ...
%else
  ...
%endif
```

Arguments

`block` — Block record

`pidx` — Port index

Returns

`""` for a nonindex port, and `"Zero-based"` or `"One-based"` otherwise.

See `LibBlockOutputPortIndexMode` in `blkiolib.tlc`.

LibBlockOutputPortIsContinuousQuantity(portIdx)

Returns whether the output port represents a co-simulation signal. This function takes the port index as an input argument.

See `LibBlockOutputPortIsContinuousQuantity` in `blkio_api.tlc`.

LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx)

Based on the output port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the output signal destination, `LibBlockOutputSignal` returns a reference to a block output signal.

The returned value is a valid `lvalue` (left-side value) for an expression. The block output destination can be a location in the block I/O vector (another block's input), the state vector, or an external output.

Note Do not use `LibBlockOutputSignal` to access the address of an output signal.

The code generator tracks when a variable (e.g., a signal or parameter) is accessed by its address. To access the address of an output signal, use `LibBlockOutputSignalAddr` as in the following example:

```
%assign yAddr = LibBlockOutputSignalAddr(0, "", lcv, sigIdx)
x = %<yAddr>;
```

See `LibBlockOutputSignal` in `blkiolib.tlc`.

LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns a string that provides the memory address of the specified block output port signal.

When an output signal address is required, you must use `LibBlockOutputSignalAddr` instead of taking the address that is returned by `LibBlockOutputSignal`. For example, `LibBlockOutputSignal` can return a literal constant, such as 5 (i.e., an invariant output signal). When `LibBlockOutputSignalAddr` is called on an invariant signal, the signal is declared as a `const` instead of being placed as a literal constant in the generated code.

Note that unlike `LibBlockOutputSignal`, the last argument, `sigIdx`, is not overloaded. Hence, if the output signal is complex, the address of the complex container is returned.

Example

To get the address of a wide output signal and pass it to a user function for processing, you could use

```
%assign u = LibBlockOutputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn (%<u>);
```

See `LibBlockOutputSignalAddr` in `blkiolib.tlc`.

LibBlockOutputSignalAliasedThruDataTypeName(portIdx, reim)

Returns the type name string (e.g., `int_T, ... creal_T`) of the aliased data type corresponding to the specified block output port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim == ""` and the first output port is real and complex, the data type placed in `dtname` is `creal_T`:

```
%assign dtname = LibBlockOutputSignalAliasedThroughDataTypeName(0x,"")
```

Specify the reim argument as `tRealPart` if you want the raw element type name. For example, if `reim == tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalAliasedThroughDataTypeName(0,tRealPart)
```

See `LibBlockOutputSignalAliasedThruDataTypeName` in `blkiolib.tlc`.

LibBlockOutputSignalBeingMerged(portIdx)

Returns whether the specified output port is connected to a Merge block.

See `LibBlockOutputSignalBeingMerged` in `blkio_api.tlc`.

LibBlockOutputSignalConnected(portIdx)

Returns 1 if the specified output port is connected to a block other than the Ground block and 0 otherwise.

See `LibBlockOutputSignalConnected` in `blkio_api.tlc`.

LibBlockOutputSignalDataTypeId(portIdx)

Returns the numeric ID corresponding to the data type of the specified block output port.

If the output port signal is complex, `LibBlockOutputSignalDataTypeId` returns the data type of the real (or the imaginary) part of the signal.

See `LibBlockOutputSignalDataTypeId` in `blkiolib.tlc`.

LibBlockOutputSignalDataTypeName(portIdx, reim)

Returns the type name string (e.g., `int_T`, ... `creal_T`) of the data type corresponding to the specified block output port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim==""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0x,"")
```

Specify the `reim` argument as `tRealPart` if you want the raw element type name. For example, if `reim==tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See `LibBlockOutputSignalDataTypeName` in `blkiolib.tlc`.

LibBlockOutputSignalDimensions(portIdx)

Returns the dimensions of the specified block output port.

See `LibBlockOutputSignalDimensions` in `blkio_api.tlc`.

LibBlockOutputSignalIsComplex(portIdx)

Returns 1 if the specified block output port is complex, 0 otherwise.

See `LibBlockOutputSignalIsComplex` in `blkio_api.tlc`.

LibBlockOutputSignalIsConstExpr(portIdx)

Returns 1 (true) if the output signal is a constant expression, and 0 (false) otherwise. Note that a constant expression is constant even if a block output function calls it. This function takes the port index as an input argument.

See `LibBlockOutputSignalIsConstExpr` in `blkio_api.tlc`.

LibBlockOutputSignalIsExpr(portIdx)

Returns 1 (true) if the output signal is an expression, and 0 (false) otherwise.

See `LibBlockOutputSignalIsExpr` in `blkio_api.tlc`.

LibBlockOutputSignalIsExprInOutputs(portIdx)

Returns 1 (true) if the output signal is an expression in the outputs function, and 0 (false) otherwise. You can use this function as a low-level utility to get access to what the value will be when generating the block's output code. Use this function in the `BlockInstanceSetup` function. This function takes the port index as an input argument.

See `LibBlockOutputSignalIsExprInOutputs` in `blkio_api.tlc`.

LibBlockOutputSignalIsFrameData(portIdx)

Returns 1 if the specified block output port is frame based, 0 otherwise.

See `LibBlockOutputSignalIsFrameData` in `blkio_api.tlc`.

LibBlockOutputSignalIsNonConstExpr(portIdx)

Returns 1 (true) if the output signal is an non-constant expression, and 0 (false) otherwise. This function takes the port index as an input argument.

See `LibBlockOutputSignalIsNonConstExpr` in `blkio_api.tlc`.

LibBlockOutputSignalIsTrivialExpr(portIdx)

Returns 1 (true) if the output signal is a trivial expression, and 0 (false) otherwise. This function takes the port index as an input argument.

See `LibBlockOutputSignalIsTrivialExpr` in `blkio_api.tlc`.

LibBlockOutputSignalLocalSampleTimeIndex(portIdx)

Returns the local sample time index corresponding to the specified block output port.

See `LibBlockOutputSignalLocalSampleTimeIndex` in `blkio_api.tlc`.

LibBlockOutputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block output port.

See `LibBlockOutputSignalNumDimensions` in `blkio_api.tlc`.

LibBlockOutputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block output port.

See `LibBlockOutputSignalOffsetTime` in `blkiolib.tlc`.

LibBlockOutputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block output port.

See `LibBlockOutputSignalSampleTime` in `blkiolib.tlc`.

LibBlockOutputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block output port.

See `LibBlockOutputSignalSampleTimeIndex` in `blkiolib.tlc`.

LibBlockOutputSignalSymbolicDimensions(portIdx)

Returns the symbolic dimensions of specified block output port.

See `LibBlockOutputSignalSymbolicDimensions` in `blkiolib.tlc`.

LibBlockOutputSignalSymbolicWidth(portIdx)

Returns the symbolic width of specified block output port.

See `LibBlockOutputSignalSymbolicWidth` in `blkiolib.tlc`.

LibBlockOutputSignalWidth(portIdx)

Returns the width of the specified block output port.

See `LibBlockOutputSignalWidth` in `blkio_api.tlc`.

LibBlockOutputSignalUnitExpr(portIdx)

Returns the type name string of the units corresponding to the specified output port. This function takes the port index as an input argument.

See `LibBlockOutputSignalUnitExpr` in `blkio_api.tlc`.

LibBlockOutputSignalUnitId(portIdx)

Returns the numeric ID corresponding to the units of the specified block output port. This function takes the port index as an input argument.

See `LibBlockOutputSignalUnitId` in `blkio_api.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Parameter Functions

LibBlockMatrixParameter

`LibBlockMatrixParameter(param, rucv, rlcv, ridx, cucv, clcv, cidx)` returns a matrix parameter for a block, given the row and column user control variables (`rucv`, `cucv`), loop control variables (`rlcv`, `clcv`), and indices (`ridx`, `cidx`). Generally, blocks should use `LibBlockParameter`. If you have a matrix parameter, you should write it as a column-major vector and access it via `LibBlockParameter`.

Note Loop rolling is currently not supported, and will generate an error if requested (i.e., if either `rlcv` or `clcv` is not equal to `""`).

The row and column index arguments are similar to the arguments for `LibBlockParameter`. The column index (`cidx`) is overloaded to handle complex numbers.

See `LibBlockMatrixParameter` in `paramlib.tlc`.

LibBlockMatrixParameterAddr

`LibBlockMatrixParameterAddr(param, rucv, rlcv, ridx, cucv, clcv, cidx)` returns the address of a matrix parameter.

Note `LibBlockMatrixParameterAddr` returns the address of a matrix parameter. Loop rolling is not supported (i.e., `rlcv` and `clcv` should both be an empty string).

See `LibBlockMatrixParameterAddr` in `paramlib.tlc`.

LibBlockMatrixParameterBaseAddr

`LibBlockMatrixParameterBaseAddr(param)` returns the base address of a matrix parameter.

See `LibBlockMatrixParameterBaseAddr` in `paramlib.tlc`.

LibBlockParamSetting

`LibBlockParamSetting(bType, psType)` returns the string of a specified parameter setting for a specified block type. If you pass an empty block type into this function, the parameter setting will be assumed to be in the `ParamSettings` record of the block. If a nonempty block type is passed into the function, the parameter settings will be assumed to be in the `%<Btype>ParamSettings` record of that block.

See `LibBlockParamSetting` in `paramlib.tlc`.

LibBlockParameter

Based on the parameter reference (`param`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the state of parameter inlining, `LibBlockParameter(param,`

`ucv, lcv, sigIdx)` returns a reference to a block parameter. The returned value is a valid `rvalue` (right-side value for an expression). For example,

Case	Function Call	Can Produce
1	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname[i]`
2	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname`
3	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain[i]`
4	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain`
5	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`4.55`
6	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.re`
7	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.im`

To illustrate the basic workings of `LibBlockParameter`, assume a noncomplex vector signal where `Gain[0]=4.55`:

```
LibBlockParameter(Gain, "", "i", 0)
```

Case	Rolling	Inline Parameter	Type	Result	Required in Memory
1	0	Yes	Scalar	4.55	No
2	1	Yes	Scalar	4.55	No
3	0	Yes	Vector	4.55	No
4	1	Yes	Vector	`p_Gain[i]`	Yes
5	0	No	Scalar	`rtP.blk.Gain`	No
6	0	No	Scalar	`rtP.blk.Gain`	No
7	0	No	Vector	`rtP.blk.prm[0]`	No
8	0	No	Vector	`p.Gain[i]`	Yes

Note Case 4. Even though Inline Parameter is Yes, the parameter must be placed in memory (RAM), because it is accessed inside a `for` loop.

Note `LibBlockParameter` also supports expressions when used with inlined parameters and parameter tuning.

For example, if the parameter field had the MATLAB expression `'2*a'`, `LibBlockParameter` would return the C expression `'(2*a)'`. The list of functions supported by `LibBlockParameter` is determined by the functions `FcnConvertNodeToExpr` and `FcnConvertIdToFcn`. To enhance functionality, augment or update either of these functions.

Note that certain types of expressions are not supported, such as `x*y` where *both* `x` and `y` are nonscalar expressions.

See the documentation about tunable parameters for more details on the exact functions and syntax that are supported.

Warning

Do not use `LibBlockParameter` to access the address of a parameter, or you may might erroneously reference a number (i.e., `&4.55`) when the parameter is inlined. You can avoid this situation by using `LibBlockParameterAddr`.

See `LibBlockParameter` in `paramlib.tlc`.

LibBlockParameterAddr

`LibBlockParameterAddr(param, ucv, lcv, idx)` returns the address of a block parameter.

Using `LibBlockParameterAddr` to access a parameter when the global `InlineParams` variable is equal to 1 will cause the variable to be declared `const` in RAM instead of being inlined.

Accessing the address of an expression when the expression has multiple tunable/rolled variables in it will result in an error.

See `LibBlockParameterAddr` in `paramlib.tlc`.

LibBlockParameterBaseAddr

`LibBlockParameterBaseAddr(param)` returns the base address of a block parameter.

Using `LibBlockParameterBaseAddr` to access a parameter when the global `InlineParams` variable is equal to one will cause the variable to be declared `const` in RAM instead of being inlined.

Accessing the address of an expression when the expression has multiple tunable/rolled variables in it will result in an error.

See `LibBlockParameterBaseAddr` in `paramlib.tlc`.

LibBlockParameterDataTypeId

`LibBlockParameterDataTypeId(param)` returns the numeric ID corresponding to the data type of the specified block parameter.

See `LibBlockParameterDataTypeId` in `paramlib.tlc`.

LibBlockParameterDataTypeName

`LibBlockParameterDataTypeName(param, reim)` returns the name of the data type corresponding to the specified block parameter.

See `LibBlockParameterDataTypeName` in `paramlib.tlc`.

LibBlockParameterDimensions

`LibBlockParameterDimensions(param)` returns a row vector of length `N` (where `N >= 1`) giving the dimensions of the parameter data.

For example,

```
%assign dims  = LibBlockParameterDimensions("paramName")
%assign nDims = SIZE(dims,1)
%foreach i=nDims
   /* Dimension %<i+1> = %<dims[i]> */
%endforeach
```

`LibBlockParameterDimensions` differs from `LibBlockParameterSize` in that it returns the dimensions of the parameter data prior to collapsing the `Matrix` parameter to a column-major vector. The collapsing occurs for run-time parameters that have specified their `outputAsMatrix` field as `False`.

See `LibBlockParameterDimensions` in `paramlib.tlc`.

LibBlockParameterIsComplex

`LibBlockParameterIsComplex(param)` returns 1 if the specified block parameter is complex, 0 otherwise.

See `LibBlockParameterIsComplex` in `paramlib.tlc`.

LibBlockParameterSize

`LibBlockParameterSize(param)` returns a vector of size 2 in the format `[nRows, nCols]` where `nRows` is the number of rows and `nCols` is the number of columns.

See `LibBlockParameterSize` in `paramlib.tlc`.

LibBlockParameterString

Based on the block parameter reference (`param`), `LibBlockParameterString(param)` returns the specified block parameter interpreted as a string, for example, this function returns:

- `STRINGOF(param.Value[0])` if the parameter is a row matrix
- `STRINGOF(param.Value)` otherwise

Note It is an error to invoke this function with a matrix-valued parameter with more than one row.

If you are accessing only a parameter value by using `LibBlockParameterValue` or `LibBlockParameterString`, consider converting the parameter from Tunable to Nontunable. Then, use ssWriteRTWParamSettings to write the value of the parameter to the *`model`*`.rtw` file. Inlining parameters reduces RAM usage because the code generator uses the numeric values of parameters, instead of their symbolic names, in the generated code.

See `LibBlockParameterString` in `paramlib.tlc`.

LibBlockParameterValue

Based on the block parameter reference (`param`) and the index element of the array (`elidx`), `LibBlockParameterValue(param, elIdx)` returns the numeric value of a parameter. You can use

the `LibBlockParameterWidth(param)` function to get the width of the array, and then use `elidx` (valid values of `0` to `width-1`) to get a particular element of the array.

If you are accessing only a parameter value by using `LibBlockParameterValue` or `LibBlockParameterString`, consider converting the parameter from Tunable to Nontunable. Then, use ssWriteRTWParamSettings to write the value of the parameter to the *`model`*`.rtw` file. Inlining parameters reduces RAM usage because the code generator uses the numeric values of parameters, instead of their symbolic names, in the generated code.

Example

If you want to generate code for a different integrator depending on a parameter for a block, use:

```
%assign mode = LibBlockParameterValue(Integrator, 0)
%switch (mode)
   %case 1
      %<CodeForIntegrator1>
      %break
   %case 2
      %<CodeForIntegrator2>
      %break
   %default
      Error: Unrecognized integrator value.
      %break
%endswitch
```

See `LibBlockParameterValue` in `paramlib.tlc`.

LibBlockParameterWidth

`LibBlockParameterWidth(param)` returns the number of elements (width) of a parameter.

See `LibBlockParameterWidth` in `paramlib.tlc`.

LibGetParameterValue

`LibGetParameterValue` returns the data vector of the model parameter written to the `ModelParameters` record in the *`model`*`.rtw`. For example, you can use:

```
%assign numRowValues = SIZE(LibGetParameterValue(RowIndex), 1)
```

For records that are not model parameters, the code generator returns the *`record`*`.Value`. Use the `LibGetParameterValue` function to access the data vector of the model parameter instead of fetching the *`record`*`.Value`.

If the data vector of the model parameter that the function `LibGetParameterValue` returns is a numeric scalar and results in an error for a numeric expression, append a `[0]` to it. For example, if

```
%assign sum = LibGetParameterValue(RowIndex) + 10
```

returns this error:

```
“The + operator only works on numeric arguments”
```

Then modify the expression as:

```
%assign sum = LibGetParameterValue(RowIndex)[0] + 10
```

See `LibGetParameterValue` in `paramlib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Block State and Work Vector Functions

LibBlockAssignDWork(dwork, ucv, lcv, sigIdx, rhs)

Based on the block's dwork index or record (`dwork`), the user control variable (`ucv`), the loop control variable (`lcv`), and the signal index (`sigIdx`), `LibBlockAssignDWork` assigns a block's dwork to a specified right hand side value (`rhs`).

See `LibBlockAssignDWork` in `customstoragelib.tlc`.

LibBlockContinuousState(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See `LibBlockContinuousState` in `blocklib.tlc`.

LibBlockContinuousStateDerivative(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See also `LibBlockDiscreteState`.

See `LibBlockContinuousStateDerivative` in `blocklib.tlc`.

LibBlockContStateDisabled(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See also `LibBlockDiscreteState`.

See `LibBlockContStateDisabled` in `blocklib.tlc`.

LibBlockDWork(dwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `dwork` element. The last input argument is overloaded to handle complex `dworks`.

`idx = "re3"` — Returns the real part of element 3 if `dwork` is complex, otherwise returns element 3.

`idx = "im3"` — Returns the imaginary part of element 3 if `dwork` is complex, otherwise returns `""`.

`idx = "3"` — Returns the complex container of element 3 if `dwork` is complex, otherwise returns element 3.

If either `ucv` or `lcv` is specified (i.e., it is not equal to `""`) then the index part of the last input argument (`sigIdx`) is ignored.

See `LibBlockDWork` in `blocklib.tlc`.

LibBlockDWorkAddr(dwork, ucv, lcv, idx)

Returns a string corresponding to the address of the specified block `dwork` element.

See `LibBlockDWorkAddr` in `blocklib.tlc`.

LibBlockDWorkDataTypeId(dwork)

Returns the data type ID of the specified block `dwork`.

See `LibBlockDWorkDataTypeId` in `blocklib.tlc`.

LibBlockDWorkDataTypeName(dwork, reim)

Returns the data type name of the specified block `dwork`.

See `LibBlockDWorkDataTypeName` in `blocklib.tlc`.

LibBlockDWorkIsComplex(dwork)

Returns 1 if the specified block `dwork`is complex. Returns 0 otherwise.

See `LibBlockDWorkIsComplex` in `blocklib.tlc`.

LibBlockDWorkName(dwork)

Returns the name of the specified block `dwork`.

See `LibBlockDWorkName` in `blocklib.tlc`.

LibBlockDWorkStorageClass(dwork)

Returns the storage class of the specified block `dwork`.

See `LibBlockDWorkStorageClass` in `blocklib.tlc`.

LibBlockDWorkStorageTypeQualifier(dwork)

Returns the storage type qualifier of the specified block `dwork`.

See `LibBlockDWorkStorageTypeQualifier` in `blocklib.tlc`.

LibBlockDWorkUsedAsDiscreteState(dwork)

Returns 1 if the specified block `dwork` is used as a discrete state, returns 0 otherwise.

See `LibBlockDWorkUsedAsDiscreteState` in `blocklib.tlc`.

LibBlockDWorkWidth(dwork)

Returns the width of the specified block `dwork`.

See `LibBlockDWorkWidth` in `blocklib.tlc`.

LibBlockDiscreteState(ucv, lcv, idx)

Returns a string corresponding to the specified block discrete state (`DSTATE`) element.

See `LibBlockDiscreteState` in `blocklib.tlc`.

LibBlockIWork(definediwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `IWORK` element. See `LibBlockRWork`.

See `LibBlockIWork` in `blocklib.tlc`.

LibBlockMode(ucv, lcv, idx)

Returns a string corresponding to the specified block `MODE` element.

See `LibBlockMode` in `blocklib.tlc`.

LibBlockNonSampledZC(ucv, lcv, NSZCIdx)

Returns a string corresponding to the specified block `NSZC`.

`LibBlockNonSampledZC` returns an element for the nonsampled zero-crossing state based on `ucv`, `lcv`, and `NSZCIdx`.

Arguments

`ucv` — User control variable string

`lcv` — Loop control variable string

`NSZCIdx` — Nonsampled zero-crossing index

See `LibBlockNonSampledZC` in `blocklib.tlc`.

LibBlockPWork(definedpwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `PWORK` element. See `LibBlockRWork`.

See `LibBlockPWork` in `blocklib.tlc`.

LibBlockRWork(definedrwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `RWORK` element. The first argument, `definedrwork`, is a symbol defined in the `mdlRTW` routine of the C MEX file with code like:

```
ssWriteRTWWorkVect([...], "RWork", [...], "MyRWorkName", [...])
```

Alternatively, if `RWork` defines have not been made, `definedrwork` is ignored and the raw `RWork` vector is accessed. In this case, uses in a loop rolling context are disallowed.

See `LibBlockRWork` in `blocklib.tlc`.

LibBlockZCSignalValue(ucv, lcv, zcsIdx, zcElIdx)

Purpose

Returns a string corresponding to the specified block `ZCSignalValue`

Arguments

`ucv`

User control variable string.

`lcv`

Loop control variable string.

`zcsIdx`

zc signal Idx

`zcElIdx`

Idx of zc signal element in the zc signal

Description

`LibBlockZCSignalValue` returns an element for the zero crossing state based on `ucv`, `lcv`, and `zcsIdx`.

See `LibBlockZCSignalValue` in `blocklib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Block Path and Error Reporting Functions

LibBlockReportError(block, errorstring)

Use `LibBlockReportError` when reporting errors for a block. `LibBlockReportError` is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

`LibBlockReportError` can be called with or without the block record scoped. To call the function without a block record scoped, pass the block record. To call the function when the block is scoped, pass `block = []`.

```
LibBlockReportError([],"error string")
  -- If block is scoped
LibBlockReportError(blockrecord,"error string")
  -- If block record is available
```

See `LibBlockReportError` in `utillib.tlc`.

LibBlockReportFatalError(block, errorstring)

Use `LibBlockReportFatalError` when reporting fatal (assert) errors for a block. Use `LibBlockReportFatalError` for defensive programming. Refer to “Generating Errors from TLC Files” on page A-2.

See `LibBlockReportFatalError` in `utillib.tlc`.

LibBlockReportWarning(block, warnstring)

Use `LibBlockReportWarning` when reporting warnings for a block. `LibBlockReportWarning` is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

`LibBlockReportWarning` can be called with or without the block record scoped. To call this function without a block record scoped, pass the block record. To call this function when the block is scoped, pass `block = []`.

```
LibBlockReportWarning([],"warn string")
--    If block is scoped
LibBlockReportWarning(blockrecord,"warn string)
--    If block record is available
```

See `LibBlockReportWarning` in `utillib.tlc`.

LibGetBlockName(block)

`LibGetBlockName` returns the short block path name string for a block record, excluding carriage returns and other special characters that can be present in the name.

See `LibGetBlockName` in `utillib.tlc`.

LibGetBlockPath(block)

`LibGetBlockPath` returns the full block path name string for a block record, including carriage returns and other special characters that can be present in the name. Currently, the only other special string sequences defined are `'/*'` and `'*/'`.

The full block path name string is useful when you are accessing blocks from MATLAB. For example, you can use the full block name with `hilite_system` via `FEVAL` to match the Simulink path name exactly.

Use `LibGetFormattedBlockPath` to get a block path suitable for placing in a comment or error message.

See `LibGetBlockPath` in `utillib.tlc`.

LibGetFormattedBlockPath(block)

`LibGetFormattedBlockPath` returns the full path name string of a block without special characters. The string returned from `LibGetFormattedBlockPath` is suitable for placing the block name, in comments or generated code, on a single line.

Currently, the special characters are carriage returns, `'/*'`, and `'*/'`. A carriage return is converted to a space, `'/*'` is converted to `'/+'`, and `'*/'` is converted to `'+/'`. Note that a `'/'` in the name is automatically converted to a `'//'` to distinguish it from a path separator.

Use `LibGetBlockPath` to get the block path for MATLAB functions used in reference blocks in your model.

See `LibGetFormattedBlockPath` in `utillib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Code Configuration Functions

LibAddSourceFileCustomSection(file, builtInSection, newSection)

Adds a custom section to a source file. You must associate a custom section with one of the built-in sections: Includes, Defines, Types, Enums, Definitions, Declarations, Functions, or Documentation. Nothing happens if the section already exists, except to report an error if a inconsistent built-in section association is attempted. `LibAddSourceFileCustomSection` is available only with the Embedded Coder product.

Arguments

`file` — Source file reference

`builtInSection` — Name of the associated built-in section

`newSection` — Name of the new (custom) section

See `LibAddSourceFileCustomSection` in `codetemplatelib.tlc`.

LibAddToCommonIncludes(incFileName)

Adds items to a list of `#include /package` specification items. Each member of the list is unique. Attempting to add a duplicate member does nothing.

`LibAddToCommonIncludes` should be called from block TLC methods to specify generation of `#include` statements in *`model`*`.h`. Specify the names of files on the include path inside angle brackets, e.g., `<sysinclude.h>`. Specify the names of local files without angle brackets, e.g., `myinclude.h`. Each call to `LibAddToCommonIncludes` adds the specified file to the list only if it is not already there. Filenames with and without angle brackets (e.g., `<math.h>` and `math.h`) are considered different. The `#include` statements are placed inside *`model`*`.h`.

Example

```
LibAddToCommonIncludes("tpu332lib.h")
```

See `LibAddToCommonIncludes` in `cachelib.tlc`.

LibAddToModelSources(newFile)

`LibAddToModelSources` serves two purposes:

- To notify the build process that it must build with the specified source file
- To update the `SOURCES: file1.c file2.c ...` comment in the generated code.

For inlined S-functions, `LibAddToModelSources` is generally called from `BlockTypeSetup`. `LibAddToModelSources` adds a filename to the list of sources for building this model. `LibAddToModelSources` returns `1` if the filename passed in was a duplicate (i.e., it was already in the sources list) and `0` if it was not a duplicate.

You can use the `LibAddToModelSources` function for other purposes in TLC aside from writing S-functions. If you write your own S-functions, use the `SFunctionModules` block parameter instead of `LibAddToModelSources`. See `LibAddToModelSources` in `utillib.tlc`.

LibCacheDefine(buffer)

Each call to `LibCacheDefine` appends your buffer to the existing cache buffer. For blocks, `LibCacheDefine` is generally called from `BlockTypeSetup`.

`LibCacheDefine` caches `#define` statements for inclusion in *`model`*`_private.h` or *`model`*`.c`. Call `LibCacheDefine` from inside `BlockTypeSetup` to cache a `#define` statement. Each call to `LibCacheDefine` appends your buffer to the existing cache buffer. The `#define` statements are placed inside *`model`*`_private.h` or *`model`*`.c`.

Example

```
%openfile buffer
#define INTERP(x,x1,x2,y1,y2) ( y1+((y2 - y1)/(x2 - x1))*(x-x1))
#define this that
%closefile buffer
%<LibCacheDefine(buffer)>
```

See `LibCacheDefine` in `codecache_api.tlc`.

LibCacheExtern(buffer)

`LibCacheExtern` should be called from inside `BlockTypeSetup` to cache an `extern` statement. Each call to `LibCacheExtern` appends your buffer to the existing cache buffer. The `extern` statements are placed in *`model`*`_private.h`.

Example

```
%openfile buffer
 extern real_T mydata;
%closefile buffer
%<LibCacheExtern(buffer)>
```

See `LibCacheExtern` in `codecache_api.tlc`.

LibCacheFunctionPrototype(buffer)

`LibCacheFunctionPrototype` should be called from inside `BlockTypeSetup` to cache a function prototype. Each call to `LibCacheFunctionPrototype` appends your buffer to the existing cache buffer. The prototypes are placed inside *`model`*`_private.h`.

Example

```
%openfile buffer
 extern int_T fun1(real_T x);
 extern real_T fun2(real_T y, int_T i);
%closefile buffer
%<LibCacheFunctionPrototype(buffer)>
```

See `LibCacheFunctionPrototype` in `codecache_api.tlc`.

LibCacheTypedefs(buffer)

`LibCacheTypedefs` should be called from inside `BlockTypeSetup` to cache `typedef` declarations. Each call to `LibCacheTypedefs` appends your buffer to the existing cache buffer. The `typedef` statements are placed inside *`model`*`.h` (or *`model`*`_common.h`).

Example

```
%openfile buffer
typedef foo bar;
%closefile buffer
%<LibCacheTypedefs(buffer)>
```

See `LibCacheTypedefs` in `codecache_api.tlc`.

LibCallModelInitialize()

Returns code for calling the model's initialize function (valid for ERT only).

See `LibCallModelInitialize` in `codetemplatelib.tlc`.

LibCallModelStep(tid)

Returns code for calling the model's step function (valid for ERT only).

See `LibCallModelStep` in `codetemplatelib.tlc`.

LibCallModelTerminate()

Returns code for calling the model's terminate function (valid for ERT only).

See `LibCallModelTerminate` in `codetemplatelib.tlc`.

LibCallSetEventForThisBaseStep(buffername)

Returns code for calling the model's set events function (valid for ERT only).

Argument

`buffername` — Name of the variable used to buffer the events. For the example `ert_main.c`, this is `eventFlags`.

See `LibCallSetEventForThisBaseStep` in `codetemplatelib.tlc`.

LibClearFileSectionContents(fileIdx, attrib)

Before writing file to disk, clear file sections with custom values.

Arguments

`fileIdx` (scope or number) — File index

`attrib` (string) — Name of model attribute

See `LibGetSourceFileAttribute` in `codetemplatelib.tlc`.

LibCreateSourceFile(type, creator, name)

Creates a new C or C++ file and returns its reference. If the file already exists, `LibCreateSourceFile` returns the existing file's reference.

Syntax

```
%assign fileH = LibCreateSourceFile
                ("Source", "Custom", "foofile")
```

Arguments

`type` (string) — Valid values are `"Source"` and `"Header"` for `.c` and `.h` files, respectively.

`creator` (string) — Who is creating the file? An error is reported if different creators attempt to create the same file.

`name` (string) — Base name of the file (i.e., without the extension). Note that files are not written to disk if they are empty.

Returns

Reference to the model file (scope).

See `LibCreateSourceFile` in `codetemplatelib.tlc`.

LibGetFileRecordName (file)

Returns model file name (including the path) without the file extension. To retrieve the file name (including the path) with the file extension, use `LibGetSourceFileSection`.

Arguments

`file` — Source file reference

See `LibGetFileRecordName` in `codetemplatelib.tlc`.

LibGetMdlPrvHdrBaseName()

Returns the base name of the model's private header file, for example, *`model`*`_private.h`.

See `LibGetMdlPrvHdrBaseName` in `codetemplatelib.tlc`.

LibGetMdlPubHdrBaseName()

Returns the base name of the model's public header file, for example, *`model`*`.h`.

See `LibGetMdlPubHdrBaseName` in `codetemplatelib.tlc`.

LibGetMdlSrcBaseName()

Returns the base name of the model's main source file, for example, *model*.c.

See LibGetMdlSrcBaseName in codetemplatelib.tlc.

LibGetMdlDataSrcBaseName()

Returns the base name of the model's data file, for example, *model*_data.c.

See LibGetMdlDataSrcBaseName in codetemplatelib.tlc.

LibGetMdlTypesHdrBaseName()

Returns the base name of the model types file, for example, *model*_types.h.

See LibGetMdlTypesHdrBaseName in codetemplatelib.tlc.

LibGetMdlCapiHdrBaseName()

Returns the base name of the model capi header file, for example, *model*_capi.h.

See LibGetMdlCapiHdrBaseName in codetemplatelib.tlc.

LibGetMdlCapiSrcBaseName()

Returns the base name of the model capi source file, for example, *model*_capi.c.

See LibGetMdlCapiSrcBaseName in codetemplatelib.tlc.

LibGetMdlCapiHostHdrBaseName()

Returns the base name of the model capi host header file, for example, *model*_host_capi.h.

See LibGetMdlCapiHostHdrBaseName in codetemplatelib.tlc.

LibGetMdlTestIfHdrBaseName()

Returns the base name of the model testinterface header file, for example, *model*_testinterface.h.

See LibGetMdlTestIfHdrBaseName in codetemplatelib.tlc.

LibGetMdlTestIfSrcBaseName()

Returns the base name of the model testinterface source file, for example, *model*_testinterface.c.

See LibGetMdlTestIfSrcBaseName in codetemplatelib.tlc.

LibGetDataTypeTransHdrBaseName()

Returns the base name of the data type transition file, for example, *model*_dt.h for code generation's Real-Time and Embedded-C code formats.

See LibGetDataTypeTransHdrBaseName in codetemplatelib.tlc.

LibGetModelDotCFile()

Returns a reference to the *model*.c or .cpp source file. You can then cache additional code using LibSetSourceFileSection.

Syntax

```
%assign srcFile = LibGetModelDotCFile()
%<LibSetSourceFileSection(srcFile, "Functions", mybuf)>
```

Returns

Returns a reference to the *model*.c or .cpp source file.

See LibGetModelDotCFile in codetemplatelib.tlc.

LibGetModelDotHFile()

Returns a reference to the *model*.h source file. You can then cache additional code using LibSetSourceFileSection.

Syntax

```
%assign hdrFile = LibGetModelDotHFile()
%<LibSetSourceFileSection(hdrFile, "Functions", mybuf)>
```

Returns

Returns a reference to the *model*.h source file.

See LibGetModelDotHFile in codetemplatelib.tlc.

LibGetModelName()

Returns the name of the model (without an extension).

See LibGetModelName in codetemplatelib.tlc.

LibGetNumSourceFiles()

Returns the number of source files (.c or .cpp and .h) that have been created.

Syntax

```
%assign numFiles = LibGetNumSourceFiles()
```

Returns

Returns the number of files (number).

See `LibGetNumSourceFiles` in `codetemplatelib.tlc`.

LibGetRTModelErrorStatus()

Returns the code required to get the model error status.

Syntax

```
%<LibGetRTModelErrorStatus()>;
```

See `LibGetRTModelErrorStatus` in `codetemplatelib.tlc`.

LibGetSourceFileAttribute(fileIdx, attrib)

Returns the specified attribute of a file. The table lists the valid attributes.

Attribute			
Name (with file extension)	SystemsInFile	IsEmpty	SharedType
BaseName	RequiredIncludes	Indent	CodeTemplate
Type	UtilityIncludes	WrittenToDisk	OutputDirectory
Creator	Filter	Shared	Group

Arguments

`fileIdx` (scope or number) — File index

`attrib` (string) — Name of model attribute

See `LibGetSourceFileAttribute` in `codetemplatelib.tlc`.

LibGetSourceFileFromIdx(fileIdx)

Returns a model file reference based on its index. This reference can be useful for a common operation on all files, for example, to set the leading file banner of all files.

Syntax

```
%assign fileH = LibGetSourceFileFromIdx(fileIdx)
```

Argument

`fileIdx` (number) — Index of model file

Returns

Reference (scope) to the model file.

See `LibGetSourceFileFromIdx` in `codetemplatelib.tlc`.

LibGetSourceFileSection(fileIdx, section)

Returns the contents of a file. See “LibSetSourceFileSection(fileH, section, value)” on page 22-46 for a list of valid sections.

Arguments

`fileIdx` (scope or number) — File index

`section` (string) — File section of interest

See `LibGetSourceFileSection` in `codetemplatelib.tlc`.

LibGetSourceFileTag(fileIdx)

Returns *`fileName`*`_h` and *`fileName`*`_c` for header and source files, respectively, where *`fileName`* is the name of the model file.

Syntax

```
%assign tag = LibGetSourceFileTag(fileIdx)
```

Argument

`fileIndex` (number) — File index

Returns

Returns the tag (string).

See `LibGetSourceFileTag` in `codetemplatelib.tlc`.

LibMdlRegCustomCode(buffer, location)

Places declaration statements and executable code inside the *`model`*`_initialize` function.

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

See `LibMdlRegCustomCode` in `hookslib.tlc`.

LibMdlStartCustomCode(buffer, location)

Places declaration statements and executable code inside the start function. Start code is executed once, during the model initialization phase.

Syntax

```
LibMdlStartCustomCode(buffer, location)
```

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibMdlStartCustomCode` places declaration statements and executable code inside the start function. This code is output into the following functions, depending on the current value for the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
`model``_initialize`	`Embedded-C` (if not `ModelReferenceCoderTarget`)
`mdlStart`	`Embedded-C` (if `ModelReferenceCoderTarget`)
`mdlStart`	`S-Function`
`MdlStart`	`RealTime`

Each call to `LibMdlStartCustomCode` appends your buffer to the internal cache buffer.

See `LibMdlStartCustomCode` in `hookslib.tlc`.

LibMdlTerminateCustomCode(buffer, location)

Purpose

Places declaration statements and executable code inside the terminate function.

Syntax

```
LibMdlTerminateCustomCode(buffer, location)
```

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibMdlTerminateCustomCode` places declaration statements and executable code inside the terminate function. This code is output into the following functions, depending on the current value of the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
*model*_terminate	`Embedded-C`
`mdlTerminate`	`S-Function`
`MdlTerminate`	`RealTime`

Each call to `LibMdlTerminateCustomCode` appends your buffer to the internal cache buffer.

See `LibMdlTerminateCustomCode` in `hookslib.tlc`.

LibNotifySymbolExportedFromFile

Notifies the code generation infrastructure that the symbol is being exported from the file.

This function is recommended for custom user symbols such as variable, type, macro, and function declarations. The function signals the code generation infrastructure to include appropriate headers when the passed symbol is used in auto generated files.

Arguments

`symbol` — Macro, variable, type, or function name, specified as a string.

`fileNameWithExtension` — File name with language extension, specified as a string. For example, `<modelName>.h`.

Example

```
%openfile myFcnDecl
  extern real_T myCustomFcn(void);
%closefile myFcnDecl
%<LibCacheFunctionPrototype(myFcnDecl)>
%assign modelPrivateFile = "%<LibGetMdlPrvHdrBaseName()>.h
%<LibNotifySymbolExportedFromFile("myCustomFcn",modelPrivateFile)>
```

LibNotifySymbolUsedByFile

Notifies the code generation infrastructure that the symbol is being used by the file.

This function is recommended for custom user symbols such as variable, type, macro, and function declarations. The function signals the code generation infrastructure to include headers for the symbol passed to the function.

Arguments

`symbol` — Macro, variable, type, or function name, specified as a string.

`fileNameWithExtension` — File name with language extension, specified as a string. For example, `<modelName>.h`.

Example

```
%% The following line is expanded and placed in <model>.c
%<y> = myCustomFcn();
%assign modelSrcFile = LibGetModelDotCFile()
%<LibNotifySymbolUsedByFile("myCustomFcn",modelSrcFile)>
```

LibSetRTModelErrorStatus(str)

Returns the code required to set the model error status.

Syntax

```
LibSetRTModelErrorStatus("\"Overrun\"")
```

Argument

`str` (string) — char * to a C string

See `LibSetRTModelErrorStatus` in `codetemplatelib.tlc`.

LibSetSourceFileCodeTemplate(opFile, name)

By default, `*.c` and `*.h` files are generated with the code templates specified in the **Code Generation > Templates** pane of the Configuration Parameters dialog box. `LibSetSourceFileCodeTemplate` allows you to change the template for a file.

Note Custom templates are a feature of the Embedded Coder product.

Syntax

```
%assign tag = LibSetSourceFileCodeTemplate(opFile,name)
```

Arguments

`opFile` (scope) — Reference to file

`name` (string) — Name of the desired template

Returns

Nothing

See `LibSetSourceFileCodeTemplate` in `codetemplatelib.tlc`.

LibSetSourceFileCustomSection(file, attrib, value)

Adds to the contents of a custom section previously created with `LibAddSourceFileCustomSection`. Available only with Embedded Coder software.

Arguments

`file` (scope or number) — Source file reference or index

`attrib` (string) — Name of custom section

`value` (string) — Value to be appended to section

See `LibSetSourceFileCustomSection` in `codetemplatelib.tlc`.

LibSetSourceFileOutputDirectory(opFile, name)

By default, `*.c` and `*.h` files are generated into a build folder at the current location. `LibSetSourceFileOutputDirectory` allows you to change the folder into which a specified source file is to be generated. Note that the caller is responsible for specifying a valid folder.

Syntax

```
%assign tag = LibSetSourceFileOutputDirectory(opFile,dirName)
```

Arguments

`opFile` (scope) — Reference to file

`dirName` (string) — Name of the desired output folder

Returns

Nothing

See `LibSetSourceFileOutputDirectory` in `codetemplatelib.tlc`.

LibSetSourceFileSection(fileH, section, value)

Adds to the contents of a specified section within a specified file. Valid file sections include

File Section	Description
`Banner`	Set the file banner (comment) at the top of the file.
`Includes`	Append to the `#include` section.
`Defines`	Append to the `#define` section.
`IntrinsicTypes`	Append to the intrinsic `typedef` section. Intrinsic types are those that depend only on intrinsic C types.
`PrimitiveTypedefs`	Append to the primitive `typedef` section. Primitive `typedef`s are those that depend only on intrinsic C types and `typedef`s previously defined in the `IntrinsicTypes` section.
`UserTop`	Append to the User Top section.

File Section	Description
`Typedefs`	Append to the `typedef` section. The `typedef`s can depend on a previously defined type.
`Enums`	Append to the enumerated types section.
`Definitions`	Append to the data definition section.
`ExternData`	(Reserved) Code generator `extern` data.
`ExternFcns`	(Reserved) Code generator `extern` functions.
`FcnPrototypes`	(Reserved) Code generator function prototypes.
`Declarations`	Append to the data declaration section.
`Functions`	Append to the C functions section.
`CompilerErrors`	Append to the `#error` section.
`CompilerWarnings`	Append to the `#warning` section.
`Documentation`	Append to the documentation (comment) section.
`UserBottom`	Append to the User Bottom section.

The code generator orders the code as listed above.

Syntax

Example (iterating over the files):

```
%openfile tmpBuf
  whatever
%closefile tmpBuf

%foreach fileIdx = LibGetNumSourceFiles()
  %assign fileH = LibGetSourceFileFromIdx(fileIdx)
  %<LibSetSourceFileSection(fileH,"SectionOfInterest",tmpBuf)>
%endforeach

%assign fileH = LibCreateSourceFile("Header","Custom","foofile")
%<LibSetSourceFileSection(fileH,"Defines","#define FOO 5.0\n")>
```

Arguments

`fileH` (scope or number) — Reference or index to a file

`section` (string) — File section of interest

`value` (string) — Value

See `LibSetSourceFileSection` in `codetemplatelib.tlc`.

LibSystemDerivativeCustomCode(system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's derivative function.

Syntax

```
LibSystemDerivativeCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose derivative function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemDerivativeCustomCode` places declaration statements and executable code inside the derivative function for the subsystem specified by `system`. This code is output into the following functions, depending on the current value of the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
`mdlDerivatives`	`S-Function`
`model``_derivatives`	`RealTime`

`LibSystemDerivativeCustomCode` is not relevant when the value of the `CodeFormat` TLC variable is `Embedded-C`, because blocks with continuous states cannot be used.

Each call to `LibSystemDerivativeCustomCode` appends your buffer to the internal cache buffer. An error is generated if you attempt to add code to a subsystem that does not have continuous states.

See `LibSystemDerivativeCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemDerivativeCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibSystemDisableCustomCode(system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's disable function.

Syntax

```
LibSystemDisableCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose disable function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemDisableCustomCode` places declaration statements and executable code inside the disable function for the subsystem specified by `system`. Each call to `LibSystemDisableCustomCode` appends your buffer to the internal cache buffer.

An error is generated if you attempt to add code to a subsystem that does not have a disable function.

See `LibSystemDisableCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemDisableCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibSystemEnableCustomCode(system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's enable function.

Syntax

```
LibSystemEnableCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose enable function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemEnableCustomCode` places declaration statements and executable code inside the enable function for the subsystem specified by `system`. Each call to `LibSystemEnableCustomCode` appends your buffer to the internal cache buffer.

An error is generated if you attempt to add code to a subsystem that does not have an enable function.

See `LibSystemEnableCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemEnableCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibSystemInitializeCustomCode(system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's initialize function.

Syntax

```
LibSystemInitializeCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose initialize function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemInitializeCustomCode` places declaration statements and executable code inside the initialize function for the subsystem specified by `system`. This code is output into the following functions, depending on the current value of the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
`model``_initialize`	`Embedded-C`
`mdlInitializeConditions`	`S-Function`
`MdlStart`	`RealTime`

Code for a subsystem is output into the subsystem's initialization function. Each call to `LibSystemInitializeCustomCode` appends your buffer to the internal cache buffer.

Note Enable systems that are not configured to reset on enable are inlined into `MdlStart`. For this case, the subsystem's custom code is found in `MdlStart` above and below the enable subsystem's initialization code.

See `LibSystemInitializeCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemInitializeCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibSystemOutputCustomCode(system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's output function.

Syntax

```
LibSystemOutputCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose output function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemOutputCustomCode` places declaration statements and executable code inside the output function for the subsystem specified by `system`. This code is output into the following functions, depending on the current value of the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
`model``_step`	`Embedded-C` (`CombineOutputUpdateFcns` is `1`)
`model``_output`	`Embedded-C` (`CombineOutputUpdateFcns` is `0`)
`mdlOutputs`	`S-Function`
`MdlOutputs`	`RealTime`

Each call to `LibSystemOutputCustomCode` appends your buffer to the internal cache buffer.

See `LibSystemOutputCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemOutputCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibSystemUpdateCustomCode(system, buffer, location)

Purpose

Places code inside a subsystem's update function.

Syntax

```
LibSystemUpdateCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose update function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemUpdateCustomCode` places declaration statements and executable code inside the update function for the subsystem specified by `system`. This code is output into the following functions, depending on the current value of the `CodeFormat` TLC variable:

Function Name	Value of `CodeFormat` TLC variable
`model``_step`	`Embedded-C` (`CombineOutputUpdateFcns` is `1`)
`model``_update`	Embedded-C (`CombineOutputUpdateFcns` is `0`)
`mdlUpdate`	`S-Function`
`MdlUpdate`	RealTime

Each call to `LibSystemUpdateCustomCode` appends your buffer to the internal cache buffer.

See `LibSystemUpdateCustomCode` in `hookslib.tlc`.

Note To avoid a potential mismatch between simulation and code generation results, do not use `LibSystemUpdateCustomCode` to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

LibWriteModelData()

Returns data for the model (valid for ERT only).

See `LibWriteModelData` in `codetemplatelib.tlc`.

LibWriteModelInput(tid, rollThreshold)

Returns the code for writing to a specified root input (that is, a model inport block). This function is valid for ERT only, and not valid for referenced models.

Arguments

`tid` (number) — Task identifier (`0` is fastest rate and `n` is the slowest)

`rollThreshold` — Width of signal before wrapping in a `for` loop.

See `LibWriteModelInput` in `codetemplatelib.tlc`.

LibWriteModelInputs()

Returns code that writes to all root inputs (that is, the model inport blocks). This function is valid for ERT only, and is not valid for referenced models.

See `LibWriteModelInputs` in `codetemplatelib.tlc`.

LibWriteModelOutput(tid, rollThreshold)

Returns code that writes to a specified root output (that is, a model outport block). This function is valid for ERT only, and not valid for referenced models.

Arguments

`tid` (number) — Task identifier (`0` is fastest rate and `n` is the slowest)

`rollThreshold` — Width of signal before wrapping in a `for` loop.

See `LibWriteModelOutput` in `codetemplatelib.tlc`.

LibWriteModelOutputs()

Returns code that writes to all root outputs (that is, the model outport blocks). This function is valid for ERT only, and not valid for referenced models.

See `LibWriteModelOutputs` in `codetemplatelib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Sample Time Functions

LibAsynchronousTriggeredTID(tid)

Returns whether this TID corresponds to a asynchronous triggered rate.

See `LibAsynchronousTriggeredTID` in `utillib.tlc`.

LibAsyncTaskAccessTimeInFcn(tid, fcnType)

Returns 1 if the specified asynchronous task identifier (TID) is given function type.

See `LibAsyncTaskAccessTimeInFcn` in `utillib.tlc`.

LibBlockSampleTime(block)

Returns the block's sample time. The returned value depends on the sample time classification of the block, as shown in the following table.

Block Classification	Returned Value
Discrete	The actual sample time of a block (real > `0`)
Continuous	`0.0`
Triggered	`-1.0`
Constant	`-2.0`

See `LibBlockSampleTime` in `blocklib.tlc`.

LibGetClockTick(tid)

Returns integer task time (current clock tick of the task timer). The resolution of the timer can be obtained from `LibGetClockTickStepSize(tid)`. The data type ID of the timer can be obtained from `LibGetClockTickDataTypeId(tid)`.

See `LibGetClockTick` in `utillib.tlc`.

LibGetClockTickDataTypeId(tid)

Returns clock tick data type ID.

See `LibGetClockTickDataTypeId` in `utillib.tlc`.

LibGetClockTickHigh(tid)

Returns high-order word of the integer task time. `LibGetClockTickHigh` is used when `uint32` pairs are used to store absolute time. The resolution of a clock tick can be obtained from `LibGetClockTickStepSize(tid)`.

See `LibGetClockTickHigh` in `utillib.tlc`.

LibGetClockTickStepSize(tid)

Returns clock tick step size, which is the resolution of the integer task time. `LibGetClockTickStepSize` cannot be used if the task does not have a timer.

See `LibGetClockTickStepSize` in `utillib.tlc`.

LibGetElapseTime(system)

Returns time elapsed since the last time the subsystem started to execute.

See `LibGetElapseTime` in `utillib.tlc`.

LibGetElapseTimeCounter(system)

Returns an integer elapsed time. This is the number of clock ticks elapsed since the last time the system started. To get real-world elapsed time, this integer elapsed time must be multiplied by the applicable resolution.

You can obtain the resolution by calling `LibGetElapseTimeResolution(system)`. You can obtain the data type ID of the integer elapsed time counter by calling `LibGetElapseTimeCounterDTypeId(system)`.

See `LibGetElapseTimeCounter` in `utillib.tlc`.

LibGetElapseTimeCounterDTypeId(system)

Returns the date type ID of the integer elapsed time returned by `LibGetElapseTimeCounter`.

See `LibGetElapseTimeCounterDTypeId` in `utillib.tlc`.

LibGetElapseTimeResolution(system)

Returns the resolution of the elapsed time returned by `LibGetElapseTimeCounter`.

See `LibGetElapseTimeResolution` in `utillib.tlc`.

LibGetGlobalTIDFromLocalSFcnTID(sfcnTID)

Returns the model task identifier (sample time index) corresponding to the specified local S-function task identifier or port sample time. `LibGetGlobalTIDFromLocalSFcnTID` allows you to use one function to determine a global TID, independent of port- or block-based sample times.

The input argument to `LibGetGlobalTIDFromLocalSFcnTID` should be one of the following:

- For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S, N)` with `N > 1` was specified), `sfcnTID` is a nonnegative integer giving the corresponding local S-function sample time.
- For port-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S, PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string

of the form "InputPortIdx*I*" or "OutputPortIdx*I*", where *I* is an integer ranging from 0 to the number of ports (e.g., "InputPortIdx0".

Examples

Multirate block

```
%assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
```

or

```
%assign globalTID =
LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx4")
%assign period =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
%assign offset =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
```

Inherited sample time block

```
%switch (LibGetSFcnTIDType(0))
  %case "discrete"
  %case "continuous"
    %assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
    %assign period = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
    %assign offset = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
    %break
  %case "triggered"
    %assign period = -1
    %assign offset = -1
    %break
  %case "constant"
    %assign period = rtInf
    %assign offset = 0
    %break
  %default
    %<LibBlockReportFatalError([],"Unknown tid type")>
%endswitch
```

See LibGetGlobalTIDFromLocalSFcnTID in utillib.tlc.

LibGetNumAsyncTasks()

Return the number of asynchronous tasks in generated code.

See LibGetNumAsyncTasks in utillib.tlc.

LibGetNumSFcnSampleTimes(block)

Returns the number of S-function sample times for a block.

See LibGetNumSFcnSampleTimes in utillib.tlc.

LibGetNumSyncPeriodicTasks()

Return the number of periodic tasks in generated code.

See `LibGetNumSyncPeriodicTasks` in `utillib.tlc`.

LibGetNumTasks()

Return the number of tasks in generated code.

See `LibGetNumTasks` in `utillib.tlc`.

LibGetSampleTimePeriodAndOffset(tid, idx)

Returns the sample time period value or offset value for a specified task.

Arguments

`tid` — Specify the identifier of the task for which to return information.

`idx` — Specify 0 to return the sample time period value or 1 to return the sample time offset value.

Examples

```
%% Get sample time period and offset for task 0
%assign sampleTime = LibGetSampleTimePeriodAndOffset(0,0)
%assign offsetTime = LibGetSampleTimePeriodAndOffset(0,1)

%% Get sample time periods for tasks 0 and 1
%assign periodTID0 = LibGetSampleTimePeriodAndOffset(0,0)
%assign periodTID1 = LibGetSampleTimePeriodAndOffset(1,0)
```

See `LibGetSampleTimePeriodAndOffset` in `codetemplatelib.tlc`.

LibGetSFcnTIDType(sfcnTID)

Returns the type of the specified S-function task identifier (`sfcnTID`). Possible values are:

- `"continuous"` — The specified `sfcnTID` is continuous.
- `"discrete"` — The specified `sfcnTID` is discrete.
- `"triggered"` — The specified `sfcnTID` is triggered.
- `"constant"` — The specified `sfcnTID` is constant.

The format of `sfcnTID` must be the same as for `LibIsSFcnSampleHit`.

Note This is useful primarily in the context of S-functions that specify an inherited sample time.

See `LibGetSFcnTIDType` in `utillib.tlc`.

LibGetTaskTime(tid)

Returns a string to access the absolute time of the task, which is a floating-point number. However, if you have configured the model for integer-only code generation (by deselecting **Support floating-point numbers**), the string represents an integer equal to the number of base rate ticks (the absolute time divided by the base sample time) rather than the absolute time. In integer-only code, the task time is an integer time value whose resolution is the model fundamental step size.

`LibGetTaskTime` is the TLC version of the SimStruct macro: `"ssGetTaskTime(S,tid)"`.

See `LibGetTaskTime` in `utillib.tlc`.

LibGetTaskTimeFromTID(block)

Returns a string to access the absolute time of the task associated with the block.

If the value of the `CodeFormat` TLC variable is not `Embedded-C`, `LibGetTaskTimeFromTID` returns the string `"RTMGet("T")"` if the block is constant or the system is single rate, and `"RTMGetTaskTimeForTID(tid)"` otherwise.

If the value for the `CodeFormat` TLC variable is `Embedded-C`, `LibGetTaskTimeFromTID` returns `"RTMGetTaskTimeForTID(tid)"`.

If the block is constant or the system is single rate, this is the TLC version of the SimStruct macro: `"ssGetT(S)"` and `"ssGetTaskTime(S, tid)"` otherwise.

In both cases, `S` is the name of the SimStruct.

See `LibGetTaskTimeFromTID` in `utillib.tlc`.

LibGetTID01EQ()

Returns the value of the TID01EQ flag — true (1) if sampling rates of the continuous task and the first discrete task are equal and false (0) otherwise.

When a model has continuous time and the flag TID01EQ is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and TID01EQ is true, the generated code has a single-rate call interface.

Use `LibGetTID01EQ` to detect and account for the case where continuous time and the fastest discrete time are treated as one rate.

See `LibGetTID01EQ` in `codetemplatelib.tlc`, `bareboard_mrmain.tlc`, and `ertmainlib.tlc`.

LibIsContinuous(TID)

Returns 1 if the specified task identifier (`TID`) is continuous and 0 otherwise. Note that task identifiers equal to `"triggered"` or `"constant"` are not continuous.

See `LibIsContinuous` in `utillib.tlc`.

LibIsDiscrete(TID)

Returns 1 if the specified task identifier (`TID`) is discrete and 0 otherwise. Note that task identifiers equal to `"triggered"` or `"constant"` are not discrete.

See `LibIsDiscrete` in `utillib.tlc`.

LibIsSFcnSampleHit(sfcnTID)

Returns 1 if a sample hit occurs for the specified local S-function task identifier (`TID`) and 0 otherwise.

The input argument to `LibIsSFcnSampleHit` should be one of the following:

- `sfcnTID`: integer (e.g., 2)

  For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,N)` with N > 1 was specified), `sfcnTID` is an integer starting at 0 of the corresponding local S-function sample time.
- `sfcnTID`: `"InputPortIdx`*I*`"` or `"OutputPortIdx`*I*`"` (e.g., `"InputPortIdx0"`).

  For port-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string giving the input (or output) port index.

Examples

- Consider a multirate S-function block with four block sample times. The call `LibIsSFcnSampleHit(2)` returns the code to check for a sample hit on the third S-function block sample time.
- Consider a multirate S-function block with three input and eight output sample times. The call `LibIsSFcnSampleHit("InputPortIdx0")` returns the code to check for a sample hit on the first input port. The call `LibIsSFcnSampleHit("OutputPortIdx7")` returns the code to check for a sample hit on the eighth output port.

See `LibIsSFcnSampleHit` in `utillib.tlc`.

LibIsSFcnSingleRate(block)

Returns a Boolean value (1 or 0) indicating whether the S-function is single rate (one sample time) or multirate (multiple sample times).

See `LibIsSFcnSingleRate` in `utillib.tlc`.

LibIsSFcnSpecialSampleHit(sfcnSTI, sfcnTID)

Returns the Simulink macro to promote a slow task (`sfcnSTI`) into a faster task (`sfcnTID`).

This advanced function is specifically intended for use in rate transition blocks. `LibIsSFcnSpecialSampleHit` determines the global TID from the S-function TID

The input arguments to `LibIsSFcnSpecialSampleHit` are

- For multirate S-function blocks:

  `sfcnSTI`: local S-function sample time index (`sti`) of the slow task that is to be promoted

  `sfcnTID`: local S-function task ID (`tid`) of the fast task where the slow task is to run
- For single-rate S-function blocks using `SS_OPTION_RATE_TRANSITION`, `sfcnSTI` and `sfcnTID` are ignored and should be specified as `""`.

The format of `sfcnSTI` and `sfcnTID` must follow that of the argument to `LibIsSFcnSampleHit`.

Examples

- A rate transition S-function (one sample time with `SS_OPTION_RATE_TRANSITION`)

  ```
  if (%<LibIsSFcnSpecialSampleHit("","")>) {
  ```
- A multirate S-function with port-based sample times where the output rate is slower than the input rate (e.g., a zero-order hold operation)

  ```
  if (%<LibIsSFcnSpecialSampleHit("OutputPortIdx0","InputPortIdx0")>) {
  ```

See `LibIsSFcnSpecialSampleHit` in `utillib.tlc`.

LibIsSingleRateModel()

Returns `true` if model is single rate and `false` otherwise.

See `LibIsSingleRateModel` in `codetemplatelib.tlc`.

LibIsSingleTasking()

Returns `true` if the model is configured for single-tasking execution and `false` if the model is configured for multitasking execution.

See `LibIsSingleTasking` in `codetemplatelib.tlc`.

LibIsZOHContinuous(TID)

Returns 1 if the specified task identifier (TID) is zero order hold (ZOH) continuous and 0 otherwise. A TID equal to `triggered` or `constant` is not ZOH continuous.

See `LibIsZOHContinuous` in `utillib.tlc`.

LibNumAsynchronousSampleTimes()

Returns the number of asynchronous sample times in the model.

See `LibNumAsynchronousSampleTimes` in `codetemplatelib.tlc`.

LibNumDiscreteSampleTimes()

Returns the number of discrete sample times in the model.

See `LibNumDiscreteSampleTimes` in `codetemplatelib.tlc`.

LibNumSynchronousSampleTimes()

Returns the number of synchronous sample times in the model.

See `LibNumSynchronousSampleTimes` in `codetemplatelib.tlc`.

LibPortBasedSampleTimeBlockIsTriggered(block)

Determines whether the port-based S-function block is triggered.

See `LibPortBasedSampleTimeBlockIsTriggered` in `blocklib.tlc`.

LibSetVarNextHitTime(block, tNext)

Generates code to set the next variable hit time. Blocks with variable sample time must call `LibSetVarNextHitTime` in their output functions.

See `LibSetVarNextHitTime` in `blocklib.tlc`.

LibTriggeredTID(tid)

Returns whether this TID corresponds to a triggered rate.

See `LibTriggeredTID` in `utillib.tlc`.

See Also

Related Examples

- "Target Language Compiler Library Functions Overview" on page 22-2
- "Target Language Compiler Function Conventions" on page 22-3

Miscellaneous Functions

LibAddtoSystemCustomIncludes(system, incFileName)

For use by inlined S-functions within a subsystem. Adds `#include` file to the system header file.

Call this function from the "BlockInstanceSetup(block, system)" on page 21-21 method of the block so that different S-functions can include different header files. Specify your header file without any braces (for example, `"myInclude.h"`) and include the system header file in braces (for example, `"<sysinclude.h>"`).

Each call to this function adds the file to the list of included headers files only if it is not already included.

Example

```
%function BlockInstanceSetup(block, system) Output
  %assign incFileName = "sfun_custincl.h"
  %<LibAddToSystemCustomIncludes(system, incFileName)>
%endfunction

%function Outputs(block, system) Output
%endfunction
```

See `LibAddToSystemCustomIncludes` in `commonhdrlib.tlc`.

LibBlockExecuteFcnCall(block, callIdx)

For use by inlined S-functions with function-call outputs. Returns a string to call the function-call subsystem with required arguments or generates the subsystem's code in place (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCalls
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall,...
   "unconnected")
    %continue
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnCall(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnCall` in `syslib.tlc`.

LibBlockExecuteFcnDisable(block, callIdx)

For use by inlined S-Functions to call the function-call system disable function. Returns a string to either call the function-call subsystem disable function with required arguments or to call the generated subsystem disable code (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCallDsts
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall, "unconnected")
   %continue
```

```
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnDisable(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnDisable` in `syslib.tlc`.

LibBlockExecuteFcnEnable(block, callIdx)

For use by inlined S-Functions to call the function-call system enable function. Returns a string to call the function-call subsystem enable function with required arguments or the generated subsystem enable code (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCallDsts
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall, "unconnected")
   %continue
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnEnable(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnEnable` in `syslib.tlc`.

LibBlockInputSignalAliasedThruDataTypeId(idx)

Return the data type ID the input signal is aliased thru to.

See LibBlockInputSignalAliasedThruDataTypeId in `dtypelib.tlc`.

LibBlockOutputSignalAliasedThruDataTypeId(idx)

Return the data type ID the output signal is aliased thru to.

See `LibBlockOutputSignalAliasedThruDataTypeId` in `dtypelib.tlc`

LibGenConstVectWithInit(data, typeId, varId)

Returns an initialized static constant variable string of form:

```
static const typeName varId[] = { data };
```

The `typeName` is generated from `typeId`, which can be one of

```
tSS_DOUBLE, tSS_SINGLE, tSS_BOOLEAN, tSS_INT8, tSS_UINT8,
tSS_INT16, tSS_UINT16, tSS_INT32, tSS_UINT32
```

The data input argument must be a numeric scalar or vector and must be finite (no `Inf`, `-Inf`, or `NaN` values).

See `LibGenConstVectWithInit` in `utillib.tlc`.

LibGetBlockAttribute(block, attr)

Get a field value inside a block record.

Syntax

```
%if LibIsEqual(LibGetBlockAttribute(ssBlock,"MaskType"), ...
  "Task Block")
  %assign isTaskBlock = 1
%endif
```

Returns

Returns the value of the attribute (field) or an empty string if it does not exist.

See `LibGetBlockAttribute` in `asynclib.tlc`.

LibGetCallerClockTickCounter(sfcnBlock)

For use by asynchronous S-functions with function call outputs. Asynchronous tasks can manage their own time. `LibGetCallerClockTickCounter` is used to access an upstream asynchronous task's time counter. This is preferred when being driven by another asynchronous rate (e.g., an Interrupt block driving a Task block) because the time the interrupt occurred is used, rather than the time the task is allowed to run.

Returns

Returns a string for the counter variable for the upstream asynchronous task.

See `LibGetCallerClockTickCounter` in `asynclib.tlc`.

LibGetCallerClockTickCounterHigh(sfcnBlock)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. `LibGetCallerClockTickCounterHigh` is used to access the high word of an upstream asynchronous task's time counter. This is preferred when being driven by another asynchronous rate (for example, an Interrupt block driving a Task block) because the time the interrupt occurred is used rather than the time the Task is allowed to run.

Returns

Returns a string for the high word of the counter variable for the upstream asynchronous task.

See `LibGetCallerClockTickCounterHigh` in `asynclib.tlc`.

LibGetDataTypeComplexNameFromId(id)

Returns the name of the complex data type corresponding to a data type ID. For example, if `id==tSS_DOUBLE` then `LibGetDataTypeComplexNameFromId` returns `"creal_T"`.

See `LibGetDataTypeComplexNameFromId` in `dtypelib.tlc`.

LibGetDataTypeEnumFromId(id)

Returns the data type `enum` corresponding to a data type ID. For example, if `id==tSS_DOUBLE`, then `enum` is `"SS_DOUBLE"`. If `id` does not correspond to a built-in data type, `LibGetDataTypeEnumFromId` returns `""`.

See `LibGetDataTypeEnumFromId` in `dtypelib.tlc`.

LibGetDataTypeIdAliasedThruToFromId(id)

Returns the data type `IdAliasedThruTo` that corresponds to a data type ID. For example, if `yourfloat` is an alias to `myfloat`, and `myfloat` is an alias to `double`, then the `IdAliasedThruTo` for both `yourfloat` and `myfloat` is 0 (because the ID for `double` is 0).

See `LibGetDataTypeIdAliasedThruToFromId` in `dtypelib.tlc`.

LibGetDataTypeIdAliasedToFromId(id)

Returns the data type `IdAliasedTo` that corresponds to a data type ID. For example, if `yourfloat` is an alias to `myfloat`, and `myfloat` is an alias to `double`, then the `IdAliasedTo` for `yourfloat` is the ID for `myfloat`, and the `IdAliasedTo` for `myfloat` is 0 (because the ID for `double` is 0).

See `LibGetDataTypeIdAliasedToFromId` in `dtypelib.tlc`.

LibGetDataTypeIdResolvesToFromId(id)

Returns the data type `IdResolvesTo` that corresponds to a data type ID.

See `LibGetDataTypeIdResolvesToFromId` in `dtypelib.tlc`.

LibGetDataTypeNameFromId(id)

Returns the data type name that corresponds to a data type `ID`.

See `LibGetDataTypeNameFromId` in `dtypelib.tlc`.

LibGetDataTypeSLSizeFromId(id)

Return the size (as Simulink knows it) corresponding to a data type ID.

See `LibGetDataTypeSLSizeFromId` in `dtypelib.tlc`.

LibGetDataTypeStorageIdFromId(id)

Returns the data type `StorageId` corresponding to a data type ID. For example, if the input ID is the ID for a 16-bit signed fixed-point data type, then the `StorageId` corresponds to `int16`.

See `LibGetDataTypeStorageIdFromId` in `dtypelib.tlc`.

LibGetFcnCallBlock(sfcnblock,callIdx)

Given an S-function block and call index, return the block record for the downstream function-call subsystem block.

Syntax

```
%assign ssBlock = LibGetFcnCallBlock(block,0)
```

Returns

The block record of the downstream function-call subsystem connected to that element (call index).

See `LibGetFcnCallBlock` in `asynclib.tlc`.

LibGetRecordDataTypeId(rec)

Returns the data type identifier of the specified record as an integer.

See `LibGetRecordDataTypeId` in `dtypelib.tlc`.

LibGetRecordDimensions(rec)

Returns the dimensions of the specified record as a vector of integers.

See `LibGetRecordDimensions` in `dtypelib.tlc`.

LibGetRecordIsComplex(rec)

Returns the value 1 if the specified record is complex, and zero otherwise.

See `LibGetRecordIsComplex` in `dtypelib.tlc`.

LibGetRecordWidth(rec)

Returns the width of the specified record as an integer.

See `LibGetRecordWidth` in `dtypelib.tlc`.

LibGetT()

Returns a string to access the absolute time, which is a floating-point number. However, if you have configured the model for integer-only code generation (by deselecting **Support floating-point numbers**), the string represents an integer equal to the number of base rate ticks (the absolute time divided by the base sample time) rather than the absolute time.

You should use `LibGetT` to access time only.

`LibGetT` is the TLC version of the SimStruct macro `ssGetT`.

See `LibGetT` in `utillib.tlc`.

LibIsComplex(arg)

Returns `1` if the argument passed in is complex, `0` otherwise.

See `LibIsComplex` in `utillib.tlc`.

LibIsFirstInitCond()

`LibIsFirstInitCond` returns generated code intended for placement in the initialization function. This code determines, during run-time, whether the initialization function is being called for the first time.

`LibIsFirstInitCond` also sets a flag that tells the code generator if it needs to declare and maintain the `first-initialize-condition` flag.

`LibIsFirstInitCond` is the TLC version of the SimStruct macro `ssIsFirstInitCond`.

See `LibIsFirstInitCond` in `syslib.tlc`.

LibIsMajorTimeStep()

Returns a string to access whether the current simulation step is a major time step.

`LibIsMajorTimeStep` is the TLC version of the SimStruct macro `ssIsMajorTimeStep`.

See `LibIsMajorTimeStep` in `utillib.tlc`.

LibIsMinorTimeStep()

Returns a string to access whether the current simulation step is a minor time step.

`LibIsMinorTimeStep` is the TLC version of the SimStruct macro `ssIsMinorTimeStep`.

See `LibIsMinorTimeStep` in `utillib.tlc`.

LibManageAsyncCounter(sfcnBlock, callIdx)

For use by asynchronous S-Functions with function-call outputs. Asynchronous tasks can manage their own time, and use `LibManageAsyncCounter` to determine whether a need exists for an asynchronous counter to manage its own timer.

Example

```
%if LibManageAsyncCounter(block,callIdx)
  %%      %<LibSetAsyncCounter(block,callIdx), CodeGetCounter)>
```

where `CodeGetCounter` is target specific code reading hardware timer.

Returns

Returns `TLC_TRUE` if an asynchronous counter is required to manage its own counter, otherwise `TLC_FALSE`.

See `LibManageAsyncCounter` in `asynclib.tlc`.

LibMaxIntValue(dtype)

For a built-in integer data type, `LibMaxIntValue` returns the formatted maximum value of that data type.

See `LibMaxIntValue` in `dtypelib.tlc`.

LibMinIntValue(dtype)

For a built-in integer data type, `LibMinIntValue` returns the formatted minimum value of that data type.

See `LibMinIntValue` in `dtypelib.tlc`.

LibNeedAsyncCounter(sfcnBlock, callIdx)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks use `LibNeedAsyncCounter` to determine whether a need exists for an asynchronous counter.

Example

```
%if LibNeedAsyncCounter(block,0)
  %<LibSetAsyncCounter(block,0), "tickGet()")>
```

Returns

Returns `TLC_TRUE` if an asynchronous counter is required, otherwise `TLC_FALSE`.

See `LibNeedAsyncCounter` in `asynclib.tlc`.

LibSetAsyncClockTicks(sfcnBlock, callIdx, buf1, buf2)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncClockTicks` to return code that sets `clockTick` counters that are to be maintained by the asynchronous task. If the data type of a `clockTick` counter maintained by the asynchronous task is `tSS_TIMER_UINT32_PAIR`, the low and high word of the `clockTick` counter are set.

Arguments

`buf1` — Code that reads the low word of the hardware counter

`buf2` — Code that reads the high word of the hardware counter. Leave `buf2` empty if hardware counter length is less than 32 bits.

Returns

Returns code that sets `clockTick` counters that are to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block, callIdx)
%<LibSetAsyncCounter(block, 0, buf1, buf2)>
%endif
```

See `LibSetAsyncClockTicks` in `asynclib.tlc`.

LibSetAsyncCounter(sfcnBlock, callIdx, buf)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncCounter` to return code that sets a counter variable that is to be maintained by the asynchronous task.

Returns

Returns code that sets the counter variable that is to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block,0)
  %<LibSetAsyncCounter(block,0, "tickGet()")>
%endif
```

See `LibSetAsyncCounter` in `asynclib.tlc`.

LibSetAsyncCounterHigh(sfcnBlock, callIdx, buf)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncCounterHigh` to return code that sets the higher word of the counter variable that is to be maintained by the asynchronous task.

Returns

Returns code that sets the higher word of the counter variable that is to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block,0)
%<LibSetAsyncCounterHigh(block,0, "hightTickGet()")>
%endif
```

See `LibSetAsyncCounterHigh` in `asynclib.tlc`.

LibTIDInSystem(system, fcnType)

Purpose

Returns a task identifier (TID) if it is in the scope of a subsystem function and can be called before or during TLC generating code.

Syntax

`LibTIDInSystem(`*`system, fcnType`*`)`

Arguments

system

A record within the global CompiledModel record.

fcnType

One of the following: `'Output'`, `'Update'`, `'Outputupdate'`.

Description

This function returns a TID if it is in the scope of a subsystem function and can be called before or during TLC generating code. It returns the TID argument name, if a TID is passed as argument in the system function scope. If TID is not passed as argument in the scope, this function returns:

- '0' if model is single tasking.
- The TID value of the subsystem if the subsystem is single rate.
- A local TID variable name, if the subsystem is multirate. A local TID variable is added to the subsystem code.

Note This function issue an error message if it is called for a reusable subsystem whose instances run at different rates.

See `LibTIDInSystem` in `syslib.tlc`.

LibIsRowMajor

Returns true when the current model uses the row-major array layout.

See `LibIsRowMajor` in `dtypelib.tlc`.

Obsolete Functions

The following table shows obsolete functions and the functions that have replaced them.

Obsolete Function	Equivalent Replacement Function
`LibBlockOutportLocation`	`LibBlockDstSignalLocation`
`LibCacheGlobalPrmData`	Use the block function `Start`
`LibCacheIncludes`	`LibAddToCommonIncludes`
`LibContinuousState`	`LibBlockContinuousState`
`LibControlPortInputSignal`	`LibBlockSrcSignalLocation`
`LibConvertZCDirection`	Function is not used in code generation.
`LibDataInputPortWidth`	`LibBlockInputSignalWidth`
`LibDataOutputPortWidth`	`LibBlockOutputSignalWidth`

Obsolete Function	Equivalent Replacement Function
LibDefineIWork LibDefinePWork LibDefineRWork	IWork , PWork, and RWork names are now specified via the mdlRTW function in your C MEX S-function.
LibDiscreteState	LibBlockDiscreteState
LibExportFileCustomCode	LibSetSourceFileSection
LibExternalResetSignal	LibBlockInputSignal
LibHeaderFileCustomCode	LibSetSourceFileSection
LibIsEqual	Use built-in function ISEQUAL
LibMapSignalSource	FcnMapDataTypedSignalSource
LibMaxBlockIOWidth	Function is not used in Simulink Coder code generation.
LibMaxDataInputPortWidth	Function is not used in Simulink Coder code generation.
LibMaxDataOutputPortWidth	Function is not used in Simulink Coder code generation.
LibPathName	LibGetBlockPath, LibGetFormattedBlockPath
LibPrevZCState	LibBlockPrevZCState
LibPrmFileCustomCode	LibSetSourceFileSection
LibRegFileCustomCode	LibSetSourceFileSection
LibRegisterGNUMathFcnPrototypes	Function is not used in Simulink Coder code generation.
LibRegisterISOCMathFcnPrototypes	Function is not used in Simulink Coder code generation.
LibRegisterMathFcnPrototype	Function is not used in Simulink Coder code generation.
LibRenameParameter	Specifying parameter names is now supported via the mdlRTW function in your C MEX S-function.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Advanced Functions

LibAppendToModelReferenceUserData(data)

Appends the given data object to the user data in the `binfo` file for the model currently being built. This function is only called during builds for model reference targets.

Note The data argument cannot be a vector or matrix. To work around this limitation, create a record with a field containing the vector or matrix data and pass this record into the function.

See `LibAppendToModelReferenceUserData` in `modelrefutil.tlc`.

LibBlockInputSignalBufferDstPort(portIdx)

Returns the output port corresponding to input port (`portIdx`) that share the same memory, otherwise (-1) is returned. You will need to use `LibBlockInputSignalBufferDstPort` when you specify `ssSetInputPortOverWritable(S,portIdx,TRUE)` in your S-function.

If an input port and some output port of a block are

- Not test points, and
- The input port is overwritable

then the output port might reuse the same buffer as the input port. In this case, `LibBlockInputSignalBufferDstPort` returns the index of the output port that reuses the specified input port's buffer. If none of the block's output ports reuse the specified input port buffer, then `LibBlockInputSignalBufferDstPort` returns `-1`.

`LibBlockInputSignalBufferDstPort` is the TLC version of the Simulink macro `ssGetInputPortBufferDstPort`.

Example

Assume you have a block that has two input ports, both of which receive a complex number in 2-wide vectors. The block outputs the product of the two complex numbers.

```
%assign u1r = LibBlockInputSignal (0, "", "", 0)
%assign u1i = LibBlockInputSignal (0, "", "", 1)
%assign u2r = LibBlockInputSignal (1, "", "", 0)
%assign u2i = LibBlockInputSignal (1, "", "", 1)
%assign yr  = LibBlockOutputSignal (0, "", "", 0)
%assign yi  = LibBlockOutputSignal (0, "", "", 1)
%if (LibBlockInputSignalBufferDstPort(0) != -1)
  %% The first input is going to be overwritten by yr so
  %% we need to save the real part in a temporary variable.
  {
  real_T tmpRe = %<u1r>;
%assign u1r = "tmpRe";
%endif
%<yr> = %<u1r> * %<u2r> - %<u1i> * %<u2i>;
%<yi> = %<u1r> * %<u2i> + %<u1i> * %<u2r>;
%if (LibBlockInputSignalBufferDstPort(0) != -1)
```

```
  }
%endif
```

Note that, because only one output port exists, this example could have used `(LibBlockInputSignalBufferDstPort(0) == 0)` as the Boolean condition for the `%if` statements.

See `LibBlockInputSignalBufferDstPort` in `blkiolib.tlc`.

LibBlockInputSignalStorageClass(portIdx, sigIdx)

Returns the storage class of the specified block input port signal. The storage class can be `"Auto"`, `"ExportedSignal"`, `"ImportedExtern"`, or `"ImportedExternPointer"`.

See `LibBlockInputSignalStorageClass` in `blkiolib.tlc`.

LibBlockInputSignalStorageTypeQualifier(portIdx, sigIdx)

Returns the storage type qualifier of the specified block input port signal. The type qualifier can be anything entered by the user, such as `const`. The default type qualifier is `"Auto"`, which means do the default action.

See `LibBlockInputSignalStorageTypeQualifier` in `blkiolib.tlc`.

LibBlockOutputSignalIsGlobal(portIdx)

Returns 1 if the specified block output port signal is declared in the global scope, otherwise returns `0`.

If `LibBlockOutputSignalIsGlobal` returns `1`, then the variable holding this signal is accessible from anywhere in generated code. For example, `LibBlockOutputSignalIsGlobal` returns `1` for signals that are test points, external, or invariant.

See `LibBlockOutputSignalIsGlobal` in `blkiolib.tlc`.

LibBlockOutputSignalIsInBlockIO(portIdx)

Returns 1 if the specified block output port exists in the global block I/O data structure. You might need to use this if you specify `ssSetOutputPortReusable(S,portIdx,TRUE)` in your S-function.

See `sfun_multiport.tlc`.

See `LibBlockOutputSignalIsInBlockIO` in `blkiolib.tlc`.

LibBlockOutputSignalIsValidLValue(portIdx)

Returns `1` if the specified block output port signal can be used as a valid left-side argument (`lvalue`) in an assignment expression, otherwise returns `0`. For example, `LibBlockOutputSignalIsValidLValue` returns `1` if the block output port signal is in read/write memory.

See `LibBlockOutputSignalIsValidLValue` in `blkiolib.tlc`.

LibBlockOutputSignalStorageClass(portIdx)

Returns the storage class of the block's specified output signal. The storage class can be `"Auto"`, `"ExportedSignal"`, `"ImportedExtern"`, or `"ImportedExternPointer"`.

See `LibBlockOutputSignalStorageClass` in `blkiolib.tlc`.

LibBlockOutputSignalStorageTypeQualifier(portIdx)

Returns the storage type qualifier of the block's specified output signal. The type qualifier can be anything entered by the user, such as `const`. The default type qualifier is `Auto`, which means do the default action.

See `LibBlockOutputSignalStorageType` in `blkiolib.tlc`.

LibBlockSrcSignalBlock(portIdx, sigIdx)

Returns a reference to the block that is the source of the specified block input port element. The return argument is one of the following:

`[systemIdx, blockIdx]`	If the block output or state is unique
`"ExternalInput"`	If external input (root inport)
`"Ground"`	If unconnected or connected to ground
`"FcnCall"`	If function-call output
`0`	If not unique (i.e., source for a Merge block or a signal reused because of block I/O optimization)

Example

The following code fragment finds the block that drives the second input on the first port of the current block, then assigns the input signal of this source block to the variable `y`:

```
%assign srcBlock = LibBlockSrcSignalBlock(0, 1)
%% Make sure that the source is a block
%if TYPE(srcBlock) == "Vector"
  %assign sys = srcBlock[0]
  %assign blk = srcBlock[1]
  %assign block = CompiledModel.System[sys].Block[blk]
  %with block
    %assign u = LibBlockInputSignal(0, "", "", 0)
    y = %<u>;
  %endwith
%endif
```

See `LibBlockSrcSignalBlock` in `blkiolib.tlc`.

LibBlockSrcSignalIsDiscrete(portIdx, sigIdx)

Returns 1 if the source signal corresponding to the specified block input port element is discrete, otherwise returns 0.

Note that `LibBlockSrcSignalIsDiscrete` also returns 0 if the driving block cannot be uniquely determined to be a merged or reused signal (i.e., the source is a Merge block or the signal has been reused because of optimization).

See `LibBlockSrcSignalIsDiscrete` in `blkiolib.tlc`.

LibBlockSrcSignalIsGlobalAndModifiable(portIdx, sigIdx)

`LibBlockSrcSignalIsGlobalAndModifiable` returns 1 if the source signal corresponding to the specified block input port element satisfies the following three conditions:

- It is readable everywhere in the generated code.
- It can be referenced by its address.
- Its value can change (i.e., it is not declared as a `const`).

Otherwise, `LibBlockSrcSignalIsGlobalAndModifiable` returns `0`.

See `LibBlockSrcSignalIsGlobalAndModifiable` in `blkiolib.tlc`.

LibBlockSrcSignalIsInvariant(portIdx, sigIdx)

Returns `1` if the source signal corresponding to the specified block input port element is invariant (i.e., the signal does not change).

For example, a source block with a constant `TID` (or equivalently, an infinite sample time) would output an invariant signal.

See `LibBlockSrcSignalIsInvariant` in `blkiolib.tlc`.

LibGetModelReferenceUserData(modelName)

Gets the user data for the given model. This returns a vector with one element for each time `LibAppendToUserData` is called in the given model. This function cannot be called during builds where the target type is SIM.

See `LibGetModelReferenceUserData` in `modelrefutil.tlc`.

LibGetReferencedModelNames()

Gets the names of the models referenced by the model that is currently being built. Returns the data as a structure with two fields:

- `NumReferencedModels`: an integer with the number of model names

- `ReferencedModel`: an array of structures, where each structure has a field `–Name–` containing the name of a referenced model

See `LibGetReferencedModelNames` in `modelrefutil.tlc`.

LibIsModelReferenceRTWTarget()

Returns true if the build process is generating code for model reference target.

See `LibIsModelReferenceRTWTarget` in `utillib.tlc`.

LibIsModelReferenceSimTarget()

Return true if we are generating code for model reference Simulation target.

See `LibIsModelReferenceSimTarget` in `utillib.tlc`.

LibIsModelReferenceTarget()

Return true if we are generating code for model reference target.

See `LibIsModelReferenceTarget` in `utillib.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

Block Compatibility Functions

LibBlockSetIsExpressionCompliant(block)

Specify that a block's TLC file is compliant with expression folding. Call this function from within the `BlockInstanceSetup` function. The input to this function is the block record.

See `LibBlockSetIsExpressionCompliant(block)` in `block_api.tlc`.

LibBlockSetIntegerDowncastUnnecessary(block)

Specify that this block does not need to enforce integer downcasts for its output expression. The input to this function is the block record.

See `LibBlockSetIntegerDowncastUnnecessary(block)` in `block_api.tlc`.

LibBlockSetCustomStorageComplianceLevel (block, level)

Set the custom storage compliance level of a block.

The inputs to this function are the block record and the custom storage class compliance level. The possible levels are as follows:

- 0—The block does not support custom storage classes.
- 1— The block supports level 1 custom storage classes. Level 1 custom storage classes do not support the `set` method.
- 2—The block supports level 1 and 2 custom storage classes. Level 2 custom storage classes include level 1 custom storage classes and those that implement a `set` method.

See `LibBlockSetCustomStorageComplianceLevel(block,level)` in `block_api.tlc`.

See Also

Related Examples

- “Target Language Compiler Library Functions Overview” on page 22-2
- “Target Language Compiler Function Conventions” on page 22-3

A

TLC Error Handling

TLC Error Handling

Error Reporting

You might need to detect and report error conditions in your TLC code. Error detection and reporting are used most often in library functions. While rare, it is also possible to encounter error conditions in block target file code if the S-function `mdlCheckParameters` function does not detect an unforeseen condition.

To report an error condition detected in your TLC code, use the `LibBlockReportError` or `LibBlockReportFatalError` utility functions. Here is an example of using `LibBlockReportError` in the `paramlib.tlc` function `LibBlockParameter` to report the condition of an improper use of that function:

```
%if TYPE(param.Value) == "Matrix"
    %% exit if the parameter is a true matrix,
    %% i.e., has more than one row or columns.
    %if nRows > 1
      %assign errTxt = "Must access parameter %<param.Name> using "...
        "LibBlockMatrixParameter."
      %<LibBlockReportError([], errTxt)>
    %endif
  %endif
```

Browse through the files in the folder *`matlabroot`*`/rtw/c/tlc` (open) for more examples of the use of `LibBlockReportError`.

Generating Errors from TLC Files

TLC Error Generation Overview

To generate errors from TLC files, you can use the `%exit` directive. Alternatively, you can use one of the library functions described below that calls `%exit` for you. The two types of errors are

Usage errors	These can be caused by incorrect models.
Fatal (internal) TLC coding errors	These *cannot* be caused by incorrect models.

Usage Errors

Usage errors are errors resulting from incorrect models or attributes defined on a model. For example, suppose you have an S-Function block and an inline TLC file for a specific D/A device. If a model can contain only one copy of this S-function, then an error needs to be generated for a model that contains two copies of this S-Function block.

Using Library Functions

To generate usage errors related to a specific block, use the library function

```
LibBlockReportError(block,"error string")
```

The `block` argument is the block record if it isn't scoped. If the block is currently scoped, then you can specify `block` as [ ].

To generate general usage errors that are not related to a specific block, use

```
LibReportError("error string")
```

These library functions prefix the string `Simulink Coder Error:` to the message you provide when reporting the error.

For a usage example of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the TLC source folders within *`matlabroot`*`/rtw/c/tlc` (open.

Fatal (Internal) TLC Coding Errors

Suppose you have an S-function that has a local function that can accept only numerical numbers. You might want to add an assert requiring that the inputs be only numerical numbers. These asserts can indicate fatal coding errors for which the user does not have a way of building a model or specifying attributes that can cause the error to occur.

Using Library Functions

The two available library functions are

```
LibBlockReportFatalError(block,"fatal coding error message")
```

where *`block`* is the offending block record (or `[]` if the block is already scoped), and

```
LibReportFatalError("fatal coding error message")
```

for error messages that are not block specific. For example, to add assert code you could use

```
%if TYPE(argument) != "Number"
  %<LibBlockReportFatalError(block,"unexpected argument type")>
%endif
```

These library functions prefix the string `Simulink Coder Fatal:` to the message you provide and display the call stack when reporting the error.

For a usage example of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the folder *`matlabroot`*`/rtw/c/tlc` (open).

Using %exit

You can call `%exit` to generate fatal error messages. However, MathWorks suggests that you use one of the library functions described above.

When generating fatal error messages directly with `%exit`, it is good practice to give a stack trace with the error message. This lets you see the call chain of functions that caused the error. To generate a stack trace, generate the message using the format:

```
%setcommandswitch "-v1"
%exit Simulink Coder Fatal: error string
```

Formatting Error Messages

If you want to display a formatted, multiple-line error message, create a local variable that contains the message text. For example:

```
%openfile message
My message text
```

```
with newlines
%closefile message
```

After formatting your error message, use one of the error reporting library functions described above, such as `LibReportError`, to report your error when it occurs. For example:

```
%<LibReportError(message)>
```

The error reporting library functions provide an error message prefix, such as `Simulink Coder Error:`.

Testing Error Messages

It is strongly suggested that you test your error messages before releasing your new TLC code. To test your error messages, copy the relevant code into a `test.tlc` file and run

```
tlc test.tlc
```

at the MATLAB prompt.

Using TLC Error Messages to Troubleshoot

This section lists and describes error messages generated by the Target Language Compiler. Use this reference to

- Confirm that an error has been reported
- Determine possible causes for an error
- Determine possible ways to fix an error

%closefile or %selectfile or %flushfile argument must be a valid open file

In `%closefile` or `%selectfile` or `%flushfile`, the argument must be a valid file variable opened with `%openfile`.

%define no longer supported, use %function instead

Macros are not supported. You must rewrite macros as functions or inline them in your code.

%error directive: text

Code containing the `%error` directive generates this message. It normally indicates some condition that the code was unable to handle and displays the text following the `%error` directive.

%exit directive: text

Code containing the `%exit` directive causes this message. It typically indicates some condition that the code was unable to handle and displays the text following the `%exit` directive. Note that this directive causes the Target Language Compiler to terminate regardless of the `-mnumber` command-line option.

%filescope has already been used in this file

The user attempted to use the `%filescope` directive more than once in a file.

%trace directive: text

The `%trace` directive produces this error message and displays the text following the `%trace` directive. Trace directives are reported only when the `-v` option (verbose mode) appears on the command line. Note that `%trace` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

%warning directive: text

The `%warning` directive produces this error message and displays the text following the `%warning` directive. Note that `%warning` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

A %implements directive must appear within a block template file and must match the %language and type specified

A block template file was found, but it did not contain an `%implements` directive. An `%implements` directive is required so that the expected language and type are implemented by this block template file. See “Object-Oriented Facility for Generating Target Code” on page 19-23 for more information.

A %switch statement can only have one %default

The user has written a `%switch` statement with multiple `%default` cases, as in the following example:

```
%switch expr
  %case 1
    code...
    %break
  %default

more code...
    %break
  %default          %% error
    even more code...
    %break
%endswitch
```

A language choice must be made using the %language directive prior to using GENERATE or GENERATE_TYPE

To use the `GENERATE` or `GENERATE_TYPE` built-in functions, the Target Language Compiler requires that you first specify the language being generated. This causes the block-level target file to implement the same language and type as specified in the `%language` directive.

A non-homogeneous vector was passed to GENERATE_FORMATTED_VALUE

The built-in `GENERATE_FORMATTED_VALUE` can process only vectors that have homogeneous elements (that is, vectors in which all the elements have the same type).

Ambiguous reference to identifier — must use array index to refer to one of multiple scopes

In a repeated scope identifier from a database file, you must specify an index to disambiguate the reference. For example

```
Database file:
block
{
    Name                "Abc2"
    Parameter {
        Name            "foo"
        Value            2
    }
}
block
{
    Name                "Abc3"
    Parameter {
        Name            "foo"
        Value            3
    }
}
TLC file:
%<GETFIELD(block, "Name")>
```

In the preceding example, the reference to `block` is ambiguous because multiple repeated scopes named `block` appear in the database file. Use an index to disambiguate the references, as in:

```
%<GETFIELD(block[0], "Name")>
```

An %if statement can only have one %else

The user has written an `%if` statement with multiple `%else` blocks, as in the following example:

```
%if expr
  code...
%else
  more code...
%else          %% error
  even mode code...
%endif
```

Argument to identifier must be a string

The following built-in functions expect a string and report this error if the argument passed is not a string.

CAST	GENERATE_FILENAME
EXISTS	GENERATE_FUNCTION_EXISTS
FEVAL	GENERATE_TYPE
FILE_EXISTS	GET_COMMAND _SWITCH
FORMAT	IDNUM
GENERATE	SYSNAME

Arguments to directive must be records

Arguments to `%mergerecord` and `%copyrecord` must be records. Also, the first argument to the following built-in functions must be a record:

- `ISALIAS`
- `REMOVEFIELD`
- `FIELDNAMES`
- `ISFIELD`
- `GETFIELD`
- `SETFIELD`

Arguments to TLC from the MATLAB command line must be strings

An attempt was made to invoke the Target Language Compiler from MATLAB, but some of the arguments that were passed were not strings.

Assertion failed

An expression in an `%assert` statement evaluated to false.

Assignment to scope identifier is only allowed when using the + operator to add members

Scope assignment must be `scope = scope + variable`.

Attempt to define a function identifier on top of an existing variable or function

A function cannot be defined twice. Make sure that you don't have the same function defined in separate TLC files.

Attempt to divide by zero

The Target Language Compiler does not allow division by zero.

Bad cast - unable to cast this expression to type

The Target Language Compiler cannot cast this expression from its current type to the specified type. For example, the Target Language Compiler cannot cast a string to a number, as in

```
%assign x = "1234"
%assign y = CAST("Number", x );
```

Bad directory (dirname) in O: filename

The `-O` option did not specify a valid folder.

builtin was expecting expression of type type, got one of type type

A built-in was passed an expression of incorrect type.

Cannot %undef any builtin functions or variables

User is not allowed to undefine a TLC built-in or variable. For example

```
%undef FORMAT  %% error
```

Cannot convert string your_string to a number

Cannot convert the string to a number.

Changing value of identifier from the RTW file

You have overwritten the value that appeared in the `.rtw` file.

Error opening filename

The Target Language Compiler could not open the file specified on the command line.

Error writing to file error

There was an error while writing to the current output stream; `error` contains the system specific error message.

Errors occurred — aborting

This error message is the last error to be reported. It occurs when either

- The number of error messages exceeds the error message threshold (5 by default).
- Processing completes and errors have occurred.

Expansion directives %<> cannot be nested

It is illegal to nest expansion directives. For example,

```
%<foo(%<expr>)>
```

Instead, do the following:

```
%assign tmp = %<expr>
%<foo(tmp)>
```

Expansion directives %<> cannot span multiple lines; use \ at end of line

An expansion directive cannot span multiple lines. To work around this restriction, use the \ line continuation character. The following is incorrect:

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name +
"Hello">
```

Instead, use:

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name + \
"Hello">
```

Extra arguments to the function-name built-in function were ignored (Warning)

The following built-in functions report this warning when too many arguments are passed to them:

CAST	NUMTLCFILES
EXISTS	OUTPUT_LINES
FILE_EXISTS	SIZE
FORMAT	STRING
GENERATE_FILENAME	STRINGOF
GENERATE_FUNCTION_EXISTS	SYSNAME
IDNUM	TLCFILES
ISFINITE	TYPE
ISINF	WHITE_SPACE
ISNAN	WILL_ROLL

File name too long (directory =dirname, name =filename)

The specified *filename* was too long. The default limits are 256 characters for *filename* and 1024 characters for *dirname*, but the limits can be larger, depending on the platform.

format is not a legal format value

The specified format was not legal for the `%realformat` directive. Valid format strings are `"EXPONENTIAL"` and `"CONCISE"`.

Function argument mismatch; function function_name expects number arguments

When calling a function, too many or too few arguments were passed to it.

Function reached the end and did not return a value

Functions that are not declared as `void` or `Output` must return a value. If a return value is not desired, declare the function as `void`, otherwise make it return a value.

Function values are not allowed

Attempt to use a TLC function as a variable.

Identifier identifier multiply defined. Second and succeeding definitions ignored.

The user is attempting to add the same field to a record more than once, as in the following code.

```
%createrecord err { foo 1; rec { val 2 } }
%addtorecord err foo 2            %% error
```

Identifier identifier used on a %foreach statement was already in scope (Warning)

The argument to a `%foreach` statement cannot be defined prior to entering the `%foreach`.

Illegal use of eval (i.e., %<...>)

It is illegal to use evals in `.rtw` files. There are also some places where evals are not allowed in directives. For example:

```
%function %<foo>(a, b, c) void  %% error
%endfunction
```

Indices may not be negative

An index used in a [ ] expression must be a nonnegative integer.

Indices must be constant integral numbers

An index used in a [ ] expression must be an integer number.

Invalid handle

An invalid handle was passed to the Target Language Compiler server mode.

Invalid identifier range, the leading strings string1 and string2 must match

In a range of signals, for example, `u1:u10`, the identifier in the first argument did not match the identifier in the second.

Invalid identifier range, the lower bound (bound) must be less than the upper bound (bound)

In a range of signals, for example, `u1:u10`, the lower bound was higher than the upper bound.

Invalid type for unary operator

Unary operators − and + require numeric types. Unary operator − requires an integral type. Unary operator `!` requires a numeric type.

Invalid type type

An invalid type was passed to a built-in function.

It is illegal to return a function from a function

A function value cannot be returned from a function call.

Named value identifier already exists within this scope-identifier; use %assign to change the value

You cannot use the block addition operator + to add a value that is already a member of the indicated block. Use `%assign` to change the value of an existing value. This example produces this error:

```
%assign x = BLK { a 1; b 2 }
%assign a = 3
%assign x = x + a
```

Use this instead:

```
%assign x.a = 3
```

No %case statement(s) seen yet, statement ignored

Statements that appear inside a `%switch` statement but precede `%case` statements are ignored, as in the following code:

```
%switch expr
%assign x = 2  %% this statement will be ignored
  %case 1
    code
    %break
%endswitch
```

Only double and character arrays can be converted from MATLAB to TLC. This can occur if the MATLAB function does not return a value (see %matlab)

Only double and character arrays can be converted from MATLAB to the Target Language Compiler. This error can occur if the MATLAB function does not return a value (see `%matlab`). For example,

```
%assign a = FEVAL("int8",3)
%matlab disp(a)
```

Only one output is allowed from the TLC

An attempt was made to receive multiple outputs from the MATLAB version of the Target Language Compiler.

Only strings of length 1 can be assigned using the [ ] notation

The right-hand side of a string assignment using the [ ] operator must be a string of length 1. You can replace only a single character using this notation.

Only strings or cells of strings may be used as the argument to Query and ExecString

A cell containing nonstring data was passed as the third argument to `Query` or `ExecString` in server mode.

Only vectors of the same length as the existing vector value can be assigned using the [ ] notation

In the [ ] notation to replace a row of a matrix, the row must be a vector of the same length as the existing rows.

Output file identifier opened with %openfile was not closed

Output files opened with `%openfile` must be closed with `%closefile`. The *`identifier`* is the name of the variable specified in the `%openfile` directive.

Note This might also occur a syntax error is present in a code section between an `openfile` and `closefile`, or if you try to assign the output of a function of type `void` or `Output` to a variable.

Ranges, identifier ranges, and repeat values cannot be repeated

You cannot repeat a range, identifier range, or repeat value. This prevents things like `[1@2@3]`.

String cannot modify the setting for the command line switch '-switch'

`%setcommandswitch` does not recognize the specified switch, or cannot modify it (e.g., `-r` cannot be modified).

string is not a recognized user defined property of this handle

The query performed on a TLC server mode handle is looking for an undefined property.

Syntax error

The indicated line contains a syntax error, for more information on syntax, see “Target Language Compiler Directives” on page 19-2.

The %break directive can only appear within a %foreach, %for, %roll, or %switch statement

The `%break` directive can be used only in a `%foreach`, `%for`, `%roll`, or `%switch` statement.

The %case and %default directives can only be used within the %switch statement

A `%case` or `%default` directive can appear only within a `%switch` statement.

The %continue directive can only appear within a %foreach, %for, or %roll statement

The `%continue` directive can be used only in a `%foreach`, `%for`, or `%roll` statement.

The %foreach statement expects a constant numeric argument

The argument of a `%foreach` must be a numeric type. For example:

```
%foreach Index = [1 2 3 4]
...
%endforeach
```

`%foreach` cannot accept a vector as input.

The %if statement expects a constant numeric argument

The argument of an `%if` statement must be a numeric type. For example,

```
%if [ 1 2 3 ]
...
%endif
```

`%if` cannot accept a vector as input.

The %implements directive expects a string or string vector as the list of languages

You can use the `%implements` directive to specify a string for the language being implemented, or to indicate that it implements multiple languages by using a vector of strings. You cannot specify any other argument type to the `%implements` directive.

The %implements directive specifies type as the type where type was expected

The type specified in the `%implements` directive must exactly match the type specified in the block or on the `GENERATE_TYPE` directive. If you want to specify that the block accept multiple input types, use the `%implements *` directive, as in

```
%implements * "C"   %% I accept any type and generate C code
```

The %implements language does not match the language currently being generated (language)

The language or languages specified in the `%implements` directive must exactly match the `%language` directive.

The %return statement can only appear within the body of a function

A `%return` statement can be only in the body of a function.

The == and != operators can only be used to compare values of the same type

The == and != operator arguments must be the same type. You can use the `CAST()` built-in function to change them into the same type.

The argument for %openfile must be a valid string

When you open an output file, the name specified for the file must be a valid string.

The argument for %with must be a valid scope

The argument to `%with` must be a valid scope identifier. For example,

```
%assign x = 1
%with x
...
%endwith
```

In this code, the `%with` statement argument is a number and produces this error message.

The argument for an [ ] operation must be a repeated scope symbol, a vector, or a matrix

When you use the [ ] operator to index, the expression on the left of the brackets must be a vector, matrix, string, numeric constant, or a repeated scope identifier. When you use array indexing on a scalar, the constant is automatically scalar expanded and the value of the scalar is returned. For example,

```
%openfile x
%assign y = x[0]
```

This example causes this error because `x` is a file and is not valid for indexing.

The argument to %addincludepath must be a valid string

The argument to `%addincludepath` must be a string.

The argument to %include must be a valid string

The argument to the input file control directive must be a valid string with the filename given in double quotation marks.

The begin directive must be in the same file as the corresponding end directive.

These Target Language Compiler `begin` directives must appear in the same file as their corresponding `end` directives: `%function`, `%switch`, `%foreach`, `%roll`, and `%for`. Place the construct entirely within one Target Language Compiler source file.

The begin directive on this line has no matching end directive

For block-scoped directives, this error is produced if a matching `end` directive is missing. This error can occur for the following block-scoped Target Language Compiler directives.

Begin Directive	End Directive	Description
`%if`	`%endif`	Conditional inclusion
`%for`	`%endfor`	Looping
`%foreach`	`%endforeach`	Looping
`%roll`	`%endroll`	Loop rolling
`%with`	`%endwith`	Scoping directive
`%switch`	`%endswitch`	Switch directive
`%function`	`%endfunction`	Function declaration directive
`{`	`}`	Record creation

The error is reported on the line that opens the scope and does not have matching `end` scope.

Note Nested scopes must be closed before their parent scopes. Failure to include an `end` for a nested scope often causes this error, as in

```
%if Block.Name == "Sin 3"
    %foreach idx = Block.Width %endif
%% Error reported here that the %foreach was not terminated
```

The construct %matlab function_name(...) construct is illegal in standalone tlc

You cannot call MATLAB from stand-alone TLC.

The FEVAL() function can accept only 2-dimensional arrays from MATLAB, not number dimensions

Return values from MATLAB can have at most two dimensions.

The FEVAL() function can accept vectors of numbers or strings only when calling MATLAB

Vectors passed to MATLAB can be numbers or strings. See “FEVAL Function” on page 19-33.

The FEVAL() function requires the name of a function to call

`FEVAL` requires a function to call. This error appears only inside MATLAB.

The final argument to %roll must be a valid block scope

When you use `%roll`, the final argument (prior to extra user-specified arguments) must be a valid block scope. See “%roll” on page 19-22 for a description of this command.

The first argument of a ? : operator must be a Boolean expression

The `? :` operator must have a Boolean expression as its first operand.

The first argument to GENERATE or GENERATE_TYPE must be a valid scope

When you call `GENERATE` or `GENERATE_TYPE`, the first argument must be a valid scope. See the “GENERATE and GENERATE_TYPE Functions” on page 19-24 for more information and examples.

The function name requires at least number arguments

User is passing too few arguments to a function, as in the following code:

```
%function foo(a, b, c)
  %return a + b + c
%endfunction

%<foo(1, 2)>  %% error
```

The GENERATE function requires at least 2 arguments

When you call the `GENERATE` built-in function, the first two arguments must be the block and the name of the function to call.

The GENERATE_TYPE function requires at least 3 arguments

When you call the `GENERATE_TYPE` built-in function, the first three arguments must be the block, the name of the function to call, and the type.

The ISINF(), ISNAN(), ISFINITE(), REAL(), and IMAG() functions expect a real or complex valued argument

These functions expect a `Real` or complex value as the input argument.

The language being implemented cannot be changed within a block template file

You cannot change the language using the `%language` directive within a block template file.

The language being implemented has changed from old-language to new-language (Warning)

The language being implemented should not be changed in midstream because `GENERATE` function calls that appear prior to the `%language` directive can cause `generate` functions to load for the prior language. Only one language directive should appear in a given file.

The left-hand side of a . operator must be a valid scope identifier

When you use the `.` operator, the left-hand side of the `.` operator must be a valid in-scope identifier. For example:

```
%assign x = 1
%assign y = x.y
```

In this code, the reference to `x.y` produces this error message because `x` is not defined as a scope.

The left-hand side of an assignment must be a simple expression comprised of ., [ ], and identifiers

Illegal left-hand side of assignment.

The number of columns specified (specified-columns) did not match the actual number of columns in the rows (actual-columns)

When you specify a Target Language Compiler matrix, the number of columns specified must match the actual number of columns in the matrix. For example,

```
%assign mat = Matrix(2,1) [[1,2];[2,3]]
```

In this case, the number of columns in the declaration of the matrix (1) did not match the number of columns seen in the matrix (2). Either change the number of columns in the matrix, or change the matrix declaration.

The number of rows specified (specified-rows) did not match the actual number of rows seen in the matrix (actual-rows)

When you specify a Target Language Compiler matrix, the number of rows must match the actual number of rows in the matrix. For example,

```
%assign mat = Matrix(1,2) [[1,2];[2,3]]
```

In this case, the number of rows in the declaration of the matrix (1) did not match the number of rows seen in the matrix (2). Either change the number of rows in the matrix or change the matrix declaration.

The operator_name operator only works on Boolean arguments

The && and `||` operators work only on Boolean values.

The operator_name operator only works on integral arguments

The &, `^`, `|`, <<, >> and `%` operators work on numbers only.

The operator_name operator only works on numeric arguments

The arguments to the following operators both must be either numeric or real: <, <=, >, >=, `-`, `*`, `/`. This error can also occur when you use + as a unary operator. In addition, the `FORMAT` built-in function expects either a numeric or real argument.

The return value from the RollHeader function must be a string

When you use `%roll`, the `RollHeader()` function specified in `Roller.tlc` must return a string value. See "%roll" on page 19-22 for a complete discussion of the `%roll` construct.

The roll argument to %roll must be a nonempty vector of numbers or ranges

When you use `%roll`, the `roll` vector cannot be empty and must contain numbers or ranges of numbers. See "%roll" on page 19-22 for a complete discussion of the `%roll` construct.

The second value in a Range must be greater than the first value

In a range, for example, `1:10`, the lower bound was higher than the upper bound.

The specified index (index) was out of the range 0 - number-of-elements - 1

This error occurs when you index into a nonscalar beyond the end of the variable. For example:

```
%assign x = [1 2 3]
%assign y = x[3]
```

This example would cause this error. Remember, in the Target Language Compiler, array indices start at 0 and go to the number of elements minus 1.

The STRINGOF built-in function expects a vector of numbers as its argument

The `STRINGOF` function expects a vector of numbers. The function treats each number as the ASCII value of a valid character.

The SYSNAME built-in function expects an input string of the form xxx/yyy

The `SYSNAME` function takes a single string of the form `xxx/yyy` as it appears in the `.rtw` file and returns a vector of two strings, `xxx` and `yyy`. If the input argument does not match this format, `SYSNAME` returns this error.

The threshold on a %roll statement must be a single number

When you use `%roll`, the roll threshold specified must be a single number. See “%roll” on page 19-22 for a complete discussion of the `%roll` construct.

The use of feature is being deprecated and will not be supported in future versions of TLC. See the TLC manual for alternatives.

The `%define` and `%generate` directives are not recommended, as they are being replaced.

The WILL_ROLL built in function expects a range vector and an integer threshold

The `WILL_ROLL` function requires both arguments cited in the message.

There are no more free contexts. Use tlc('close', HANDLE) to free up a context

The global context table has filled up while the TLC server mode is in use.

There was no type associated with the given block for GENERATE

The scope specified to `GENERATE` must include a `Type` parameter that indicates which template file should be used to generate code for the specified scope. For example:

```
%assign scope = block { Name "foo" }
%<GENERATE( scope, "Output" )>
```

This example produces the error message because the scope does not include the parameter `Type`. See the “GENERATE and GENERATE_TYPE Functions” on page 19-24 for more information and examples.

This assignment would overwrite an identifier-value pair from the RTW file. To avoid this error either qualify the left-hand side, or choose another identifier.

The user is trying to modify a field of a record in a `%with` block without qualifying the left-hand side, as in this example:

```
  %createrecord foo { field 1 }
  %with foo
    %assign field = 2  %% error
  %endwith
```

Instead, use

```
%createrecord foo { field 1 }
  %with foo
    %assign foo.field = 2
  %endwith
```

TLC has leaked number symbols. You may have created a cyclic record. If this not the case then please report this leak to The MathWorks.

There has been a memory leak while running TLC. The most common cause of this is having cyclic records.

Unable to find identifier within the scope-identifier scope

The given identifier was not found in the scope specified. For example,

```
%assign scope = ascope { x 5 }
%assign y = scope.y
```

In this code, the reference to `scope.y` produces this error message.

Unable to open %include file filename

The file included in an `%include` directive was not found on the path. Either move the file to a location on the current path, or use the `-I` command-line option or the `%addincludepath` directive to specify the folder that contains the file.

Unable to open block template file filename from GENERATE or GENERATE_TYPE

You specified `GENERATE` but the given filename was not found on the Target Language Compiler path. You can

- Add the file to a folder on the path.
- Use the `%generatefile` directive to specify an alternative filename for this block type that is on the path.

- Add the folder in which this file appears to the search path using the `-I` command-line option or the `%addincludepath` directive.

Unable to open output file filename

The specified output file could not be opened. Either an invalid filename was specified or the file was read only.

Undefined identifier identifier_name

The identifier specified in this expression was undefined.

Unknown type type in CAST expression

When you call the `CAST` built-in function, the type must be a valid type from the “Target Language Value Types” on page 19-13 table.

Unrecognized command line switch passed to string: switch

You queried the current state of a switch, but the switch specified was not recognized.

Unrecognized directive directive-name seen

An illegal `%` directive was encountered. The valid directives are shown below.

`%addincludepath`	`%addtorecord`
`%assert`	`%assign`
`%break=`	`%case`
`%closefile`	`%continue`
`%copyrecord`	`%createrecord`
`%default`	`%define`
`%else`	`%elseif`
`%endbody`	`%endfor`
`%endforeach`	`%endfunction`
`%endif`	`%endroll`
`%endswitch`	`%endwith`
`%error`	`%exit`
`%filescope`	`%for`
`%foreach`	`%function`
`%generate`	`%generatefile`
`%if`	`%implements`
`%include`	`%language`
`%matlab`	`%mergerecord`

`%openfile`	`%realformat`
`%return`	`%roll`
`%selectfile`	`%setcommandswitch`
`%switch`	`%trace`
`%undef`	`%warning`
`%with`	

Unrecognized type output-type for function

The function type modifier was not `Output` or `void`. For functions that do not produce output, the default without a type modifier indicates that the function should not produce output.

Unterminated multiline comment.

A multiline comment (i.e., /% %/) does not have a terminator, as in the following code:

```
/% my comment

%assign x = 2
%assign y = x * 7
```

Unterminated string

A string must be closed prior to the end of an expansion directive or the end of a line.

Usage: tlc [options] file

A command-line problem has occurred. The error message contains a list of the available options.

Use of feature incurs a performance hit, please see TLC manual for possible workarounds.

The `%undef` and expansion (i.e., `%<expr>`) features can degrade performance.

Value of type type cannot be compared

Values of the specified *`type`* cannot be compared.

Values of specified_type type cannot be expanded

The specified type cannot be used on an expansion directive. Files and scopes cannot be expanded. This error can also occur when you expand a function without any arguments. If you use

```
%<Function>
```

call it with the required arguments.

Values of type Special, Macro Expansion, Function, File, Full Identifier, and Index cannot be converted to MATLAB variables

Values of the types listed in the message cannot converted to MATLAB variables.

When appending to a buffer stream, the variable must be a string

You can specify the `append` option for a buffer stream only if the variable currently exists as a string. Do not use the `append` option if the variable does not exist or is not a string. This example produces this error.

```
%assign x = 1
%openfile x , "a"
%closefile x
```

TLC Function Library Error Messages

The functions in the TLC function library can generate many error messages that are not documented. These messages are sufficiently self-descriptive so that they do not need additional explanation. However, if you encounter an error message that does not provide enough description to resolve your problem, contact our technical support staff.

See Also

Related Examples

- “Target Language Compiler Directives” on page 19-2

Guidelines and Standards for Embedded Coder

- “Support for Standards and Guidelines” on page 23-2
- “Modeling Guidelines for Generated Code” on page 23-5
- “Industry Standards” on page 23-7
- “Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards” on page 23-11
- “Qualification of Embedded Coder and AUTOSAR Blockset for Use in Processes that Must Comply with Industry Standards” on page 23-13
- “Coding Standards” on page 23-14
- “Model Advisor Checks for Verifying Compliance with Secure Coding Standards” on page 23-16
- “Develop a Model That Complies with the AUTOSAR Standard” on page 23-17
- “Develop a Model That Complies with the IEC 61508 Standard” on page 23-24
- “CERT C Compliance Summary Tables” on page 23-26

Support for Standards and Guidelines

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

Standard or Guidelines	Organization	For More Information, See...
MathWorks Advisory Board (MAB) Guidelines	MathWorks with NA-MAAB standards groups	• "MAB Modeling Guidelines" • "Model Advisor Checks for MAB Modeling Guidelines" (Simulink Check)
Japan MathWorks Automotive Advisory Board (JMAAB) Modeling Guidelines	Japan MathWorks Automotive Advisory Board standards group	• *Version 5.1 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow, and* • *Version 6.0 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow* • "Model Advisor Checks for JMAAB Modeling Guidelines" (Simulink Check)
Use of the C Language in Critical Systems (MISRA C™)	Motor Industry Software Reliability Association (MISRA)	• MISRA C website • Develop Models and Code that Comply with "MISRA C" on page 23-14 • MISRA C
CERT® C Coding Standards	CERT Division of the Software Engineering Institute (SEI)	• CERT C Coding Standards publication • Secure Coding on the Software Engineering Institute website • Develop Model and Code that Comply with "Modeling Guidelines for Generated Code" on page 23-5
Common Weakness Enumeration (CWE™)	The MITRE Corporation	• CWE list of software weaknesses types • Develop Model and Code that Comply with "Modeling Guidelines for Generated Code" on page 23-5

Standard or Guidelines	Organization	For More Information, See...
ISO/IEC TS 17961, *Information Technology — Programming Languages, Their Environments And System Software Interfaces — C Secure Coding Rules*	International Organization for Standardization and International Electrotechnical Commission	• ISO/IEC TS 17961:2013 standards publication • Develop Model and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5
AUTomotive Open System ARchitecture (AUTOSAR)	AUTOSAR Development Partnership	• Publications and specifications available from the AUTOSAR website • AUTOSAR Support from Embedded Coder on the MathWorks website • “AUTOSAR Blockset” documentation
IEC 61508, *Functional safety of electrical/electronic/ programmable electronic safety-related systems*	International Electrotechnical Commission	• IEC 61508 • IEC functional safety website • IEC 61508 Support in MATLAB and Simulink • Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5
Standard: IEC 62304, *Medical device software — Software life cycle processes*	International Electrotechnical Commission	• IEC 62304 • Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5
ISO 26262, *Road Vehicles — Functional Safety*	International Organization for Standardization	• ISO 26262 • ISO 26262 Support in MATLAB and Simulink • Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5
EN 50128, *Railway applications — Software for railway control and protection systems*	European Committee for Electrotechnical Standardization	• EN 50128/EN 50657 • Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5
ISO 25119, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems*	International Organization for Standardization	• Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5

Standard or Guidelines	Organization	For More Information, See...
DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*	Radio Technical Commission for Aeronautics (RTCA)	• DO-178C/DO-331 • Develop Models and Code that Comply with “Modeling Guidelines for Generated Code” on page 23-5

Modeling Guidelines for Generated Code

Use modeling guidelines when you develop models and generate code for embedded systems using Model-Based Design with MathWorks products. The guidelines provide model setting, block usage, and parameter considerations that impact code generation.

Code Generation Modeling Guidelines

You can use code generation guidelines when developing models and generating code for embedded systems. The guidelines provide guidance on modeling patterns, block usage, and configuration parameter settings that impact code generation and component model deployment.

Guidelines for blocks include:

- “cgsl_0101: Zero-based indexing”
- “cgsl_0102: Evenly spaced breakpoints in lookup tables”
- “cgsl_0103: Precalculated signals and parameters”
- “cgsl_0201: Redundant Unit Delay and Memory blocks”

Guidelines for modeling patterns include:

- “cgsl_0202: Usage of For, While, and For Each subsystems with vector signals”
- “cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks”

Guidelines for configuration parameters include:

- “cgsl_0301: Prioritization of code generation objectives for code efficiency”

Guidelines for component deployment using a service interface configuration:

- “cgsl_0401: Modeling styles for component deployment”
- “cgsl_0402: Signal interfaces for component deployment”
- “cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment”
- “cgsl_0405: Data receive for component deployment”
- “cgsl_0406: Data send for component deployment”
- “cgsl_0408: Partial data send for component deployment”
- “cgsl_0409: Data transfer for component deployment”
- “cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks”
- “cgsl_0413: Reuse memory between component state and output for component deployment”
- “cgsl_0414: Configure service interface for component model”

Fort information about verifying the readiness of your model for code generation, see “Check Model and Configuration for Code Generation” on page 49-8.

See Also

"Check Your Model Using the Model Advisor" | "Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards" on page 23-11 | "Model Advisor Checks for Verifying Compliance with Secure Coding Standards" on page 23-16

Industry Standards

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards, including:

- “IEC 61508: 2010” on page 23-7
- “ISO 26262: 2018” on page 23-7
- “AUTOSAR Standard” on page 23-9
- “IEC 62304: 2015” on page 23-7
- “EN 50128 / EN 50128: 2011” on page 23-8
- “ISO 25119 Standard” on page 23-8

If you have a Simulink Check product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with the industry standard. For more information, see “Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards” on page 23-11.

IEC 61508: 2010

Development of high-integrity systems within the automotive industry is characterized by demonstrating compliance with IEC 61508: 2010, *Functional safety of electrical/electronic/programmable electronic safety related systems*. This standard addresses E/E systems[2] in commercial vehicles – especially off-highway vehicles. IEC 61508: 2010 was published when most software was coded by hand, therefore the standard needs to be mapped to Model-Based Design technologies. For further information about MathWorks support for IEC 61508, see IEC 61508 Support in MATLAB and Simulink.

ISO 26262: 2018

Development of high-integrity systems within the automotive industry is characterized by demonstrating compliance with ISO 26262. ISO 26262: 2018 *Road vehicles — Functional safety*. This standard, which is an adaptation of IEC 61508, addresses *E/E systems* in passenger road vehicles.

For additional information about MathWorks support for ISO 26262, see ISO 26262 Support in MATLAB and Simulink.

IEC 62304: 2015

IEC 62304: 2015, *Medical device software - Software life cycle processes* is an international standard that defines the requirements of the software life-cycle processes for medical device software development. The standard specifies a set of processes, activities, and tasks that establish a common framework for designing safe and tested software for medical devices. For more information about MathWorks support for IEC 62304, see MATLAB and Simulink for Medical Devices.

If you have a Simulink Check product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the IEC 62304 standard. For

2 Systems that consists of electrical and electronic elements, including: programmable electronic elements, power supplies, input devices, communication paths, and output devices.

more information, see “Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Standards Compliance” (Simulink Check).

EN 50128 / EN 50128: 2011

High-integrity systems developed within the railway industry need to demonstrate compliance with:

- EN 50128: 2011, *Railway applications — Software for railway control and protection systems*
- EN 50657:2017, *Railways Applications - Rolling stock applications - Software on Board Rolling Stock*

These European standards specify procedures and technical requirements for the development of programmable electronic systems for use in railway control and protection applications. For more information about MathWorks support for EN 50128, see EN 50128/EN 50657.

If you have a Simulink Check product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the EN 50128 standard. For more information, see “Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Standards Compliance” (Simulink Check).

ISO 25119 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined functional safety standards. ISO 25119, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems*, is such a standard.

MathWorks provides an IEC Certification Kit product that you can use to qualify MathWorks code generation and verification tools for projects based on the ISO 25119 standard. For more information, see “Qualification of Embedded Coder and AUTOSAR Blockset for Use in Processes that Must Comply with Industry Standards” on page 23-13.

If you have a Simulink Check product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the ISO 25119 standard. For more information, see “Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Standards Compliance” (Simulink Check).

DO-178C Standard

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

For information about Model-Based Design and MathWorks support of aerospace and defense industry standards, see DO Qualification Kit.

If you have a Simulink Check product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the DO-178C standard. For

more information, see “Model Advisor Checks for DO-178C/DO-331 Standards Compliance” (Simulink Check).

AUTOSAR Standard

Simulink software supports *AUTomotive Open System ARchitecture* (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components.

The AUTOSAR standard addresses:

- Architecture – Application, run-time environment, and service layers, which serve to decouple AUTOSAR software components from the execution platform. Standard interfaces between software components and the run-time environment allow reuse or relocation of components within the Electronic Control Unit (ECU) topology of a vehicle.

  The standard defines two AUTOSAR platforms:

  - Classic Platform (CP), with Application, Runtime Environment (RTE), and Basic Software (BSW) layers
  - Adaptive Platform (AP), with Application, AUTOSAR Runtime for Adaptive Applications (ARA), Services, and Basis layers
- Methodology – Specification of code formats and description file templates, for example.
- Foundation – Requirements and specifications shared between AUTOSAR platforms, supporting platform interoperability.
- Application Interfaces – Specification of interfaces for typical automotive applications.

For more information, see:

- `www.autosar.org` for details on the AUTOSAR standard.
- “AUTOSAR Blockset” for information on modeling and simulating AUTOSAR software, from which Embedded Coder can generate code.
- MATLAB and Simulink for AUTOSAR to learn about using MathWorks products and third-party tools for AUTOSAR.

If you have an AUTOSAR Blockset product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the AUTOSAR standard. For more information, see “AUTOSAR Blockset Checks” (AUTOSAR Blockset).

See Also

“Coding Standards” on page 23-14 | “Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards” on page 23-11

External Websites

- International Organization for Standardization
- International Electrotechnical Commission
- IEC functional safety website
- European Committee for Electrotechnical Standardization

- AUTOSAR Development Partnership
- Radio Technical Commission for Aeronautics (RTCA)

Modeling Guidelines and Model Advisor Checks for Verifying Compliance with Industry Standards

Use modeling guidelines when you develop models and generate code for embedded systems that are intended for use in projects that must comply with industry standards.

High-Integrity System Modeling Guidelines

“High-Integrity System Modeling” guidelines provide model setting, block usage, and block parameter considerations for creating models that are complete, unambiguous, robust, and verifiable. Use these guidelines when you develop models and generate code for safety-critical systems, such as in projects that need to comply with the DO-178C / DO-331, IEC 61508, IEC 62304, ISO 26262, or EN 50128 industry standards.

If you have a Simulink Check product license, you can use the Model Advisor to check for compliance with the high-integrity modeling guidelines. To check your model or subsystem:

1. Open the Model Advisor.
2. Navigate to **By Product** > **Simulink Check** > **Modeling Standards** > **<Standard> Checks**.
3. Run the checks in the folder.

To review the High-Integrity System Modeling guidelines and their corresponding Model Advisor checks, see “Model Advisor Checks for High Integrity Systems Modeling Guidelines” (Simulink Check).

MAB Modeling Guidelines

MathWorks Advisory Board (MAB) guidelines stipulate important basic rules for modeling in Simulink and Stateflow. The overall purpose of these modeling guidelines is to allow for a simple, common understanding by modelers and consumers of control system models. The main objectives of these guidelines are readability, simulation and verification, and code generation.

If you have a Simulink Check license, you can check that your Simulink model or subsystem, and the code that you generate from it, complies with MAB guidelines. For more information, see “Model Advisor Checks for MAB Modeling Guidelines” (Simulink Check)

JMAAB Modeling Guidelines

Similar to MAB modeling guidelines, the Japan MathWorks Automotive Advisor Board (JMAAB) modeling guidelines stipulate rules for modeling in Simulink and Stateflow. The overall purpose of these modeling guidelines is to allow for a simple, common understanding by modelers and consumers of control system models. The main objectives of these guidelines are readability, simulation and verification, and code generation.

The JMAAB modeling guidelines are published in:

- *Version 5.1 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow, and*
- *Version 6.0 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow*

If you have a Simulink Check license, you can check that your Simulink model or subsystem, and the code that you generate from it, complies with JMAAB guidelines. For more information, see “Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

See Also

“Check Your Model Using the Model Advisor” | “Industry Standards” on page 23-7

External Websites

- MathWorks Advisory Board (MAB) Guidelines
- Version 5.1 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow
- Version 6.0 — Control Algorithm Modeling Guidelines - Using MATLAB, Simulink, and Stateflow

Qualification of Embedded Coder and AUTOSAR Blockset for Use in Processes that Must Comply with Industry Standards

You can use the IEC Certification Kit to qualify Embedded Coder for use in development processes that must comply with IEC 61508, ISO 26262, EN 50128, EN 50657, ISO 25119, ISO 21434, and related functional-safety standards such as IEC 62304. Qualification covers the code generator that is used to transform an executable model (model used for production code generation) into production C/C++ code for application software components.

You can qualify AUTOSAR Blockset when using the Embedded Coder code generator to generate code that is suitable for AUTOSAR platforms, including:

- Transformation of an executable model into production quality C/C++ code and files for AUTOSAR application software components.
- Creation of an AUTOSAR configuration for a model, model AUTOSAR elements, and generate AUTOSAR XML and AUTOSAR-compatible C/C++ code from a model.

To streamline the certification of your embedded systems, the IEC Certification Kit provides:

- Tool qualification artifacts, which provide software development and verification workflows, tool use cases, and error mitigation methods.

  Test suites, which you can use to validate tool usage and generated code.
- Functionality to generate bidirectional traceability matrices between the model and generated code. For more information, see “Validate Traceability” on page 23-13.

For more information about the IEC Certification Kit, see “IEC Certification Kit (for ISO 26262 and IEC 61508)”.

Validate Traceability

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

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Management Interface Setup” (Requirements Toolbox) that is available if you have a Requirements Toolbox™ license.
Trace model blocks and subsystems to generated code	The “Model-to-Code Traceability” on page 85-10 option when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The “Code-to-Model Traceability” on page 85-8 option when generating an HTML report during the code generation or build process.

Coding Standards

Software coding standards play an important role in preventing unreliable programming constructs in software development processes. For embedded systems, important coding standards include:

- “MISRA C” on page 23-14
- “Secure Coding” on page 23-14

The ability to check whether your model and code complies with coding standards can help to prevent security vulnerabilities. Embedded Coder and Simulink offer capabilities to minimize issues that affect compliance with MISRA C:2012 and secure coding standards. Capabilities include:

- Code Generation Advisor, which helps you configure a model or subsystem so that the code generator is most likely to produce MISRA C:2012 compliant code. For more information, see “Configure Model for Code Generation Objectives by Using Code Generation Advisor” on page 49-2.
- Model Advisor checks, which you can use as you developed your model or subsystem to increase the likelihood of generating code that complies with MISRA C:2012 and secure coding standards. For more information, see:
  - “Model Advisor Checks for Verifying Compliance with Secure Coding Standards” on page 23-16
  - “Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012” on page 24-20

MISRA C

The MISRA C and MISRA C++ standards are a set of coding guidelines for the C and C++ programing languages that promote safety, security, and reliability in embedded system software. These guidelines, published by the Motor Industry Software Reliability Association (MISRA) define a “safe-subset” of the C language to protect against language aspects that can compromise the safety and security of embedded systems. For more information, see What is MISRA C?

When using MISRA C:2012 coding guidelines to evaluate the quality of your generated C code, you are required per section 5.3 of the *MISRA C:2012 Guidelines for the Use of C Language in Critical Systems* document to prepare a compliance statement for the project being evaluated. To assist you in the development of this compliance statement, MathWorks evaluates the MISRA C:2012 guidelines against C code generated by using Embedded Coder. The results of the evaluation are published as:

- Compliance Summary Tables on page 24-6, which identify the method used to obtain compliance for each rule and directive.
- Deviations on page 24-27, which identify rules or directives that are not compliant.

For more information, see “Developing a MISRA C:2012 Compliance Statement” on page 24-2.

Secure Coding

CERT C, ISO/IEC 17961, and CWE are coding rule guidelines developed specifically to address the growing number of cyber security concerns within embedded systems. These coding standards are for software developers to use in the development of code in the C language:

- CERT C — Provides secure coding practices for the C languages. Security vulnerabilities in embedded software increase chances of attacks from malicious actors. These guidelines help eliminate constructs with undefined behavior that can lead to unexpected results at runtime and expose security weaknesses. For more information, see What is CERT C?
- Common Weakness Enumeration (CWE™) — Identifies common software weakness types that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.
- ISO/IEC TS 19761:2013 *Information Technology — Programming Languages, Their Environments And System Software Interfaces — C Secure Coding Rules* — The formal ISO® standard for secure coding in C. This standard is designed to be enforced by using static analysis tools without excessive false positives.

See Also

"Industry Standards" on page 23-7 | "Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012" on page 24-20 | "Model Advisor Checks for Verifying Compliance with Secure Coding Standards" on page 23-16

External Websites

- MISRA C
- CERT Division of the Software Engineering Institute (SEI)
- CERT C Coding Standards
- Secure Coding
- Common Weakness Enumeration
- The MITRE Corporation
- ISO/IEC TS 19761:2013
- International Organization for Standardization
- International Electrotechnical Commission

Model Advisor Checks for Verifying Compliance with Secure Coding Standards

If you have Embedded Coder or Simulink Check, you can check that your Simulink model, and the complied code, conforms to the secure coding standards. To check your model:

1. Open the Model Advisor.
2. Navigate to **By Task > Modeling Guidelines for Secure Coding (CERT C, CWE, ISO/IEC TS 17961)** .
3. Run the checks in the folder.

To review the Model Advisor checks, see “Model Advisor Checks for CERT C, CWE, and ISO/IEC TS 17961 Secure Coding Standards” (Simulink Check).

If you have a Polyspace® Bug Finder™ product license, you can evaluate your code against these secure coding standards. For more information, see:

- “Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers” (Polyspace Bug Finder)
- “Check for and Review Coding Standard Violations” (Polyspace Bug Finder)

See Also

“Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012” on page 24-20 | “Coding Standards” on page 23-14

Develop a Model That Complies with the AUTOSAR Standard

Generate AUTOSAR-compliant C code and export AUTOSAR XML (ARXML) descriptions from a Simulink® model.

AUTOSAR Blockset software supports AUTomotive Open System ARchitecture (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components. To develop AUTOSAR components in Simulink, follow this general workflow:

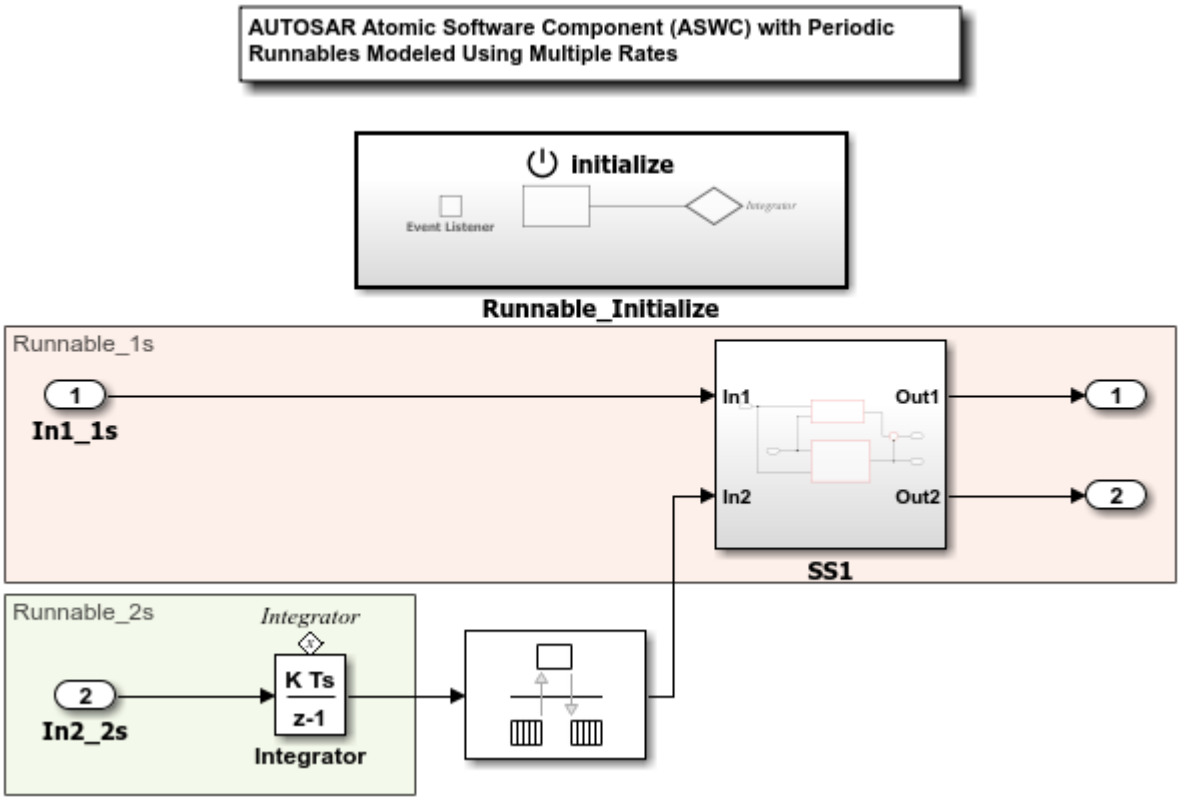
1. Create a Simulink representation of an AUTOSAR component.
2. Develop the component by refining the AUTOSAR configuration and creating algorithmic model content.
3. Generate ARXML descriptions and algorithmic C or C++ code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder™ and Embedded Coder®.)

Prepare Model for AUTOSAR Code Generation

To see the steps for generating AUTOSAR-compliant C code and exporting ARXML descriptions from an AUTOSAR model, open a model and prepare the model for AUTOSAR code generation.

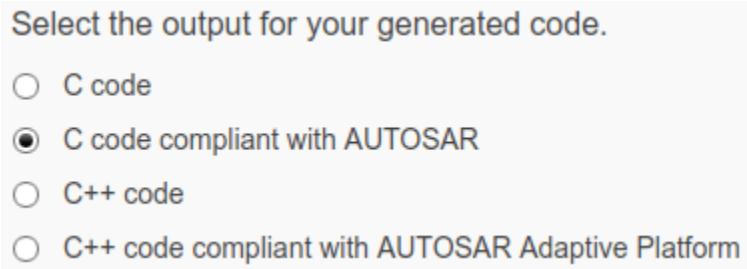
Open a model from which you want to generate AUTOSAR code and descriptions. The model can be unconfigured or only partially configured for code generation. This example uses AUTOSAR example model `autosar_swc`.

```
open_system('autosar_swc.slx');
```



To prepare the model for AUTOSAR code generation, use the Embedded Coder Quick Start. From the **Apps** tab, open the AUTOSAR Component Designer app. On the **AUTOSAR** tab, click **Quick Start**.

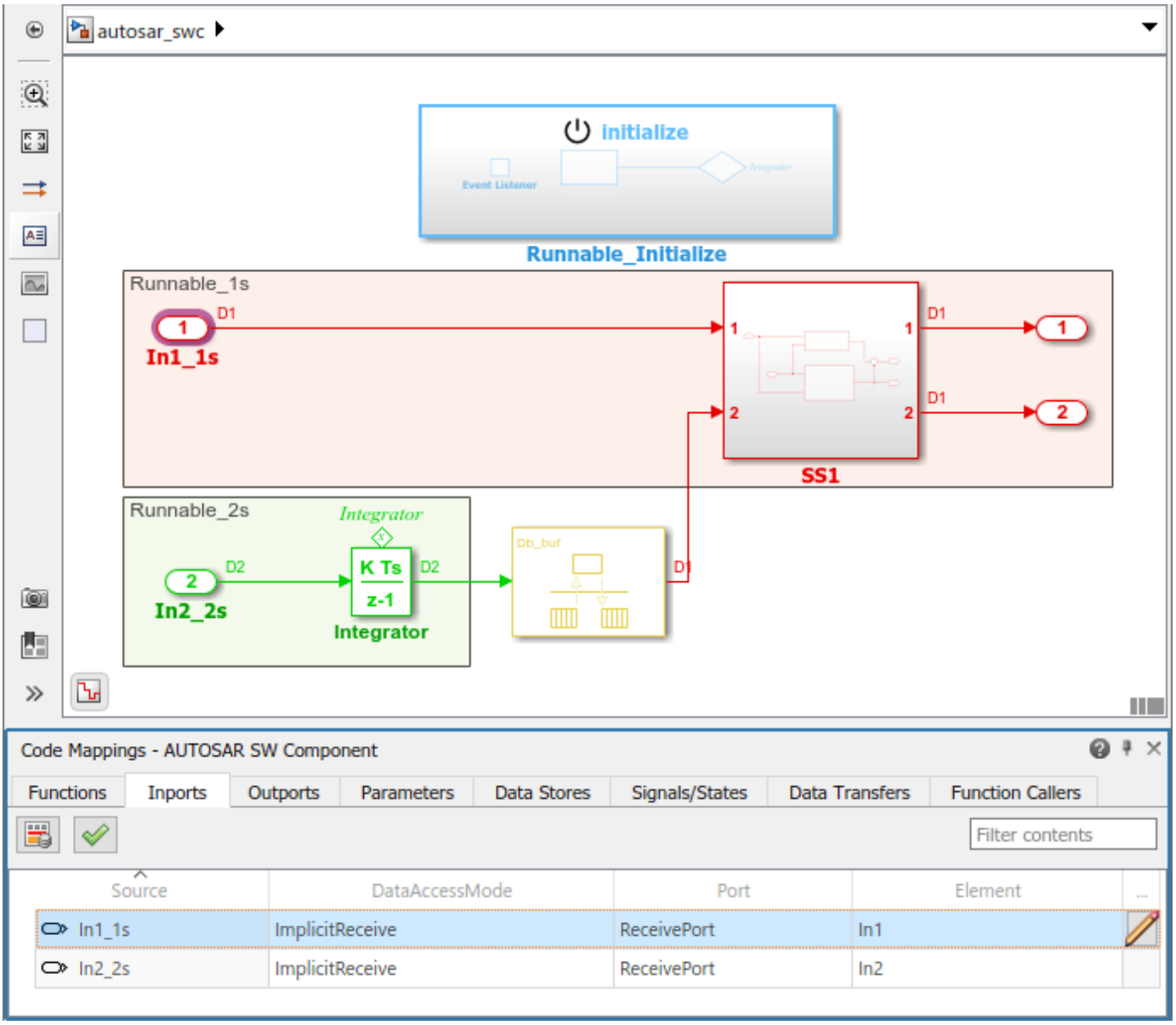
Work through the quick-start procedure. In the Output window, select output option **C code compliant with AUTOSAR**.



The quick-start software takes the following steps to configure an AUTOSAR software component model:

1. Configures code generation settings for the model. If the AUTOSAR target is not already selected, the software sets model configuration parameter **System target file** to `autosar.tlc` and **Generate XML for schema version** to a default schema value.
2. If no AUTOSAR mapping exists, creates a mapped AUTOSAR software component for the model.
3. Performs a model build.

In the last window, when you click **Finish**, your model opens in the AUTOSAR Code perspective. The Code perspective displays your model, and directly below the model, the Code Mappings editor.

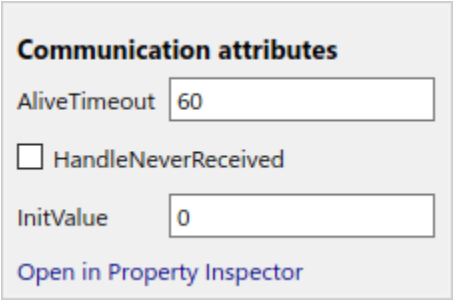


Develop Simulink Representation of AUTOSAR Software Component

After you create an AUTOSAR software component model in Simulink, use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR component. For more information, see “AUTOSAR Component Configuration” (AUTOSAR Blockset).

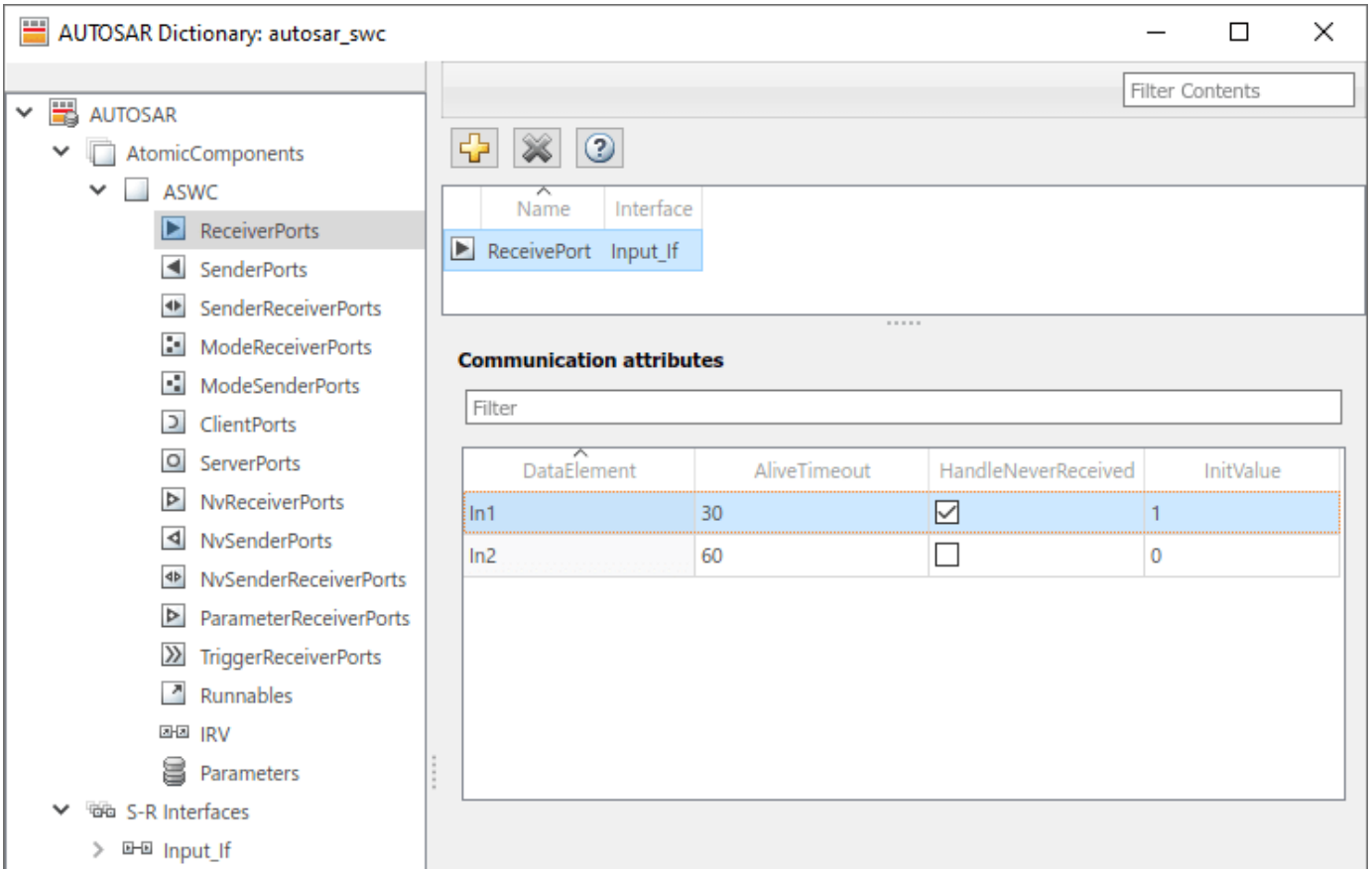
In a tabbed table format, the Code Mappings editor displays Simulink model elements, such as entry-point functions, inports, outports, and data transfers. Use the editor to map Simulink model elements to AUTOSAR component elements from a Simulink model perspective. AUTOSAR component elements are defined in the AUTOSAR standard, and include runnable entities, ports, and inter-runnable variables (IRVs).

In the Code Mappings editor, when you select a model element, it is highlighted in the model and an edit icon is displayed. To view additional AUTOSAR properties for the element, click the icon. Here is the AUTOSAR properties display for a model inport.



In each Code Mappings editor tab, you can select model elements and modify their AUTOSAR mapping and attributes. Your modifications are reflected in the generated ARXML descriptions and C code. If you are using AUTOSAR example model `autosar_swc` with this example, modify the communication attributes for the mapped Simulink inport `In1_1s`. In the AUTOSAR properties display, change the `AliveTimeout` attribute from 60 to 30, change `HandleNeverReceived` from cleared to selected, and change `InitValue` from 0 to 1.

To configure the AUTOSAR properties of the mapped AUTOSAR software component, open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button, which is the leftmost button. The AUTOSAR Dictionary opens in the AUTOSAR view that corresponds to the Simulink element that you last selected and mapped in the Code Mappings editor. If you selected and mapped a Simulink inport, the dictionary opens in ReceiverPorts view and displays the AUTOSAR port to which you mapped the inport.



In a tree format, the AUTOSAR Dictionary displays the mapped AUTOSAR component and its elements, communication interfaces, computation methods, software address methods, and XML options. Use the dictionary to configure AUTOSAR elements and properties from an AUTOSAR component perspective.

In the ReceiverPorts view, select the AUTOSAR receiver port to which the Simulink inport was mapped in the Code Mappings editor. If an AUTOSAR element has additional undisplayed attributes, selecting the element displays them. In each AUTOSAR element view, you can add or rename AUTOSAR elements and modify their displayed properties. Your modifications are reflected in the generated ARXML descriptions and C code.

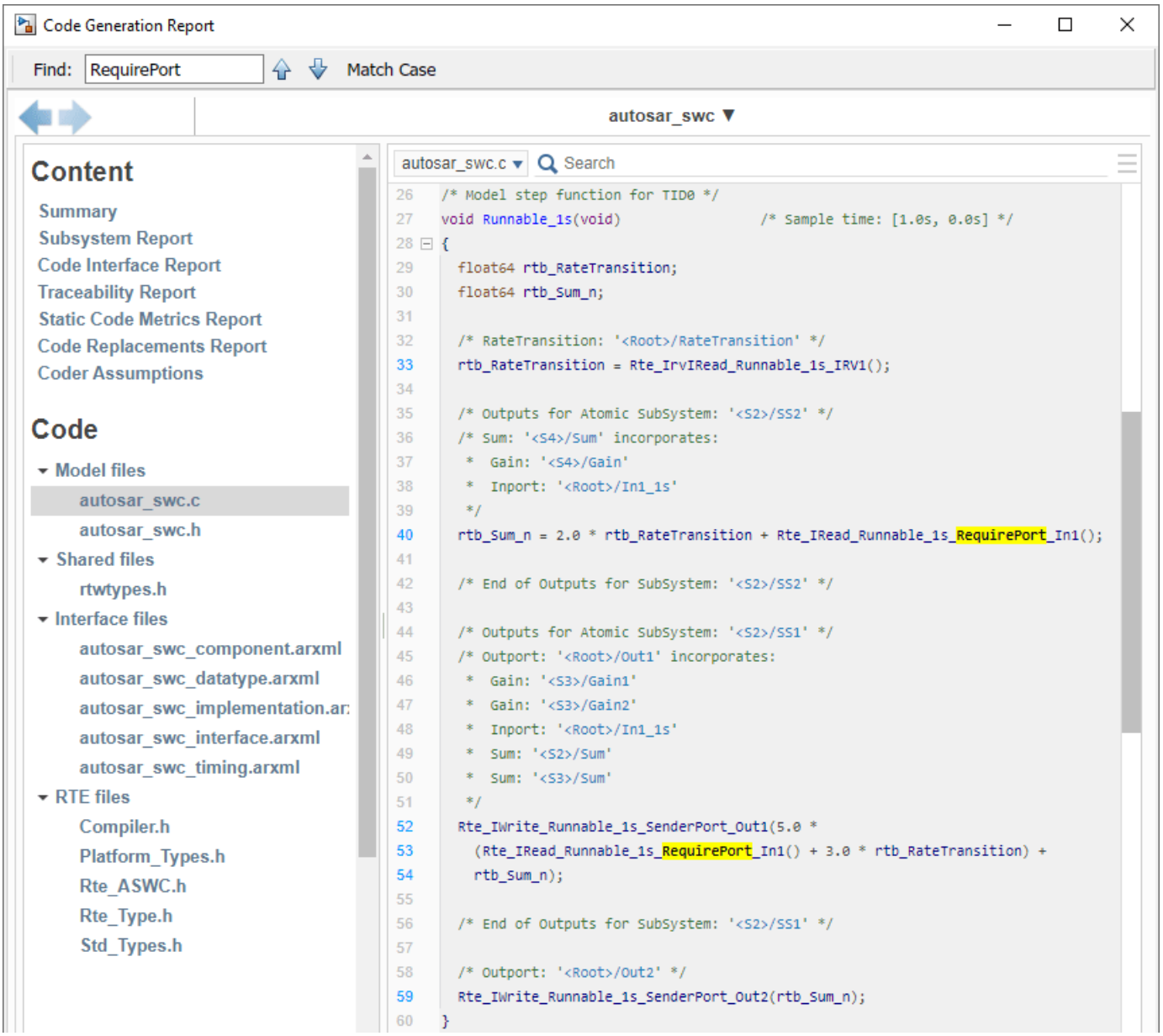
If you are using AUTOSAR example model `autosar_swc` with this example, rename the AUTOSAR receiver port from `ReceivePort` to `RequirePort`. To initiate the edit, click in the **Name** value field.

Generate C Code and ARXML Descriptions

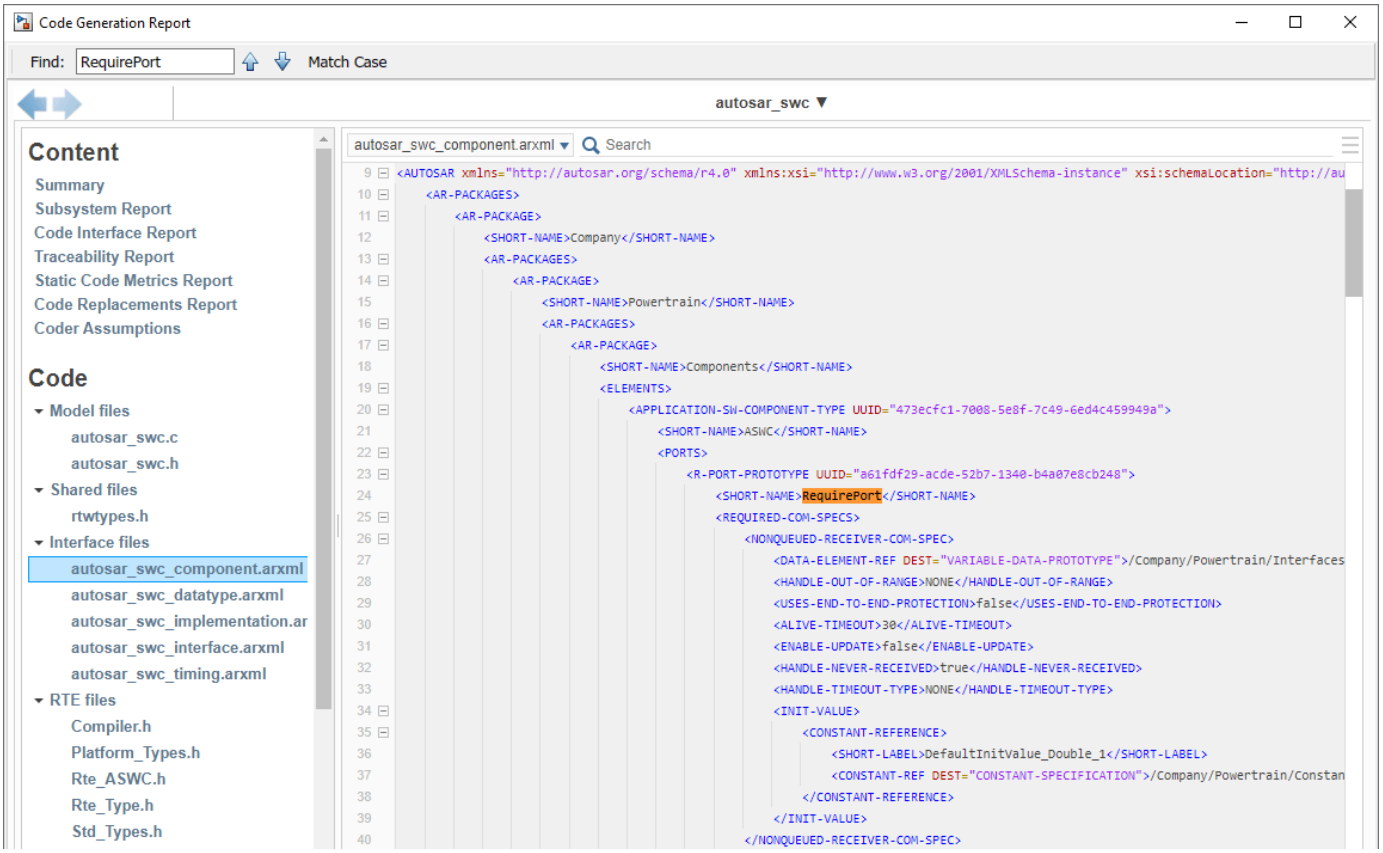
If you have Simulink Coder and Embedded Coder software, you can build the AUTOSAR model. Building the AUTOSAR model generates AUTOSAR-compliant C code and exports ARXML descriptions. In the model window, press **Ctrl+B**.

When the build completes, a code generation report opens. Examine the report. Verify that your Code Mappings editor and AUTOSAR Dictionary changes are reflected in the C code and ARXML descriptions. For example, use the **Find** field to search for the name of the AUTOSAR receiver port that you modified and renamed.

The generated C code encodes the AUTOSAR receiver port name in AUTOSAR run-time environment API read calls.



The generated ARXML description of the AUTOSAR receiver port uses the modified port name and modified values for port communication attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`.



Related Links

- “Code Generation” (AUTOSAR Blockset)
- “AUTOSAR Component Configuration” (AUTOSAR Blockset)
- “AUTOSAR Blockset”

Develop a Model That Complies with the IEC 61508 Standard

This example shows how to use the Model Advisor to check that a model complies with the IEC 61508 safety standard.

The IEC 61508 Model Advisor checks identify issues with a model that impede deployment in safety-related applications or limit traceability.

Understanding the Model

Model `rtwdemo_iec61508` check whether the 1-norm distance between points `(x1,x2)` and `(y1,y2)` is less than or equal to a given threshold `thr`. For two points `(x1,x2)` and `(y1,y2)`, the 1-norm distance is given as:

$$\sum_{i=1}^{2} |x_i - y_i|$$

Open and review the model.

```
model='rtwdemo_iec61508';
open_system(model)
```

Run the IEC 61508 Modeling Standard Checks

To deploy the model in a safety-related software component that must comply with the IEC 61508 safety standard, check the model for issues that might impede deployment in such an environment or limit traceability between the model and generated source code.

1. To open the Model Advisor, in the Simulink® editor, click the **Modeling** tab and select Model Advisor. A **System Selector - Model Advisor** dialog box opens. Select the model or system that you want to review and click **OK**. Or enter `modeladvisor('rtwdemo_IEC61508')` at the MATLAB® command line.
2. In the left pane of the Model Advisor, expand **By Task > Modeling Standards for IEC 61508**. If the **By Task** folder is not displayed in the Model Advisor window, open **Settings > Preferences** and select **Show By Task Folder**.
3. Select the checks that you want to include in your model analysis.
4. Click on the folder that contains the checks and, on the right pane of the Model Advisor, select **Show report after run** to automatically generate and display the report in HTML format.
5. Click **Run Selected Checks** to execute the analysis.

Review Check Results

The Model Advisor processes the IEC 61508 checks and displays the results.

1. After the analysis is complete, review the aggregate results in the right pane of the Model Advisor. You can see the total number of checks that passed, failed, were flagged as warnings, and did not execute.
2. To review the results for a specific check, in the **By Task > Modeling Standards for IEC 61508** folder, select a check. For more information about the check and how to resolve reported issues, click **Help**.

3. Address the reported issues and rerun the checks.
4. To review the generated HTML report of the results, select the **By Task > Modeling Standards for IEC 61508** folder and click the link in the **Report** box.
5. Print the generated HTML report. You can use the report as evidence in the IEC 61508 compliance process.
6. If desired, click the **Generate Code Using Embedded Coder** button in the model to inspect the generated code and the traceability report.

See Also

- For descriptions of the IEC 61508 checks, see “Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Industry Standards” (Simulink Check) in the Simulink Check™ documentation.
- For more information on using Model Advisor, see “Run Model Advisor Checks and Review Results” in the Simulink documentation.
- For more information on resolving issues, see “Address Model Check Results” in the Simulink documentation.

CERT C Compliance Summary Tables

In this section...

"CERT C Code Compliance Evaluation Summary" on page 23-27
"Explanatory Notes" on page 23-36

MathWorks evaluates C code generated by Embedded Coder from Simulink models and Stateflow charts against the CERT C coding standards[3]. The evaluation results are available in these summary tables.

The tables show the compliance status of generated code with CERT C rules:

- Compliant: The generated code is fully compliant with the CERT C coding standards if you follow the modeling recommendations associated with the rule.
- Noncompliant: The generated code fails to satisfy the CERT C coding standards.

Note For rules marked as Noncompliant*, you can generate CERT C compliant code in some modeling instances by following the modeling guidelines shown in the table, where applicable.

The table also provides guidance on achieving compliance, where applicable:

- Model Advisor checks, which facilitates the model assessment process to achieve code compliance.
- Modeling recommendations, which describe model settings, blocks usage, and block parameter settings that improve code compliance.
- Explanatory notes describe violations that are not within the scope of Embedded Coder, and other limitations.

When preparing the CERT C compliance statement for your project, you can use the tables in the CERT C Code Compliance Evaluation Summary. The tables align with the published CERT C Rules and Recommendations (Polyspace Bug Finder). The tables are based only on *Rules* as described in Section 1.7 of the *SEI CERT C Coding Standards, 2016 edition*. *Recommendations* are not considered.

Note To detect issues in a model that might produce CERT C coding standard violations, run the Model Advisor and select Modeling Guidelines for Secure Coding Checks (Simulink Check).

3 Polyspace is used to assess the generated code from Embedded Coder against the CERT C coding guidelines. Other CERT C code analysis tools can yield different results.

CERT C Code Compliance Evaluation Summary

Rule 01. Preprocessor (PRE)

Rule	Compliance
PRE30-C	Compliant • Model recommendations • hisl_0062
PRE31-C	Compliant
PRE32-C	Compliant

Rule 02. Declarations and Initialization (DCL)

Rule	Compliance
DCL30-C	Compliant

DCL31-C	Compliant
	• Explanatory Note
	  • “Usage of Model Configuration Parameter 'Leverage target hardware instruction set extensions'” on page 23-37
	  • “Usage of hardware board configuration” on page 23-36

DCL36-C	Compliant

DCL37-C	Noncompliant*
	• Model recommendations
	  • hisl_0032
	• Explanatory Note
	  • “Usage of Model Configuration Parameter 'Leverage target hardware instruction set extensions'” on page 23-37
	  • “Identifier specification” on page 23-37

DCL38-C	Compliant
	• Model recommendations
	  • hisl_0060
	• Explanatory Note
	  • “Flexible array violation” on page 23-36

DCL39-C	Noncompliant
DCL40-C	Compliant
DCL41-C	Compliant

Rule 03. Expressions (EXP)

Rule	Compliance

EXP30-C	Compliant
	• Model recommendations
	  • hisl_0062
	• Explanatory Note
	  • “Usage of const or volatile qualifiers” on page 23-36
	  • “Blocks usage” on page 23-36
	  • “Usage of hardware board configuration” on page 23-36

EXP32-C	Compliant
	• Model recommendations
	  • hisl_0020
	• Explanatory Note
	  • “Usage of const or volatile qualifiers” on page 23-36
	  • “Usage of hardware board configuration” on page 23-36

EXP33-C	Compliant
	• Model recommendations
	  • hisl_0010
	  • hisl_0015
	  • hisl_0029
	• Explanatory Note
	  • “Configuration of Variant choices” on page 23-36

EXP34-C	Compliant
	• Explanatory Note
	  • “Run-time error analysis” on page 23-37
	  • “Usage of Model Configuration Parameter 'Classic call interface'” on page 23-37

EXP35-C	Compliant
EXP36-C	Noncompliant* • Model recommendations • hisl_0020

Rule	Compliance

EXP37-C	Compliant
	• Model recommendations
	  • hisl_0020
	• Explanatory Note
	  • “Usage of User-Written code” on page 23-38

EXP39-C	Noncompliant* • Model recommendations • hisl_0020
EXP40-C	Compliant
EXP42-C	Compliant
EXP43-C	Noncompliant* • Model recommendations • hisl_0029
EXP44-C	Compliant
EXP45-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
EXP46-C	Noncompliant
EXP47-C	Compliant

Rule 04. Integers (INT)

Rule	Compliance
INT30-C	Noncompliant* • Model Advisor Checks • “Detect Integer Overflow” (Simulink Check)

INT31-C	Noncompliant*
	• Model recommendations
	  • hisl_0001
	  • hisl_0054
	• Model Advisor Checks
	  • “Detect Division by Zero” (Simulink Check)
	  • “Detect Integer Overflow” (Simulink Check)
	• Explanatory Note
	  • “Run-time error analysis” on page 23-37

INT32-C	Compliant
	• Model recommendations
	  • hisl_0001
	  • hisl_0006
	• Explanatory Note
	  • “Run-time error analysis” on page 23-37

INT33-C	Compliant
	• Model recommendations
	  • hisl_0002
	  • hisl_0004
	  • hisl_0028
	  • hisl_0067
	• Explanatory Note
	  • “Run-time error analysis” on page 23-37

INT34-C	Compliant
	• Model recommendations
	  • hisl_0020
	  • hisl_0073

INT35-C	Compliant
INT36-C	Noncompliant

Rule 05. Floating Point (FLP)

Rule	Compliance

FLP30-C	Noncompliant*
	• Model recommendations
	  • hisl_0022
	  • hisl_0102

FLP32-C	Compliant
	• Model recommendations
	  • hisl_0003
	  • hisl_0004

FLP34-C	Compliant • Explanatory Note • “Blocks usage” on page 23-36
FLP36-C	Compliant
FLP37-C	Compliant

Rule 06. Arrays (ARR)

Rule	Compliance

ARR30-C	Noncompliant*
	• Model recommendations
	  • hisl_0020
	• Model Advisor Checks
	  • “Detect Out Of Bound Array Access” (Simulink Check)

ARR32-C	Compliant
ARR36-C	Compliant
ARR37-C	Compliant
ARR38-C	Compliant
ARR39-C	Compliant

Rule 07. Characters and Strings (STR)

Rule	Compliance
STR30-C	Compliant
STR31-C	Compliant • Explanatory Note • “Run-time error analysis” on page 23-37
STR32-C	Compliant
STR34-C	Noncompliant* • Model recommendations • hisl_0022
STR37-C	Compliant
STR38-C	Compliant

Rule 08. Memory Management (MEM)

Rule	Compliance
MEM30-C	Compliant • Model recommendations • hisl_0060
MEM31-C	Compliant
MEM33-C	Compliant
MEM34-C	Compliant

MEM35-C	Compliant
	• Model recommendations
	  • hisl_0060
	• Explanatory Note
	  • “Run-time error analysis” on page 23-37

MEM36-C	Compliant

Rule 09. Input Output (FIO)

Rule	Compliance
FIO30-C	Compliant
FIO31-C	Compliant
FIO32-C	Noncompliant* • Model recommendations • hisl_0020
FIO34-C	Compliant
FIO37-C	Compliant
FIO38-C	Compliant
FIO39-C	Compliant
FIO40-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
FIO41-C	Compliant
FIO42-C	Compliant • Model recommendations • hisl_0020
FIO44-C	Compliant
FIO45-C	Compliant
FIO46-C	Compliant
FIO47-C	Compliant • Model recommendations • hisl_0020

Rule 10. Environment (ENV)

Rule	Compliance
ENV30-C	Compliant
ENV31-C	Compliant
ENV32-C	Compliant
ENV33-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
ENV34-C	Compliant

Rule 11. Signals (SIG)

Rule	Compliance
SIG30-C	Compliant
SIG31-C	Compliant
SIG34-C	Compliant
SIG35-C	Compliant

Rule 12. Error Handling (ERR)

Rule	Compliance
ERR30-C	Compliant
ERR32-C	Compliant
ERR33-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
ERR34-C	Compliant

Rule 14. Concurrency (CON)

Rule	Compliance
CON30-C	Compliant
CON31-C	Compliant
CON32-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
CON33-C	Compliant
CON34-C	Compliant
CON35-C	Compliant
CON36-C	Compliant
CON37-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
CON38-C	Compliant
CON39-C	Compliant
CON40-C	Compliant
CON41-C	Compliant
CON43-C	Compliant

Rule 48. Miscellaneous (MSC)

Rule	Compliance
MSC30-C	Noncompliant
MSC32-C	Noncompliant
MSC33-C	Compliant
MSC37-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
MSC38-C	Compliant
MSC39-C	Compliant
MSC40-C	Compliant
MSC41-C	Compliant

Rule 50. POSIX (POS)

Rule	Compliance
POS30-C	Compliant • Explanatory Note • “Usage of hardware board configuration” on page 23-36
POS34-C	Compliant
POS35-C	Compliant
POS36-C	Compliant
POS37-C	Compliant
POS38-C	Compliant
POS39-C	Compliant
POS44-C	Compliant
POS47-C	Noncompliant
POS48-C	Compliant
POS49-C	Compliant
POS50-C	Compliant
POS51-C	Compliant
POS52-C	Compliant
POS53-C	Compliant
POS54-C	Noncompliant

Rule 51. Microsoft Windows (WIN)

Rule	Compliance
WIN30-C	Compliant • Model recommendations • hisl_0060

Explanatory Notes

Usage of const or volatile qualifiers

The code generator produces noncompliant code for Simulink data objects with its storage class configured with `const` or `volatile` qualifiers.

Usage of hardware board configuration

Simulink models configured for support packages like Simulink Support Package for Raspberry Pi® Hardware, C2000™ Microcontroller Blockset, or such configurations may generate noncompliant code.

Note Third-party files such as drivers used with code that is generated using Embedded Coder are not considered in the evaluation.

Blocks usage

Rate Transition block and Matrix Interpolation block generate noncompliant code.

Flexible array violation

Polyspace may report violations if the generated code has a structure with one-dimensional array. For example, this code uses a structure `DW_mg_foreach_T` and a member array `CoreSubsys`.

```
typedef struct {
  real_T Delay_DSTATE[2];              /* '<Root>/Delay' */
  ...
  DW_CoreSubsys_mg_foreach_T CoreSubsys[1];/* '<Root>/S1' */
} DW_mg_foreach_T;

DW_mg_foreach_T mg_foreach_DW;
mg_foreach_DW.CoreSubsys[0].Delay_DSTATE[0] = 0.0;
```

`CoreSubsys` is a fixed size array and not a flexible array. In this case, the flexible array violation coming from the Polyspace is not an issue.

Configuration of Variant choices

Variant Source block not configured with the default variant choice generates an empty `else` statement. To avoid this, specify a default variant control variable for the variant blocks. For more information, see “Types of Variant Controls in expression Mode”.

Usage of Model Configuration Parameter 'Classic call interface'

When the model configuration parameter **Classic call interface** is enabled, Polyspace may report a null pointer dereferencing violation in the generated code. For example, this code uses a static variable `exmpModelK_m` and is dereferenced by a variable `mdlTsMap`.

```
static RT_MODEL__EXAMPMODEL exmpModelK_m;
RT_MODEL__EXMPMODEL *const exmpModelK_m = &exmpModelK_m_;

.....

  /* Initialize timing info */
  {
    int_T *mdlTsMap = exmpModelK_m->Timing.sampleTimeTaskIDArray;
    mdlTsMap[0] = 0;

    exmpModelK_m->Timing.sampleTimeTaskIDPtr = (&mdlTsMap[0]);

...

}
```

In this case of violation, it is the static memory which is dereferenced. Static memory address is of non-NULL in nature and the violation coming from the Polyspace is not an issue.

Usage of Model Configuration Parameter 'Leverage target hardware instruction set extensions'

When the model configuration parameter **Leverage target hardware instruction set extensions** is set to anything other than `None`, the generated SIMD instruction starts with the identifier `_mm`. Generation of these intrinsic functions is not supported by Embedded Coder. For more information, see “Generate SIMD Code from Simulink Blocks”.

Identifier specification

You can configure Embedded Coder to limit the number of characters imposed by the implementation. For more information, see **Maximum identifier length**.

To ensure unique names for different types of variables (such as local variables, global variables, and macros), implement a naming convention. For more information, see **Model Configuration Parameters: Code Generation Symbols**.

In case of long identifiers, use shared utilities. For more information, see “Generate Shared Utility Code”.

If you specify identifier naming rules, you are responsible for ensuring compliance.

Run-time error analysis

Use Simulink Design Verifier™ to detect design errors at the model level. For more information, see “Analyze Models for Design Errors” (Simulink Design Verifier).

To handle overflow violations, see “Handle Overflows in Simulink Models” (Fixed-Point Designer).

Use Polyspace Bug Finder to identify run-time errors and Polyspace Code Prover™ to prove the absence of run-time errors. For more information, see “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Code Prover):

- “Configure and Run Analysis” (Polyspace Code Prover)
- “Configure and Run Analysis” (Polyspace Bug Finder)

Usage of User-Written code

User-written C code and third-party libraries that are used with code generated by Embedded Coder are not considered in the evaluation.

MISRA C:2012 Compliance and Deviations for Code Generated by Using Embedded Coder

- “Developing a MISRA C:2012 Compliance Statement” on page 24-2
- “Evaluate Your Generated Code for MISRA C:2012 Compliance” on page 24-4
- “MISRA C:2012 Compliance Information Summary Tables” on page 24-6
- “Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012” on page 24-20
- “Deviations Rationale for MISRA C:2012 Compliance” on page 24-27

Developing a MISRA C:2012 Compliance Statement

As part of the model development process, it is important that C code generated by Embedded Coder from Simulink and Stateflow complies with industry coding standards.

When using MISRA C:2012 coding guidelines to evaluate the quality of your generated C code, you are required per section 5.3 of the *MISRA C:2012 Guidelines for the Use of C Language in Critical Systems* document to prepare a compliance statement for the project being evaluated. To assist you in the development of this compliance statement, MathWorks evaluates the MISRA C:2012 guidelines against C code generated by using Embedded Coder and provides the following information:

- Compliance Summary Tables on page 24-6, which identify the method used to obtain compliance for each rule and directive.
- Deviations on page 24-27, which identify rules or directives that are not compliant.

For information about the process MathWorks uses to evaluate generated C code against MISRA C:2012 guidelines, see “MathWorks Process for Identifying Violations of MISRA C:2012 Guidelines in Generated C Code” on page 24-2.

For additional information about the MISRA organization, their coding guidelines, and the MISRA publication timeline, see www.misra.org.uk .

Disclaimer While adhering to the recommendations in the MISRA C:2012 Compliance and Deviations for Code Generated using Embedded Coder documentation will reduce the risk that an error is introduced during development and not be detected, it is not a guarantee that the system being developed will be safe. Conversely, if some of the recommendations are not followed, it does not mean that the system being developed will be unsafe.

MathWorks Process for Identifying Violations of MISRA C:2012 Guidelines in Generated C Code

To determine any potential violations in the generated code, MathWorks maintains an extensive set of test models that cover the standard usage of compliant blocks. For each release, MathWorks uses these test models with the following products to evaluate the modeling, code generation, and analysis of generated code.

Product	Purpose
Simulink	Create/maintain models.
Stateflow	Create/maintain models.
Fixed-Point Designer™	Create/maintain models.
Embedded Coder	Generate C code.
Simulink Check	Execute MISRA C:2012 Model Advisor checks.[a]

Product	Purpose
Polyspace Bug Finder	Identify bug/coding defects. Use the checkers for “MISRA C:2012 Directives and Rules” (Polyspace Bug Finder) to analyze the generated code and provide information about violations.

a MISRA C:2012 checks are available only when you have a Simulink Check or Embedded Coder license.

Note The compliance analysis performed by the Polyspace MISRA C:2012 Checker assesses C code generated by Embedded Coder. No assessment is made of the Embedded Coder tool chain. Handwritten C code and third-party libraries that are used with code generated by Embedded Coder are not considered. Other MISRA C code analysis tools can yield different results.

Evaluate Your Generated Code for MISRA C:2012 Compliance

It is important to check that C code generated by Embedded Coder from Simulink and Stateflow complies with MISRA C:2012 coding standards. This workflow illustrates the process of evaluating your generated code for compliance to MISRA C:2012 guidelines.

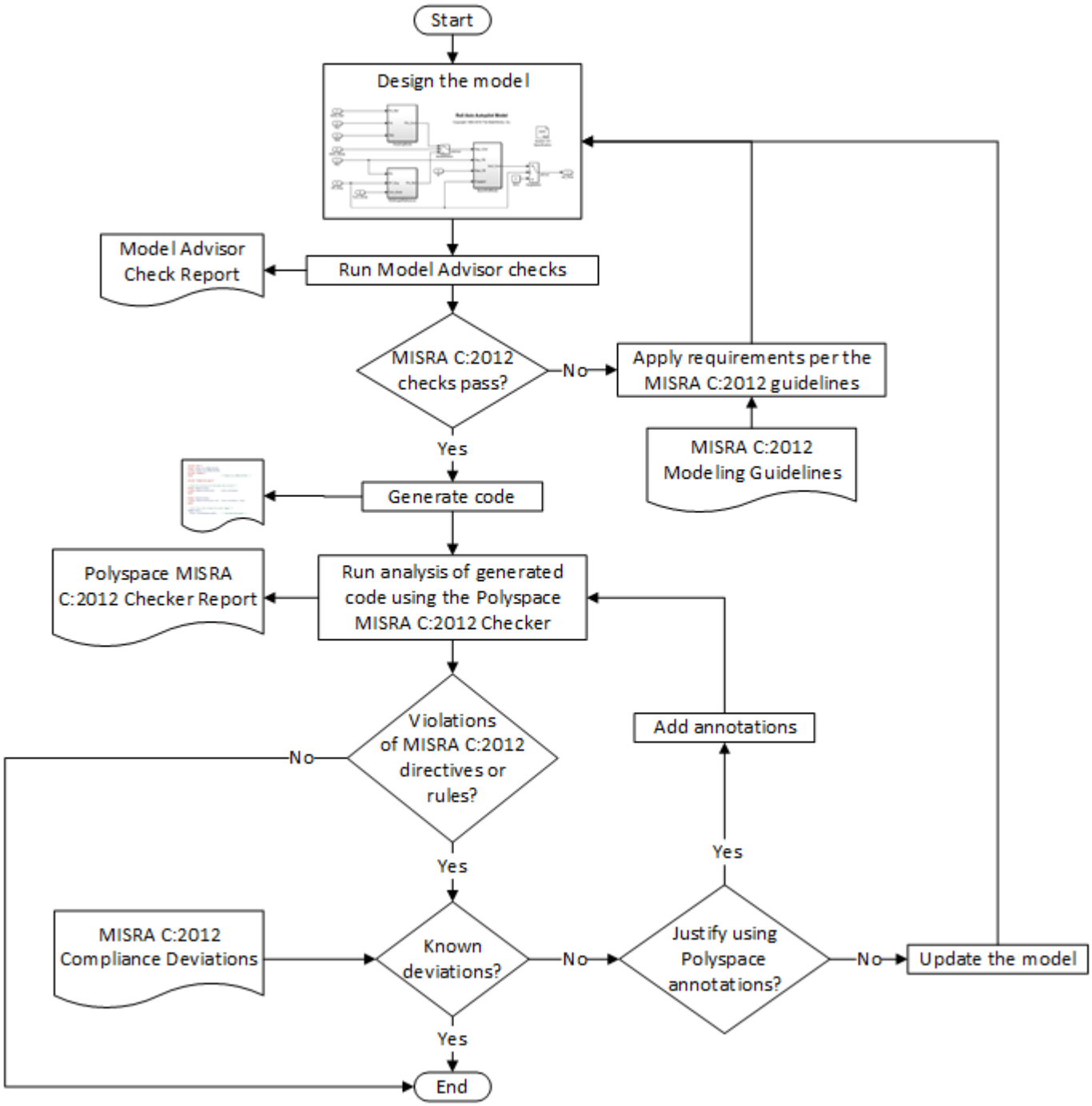
1. Design your model in Simulink or Stateflow.
2. Open the Model Advisor and run the MISRA C:2012 checks (Simulink Check), which are available in **By Task > Modeling Standards for MISRA C:2012**.
3. If necessary, modify the model to adhere to the "Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012" on page 24-20.
4. After the MISRA C:2012 checks pass, generate code by using Embedded Coder.

   To avoid possible name clashes, multiple functions definitions, and multiple data definitions, use one of these code generation approaches:

   - Single model: Generate code from a single model, including submodels, by using Model blocks.
   - Multiple models: Generate code from multiple models, including:
     - A shared utility folder to synchronize functions
     - Data ownership to control data definitions
     - Hand integration to manage code generated from various models
5. Run the checkers for "MISRA C:2012 Directives and Rules" (Polyspace Bug Finder).

   For more information about running a Polyspace analysis on your generated code, including analysis options and results, see:

   - "Bug Finder Analysis in Simulink" (Polyspace Bug Finder)
   - "Code Prover Analysis in Simulink" (Polyspace Code Prover)
6. Justify violations to the MISRA C:2012 coding standards using Polyspace annotations (Polyspace Bug Finder).


Start
Design the model
Model Advisor Check Report
Run Model Advisor checks
MISRA C:2012 checks pass?
No
Apply requirements per the MISRA C:2012 guidelines
MISRA C:2012 Modeling Guidelines
Yes
Generate code
Polyspace MISRA C:2012 Checker Report
Run analysis of generated code using the Polyspace MISRA C:2012 Checker
Add annotations
Violations of MISRA C:2012 directives or rules?
No
Yes
Yes
MISRA C:2012 Compliance Deviations
Known deviations?
No
Justify using Polyspace annotations?
No
Update the model
Yes
End

MISRA C:2012 Compliance Information Summary Tables

In this section...

"MISRA C:2012 Main Document" on page 24-6
"MISRA C:2012 Amendment 1" on page 24-15
"MISRA C:2012 Amendment 2" on page 24-16
"Explanatory Notes" on page 24-16

MathWorks evaluates C code generated by Embedded Coder from Simulink models and Stateflow charts against the MISRA C:2012 coding standards. The results from this effort are available in these compliance summary tables. These tables identify:

- Methods used to obtain compliance:
  - Compliant: Compliance to the rule/directive is achieved through adherence to the code generation process, modeling guidelines, or Model Advisor checks. When applicable, there are explanatory notes that provide information relevant to compliance methods or actions that you can perform to satisfy the directive or rule.
  - Deviation: The rule or directive is not compliant.
- Whether the Polyspace MISRA C:2012 Checker supports the rule or directive.

You can use these tables when preparing the MISRA C:2012 compliance statement for your project as required per section 5.3 of the *MISRA C:2012 Guidelines for the Use of C Language in Critical Systems* document. These tables align with the published MISRA C:2012 rule and directives tables. The categories in the table are based on Appendix E of the MISRA guidelines identified for automatically generated code. They can be used for Mandatory and Required categories only (not for Advisory and Readability categories).

MISRA C:2012 Main Document

"Implementation" MISRA C:2012 Directives

Directive	Category	Compliance	Polyspace Support?
1.1	Required	Compliant: • hisl_0060 • hisl_0016: Usage of blocks that compute relational operators • "Explanatory Note for Directive 1.1" on page 24-17	Yes, partially supported

"Compilation and Build" MISRA C:2012 Directives

Directive	Category	Compliance	Polyspace Support?
2.1	Required	Compliant	Yes

"Requirements Traceability" MISRA C:2012 Directives

Directive	Category	Compliance	Polyspace Support?
3.1	Required	Compliant: • “Explanatory Note for MISRA Directive 3.1” on page 24-17	No

"Code Design" MISRA C:2012 Directives

Directive	Category	Compliance	Polyspace Support?
4.1	Required	Compliant: • hisl_0001 • hisl_0002 • hisl_0003 • hisl_0004 • hisl_0005 • hisl_0006 • hisl_0007 • hisl_0008 • hisl_0010 • hisl_0011 • hisl_0028 • hisl_0053 • hisl_0054 • hisl_0067 • “Explanatory Note for Directive 4.1” on page 24-17	Yes
4.3	Required	Compliant: • “Explanatory Note for Directive 4.3” on page 24-18	No
4.6	Advisory	Not Applicable. • See “Explanatory Note for Directive 4.6” on page 24-18	N/A
4.7	Required	Compliant: • Check for missing error ports for AUTOSAR receiver interfaces • Check for blocks not recommended for MISRA C:2012 Deviation: • Deviation for AUTOSAR target-based code generator - sender ports. See “MISRA C:2012 Directive 4.7 Deviation Rationale” on page 24-27	Yes[a]
4.10	Required	Compliant	Yes
4.11	Required	Compliant: • “Explanatory Note for Directive 4.11” on page 24-18	Yes

Directive	Category	Compliance	Polyspace Support?
4.12	Required	Compliant: • hisl_0060	No

a The Polyspace MISRA C:2012 Checker might flag Directive 4.7 as a Rule 17.7 violation (Polyspace Bug Finder) for user-defined functions when there is no knowledge about whether the return value contains error information.

"Standard C Environment" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
1.1	Required	Compliant	Yes
1.3	Required	Compliant	Yes

"Unused Code" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
2.1	Required	Compliant: • hisl_0053 • hisl_0101 • hisl_0060 • “Explanatory Note for Rule 2.1” on page 24-18	Yes
2.2	Required	Compliant	Yes

"Comments" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
3.1	Required	Compliant: • Check for unsupported block names	Yes
3.2	Required	Compliant	Yes

"Character Sets and Lexical Conventions" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
4.1	Required	Compliant	Yes

"Identifiers" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
5.1	Required	Compliant: • hisl_0060 • hisl_0063 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.2	Required	Compliant: • hisl_0060 • hisl_0063 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.4	Required	Compliant: • hisl_0060 • hisl_0063 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.5	Required	Compliant: • hisl_0060 • hisl_0063 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.6	Required	Compliant: • hisl_0060 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.7	Required	Compliant: • hisl_0060 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes
5.8	Required	Compliant: • hisl_0060 • “Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8” on page 24-18	Yes

"Types" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
6.1	Required	Compliant: • hisl_0060	Yes
6.2	Required	Compliant: • hisl_0060	Yes

"Literals and Constants" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
7.1	Advisory	Compliant	Yes
7.2	Readability	Compliant	Yes
7.4	Required	Compliant	Yes

a Guideline suggests Rule 7.1 and 7.2 as Required for manually generated code.

"Declarations and Definitions" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
8.1	Required	Compliant	Yes
8.2	Required	Compliant	Yes
8.3	Required	Compliant	Yes
8.6	Required	Compliant	Yes
8.8	Required	Compliant	Yes
8.10	Required	Compliant	Yes
8.12	Required	Compliant: • “Explanatory Note for Rule 8.12” on page 24-18	Yes
8.14	Advisory	Compliant	Yes

a Guideline suggests Rule 8.14 as Required for manually generated code.

"Initialization" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
9.1	Mandatory	Compliant: • hisl_0029 • Check usage of Assignment blocks	Yes
9.4	Required	Compliant	Yes
9.5	Readability	Compliant	Yes

a Guideline suggests Rule 9.5 as Required for manually generated code.

"The Essential Type Model" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
10.2	Advisory	Compliant	Yes
10.3	Advisory	Compliant	Yes

a Guideline suggests Rule 10.2 and 10.3 as Required for manually generated code.

"Pointer Type Conversion" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
11.1	Required	Compliant	Yes
11.2	Required	Compliant	Yes
11.3	Required	Compliant: • hisl_0020	Yes
11.6	Required	Compliant	Yes
11.7	Required	Compliant	Yes
11.8	Required	Compliant: • hisl_0020 • “Explanatory Note for Rule 11.8” on page 24-19	Yes

"Expressions" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
12.2	Required	Compliant	Yes

"Side Effects" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
13.1	Required	Compliant	Yes
13.2	Required	Compliant: • hisl_0062	Yes
13.5	Required	Compliant: • hisl_0062 Deviation: • “MISRA C:2012 Rule 13.5 Deviation Rationale” on page 24-30	Yes
13.6	Mandatory	Compliant	Yes

"Control Statement Expressions" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
14.2	Readability	Compliant: • hisl_0006 • hisl_0007 • hisl_0008 • hisl_0010 • hisl_0011 • hisl_0101	Yes
14.3	Required	Compliant: • hisl_0101 • hisl_0060	Yes

a Guideline suggests Rule 14.2 as Required for manually generated code.

"Control Flow" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
15.2	Advisory	Compliant	Yes
15.3	Advisory	Compliant	Yes
15.6	Required	Compliant	Yes
15.7	Readability	Compliant	Yes

a Guideline suggests Rule 15.2, 15.3 and 15.7 as Required for manually generated code.

"Switch statements" MISRA C:2012 Rules

Rule	Category[a]	Compliance	Polyspace Support?
16.2	Advisory	Compliant	Yes
16.3	Advisory	Compliant	Yes
16.4	Advisory	Compliant: • hisl_0006 • hisl_0007 • hisl_0008 • hisl_0010 • hisl_0011	Yes
16.5	Advisory	Compliant	Yes
16.6	Advisory	Compliant	Yes
16.7	Advisory	Compliant	Yes

a Guideline suggests Rule 16.2, 16.3, 16.4, 16.5, 16.6 and 16.7 as Required for manually generated code.

"Functions" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
17.1	Required	Compliant	Yes
17.2	Required	Compliant: • hisl_0060 • hisf_0004	Yes
17.3	Mandatory	Compliant	Yes
17.4	Mandatory	Compliant	Yes
17.6	Mandatory	Compliant	Yes

"Pointers and Arrays" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
18.1	Required	Compliant	Yes
18.2	Required	Compliant	Yes
18.3	Required	Compliant	Yes
18.6	Required	Compliant	Yes
18.7	Required	Compliant	Yes
18.8	Required	Compliant	Yes

"Overlapping Storage" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
19.1	Mandatory	Compliant	Yes

"Preprocessing Directives" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
20.2	Required	Compliant	Yes
20.3	Required	Compliant	Yes
20.4	Required	Compliant	Yes
20.6	Required	Compliant	Yes
20.7	Required	Compliant	Yes
20.9	Required	Compliant	Yes
20.11	Required	Compliant	Yes
20.12	Required	Compliant	Yes
20.13	Required	Compliant	Yes
20.14	Required	Compliant	Yes

"Standard Libraries" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
21.1	Required	Compliant	Yes
21.2	Required	Compliant: • hisl_0032	Yes
21.3	Required	Compliant: • hisl_0060	Yes
21.4	Required	Compliant	Yes
21.5	Required	Compliant	Yes
21.6	Required	Compliant: • hisl_0060 • Check for blocks not recommended for MISRA C:2012	Yes
21.7	Required	Compliant	Yes
21.8	Required	Compliant	Yes
21.9	Required	Compliant	Yes
21.10	Required	Compliant	Yes
21.11	Required	Compliant	Yes
21.12	Required	Compliant	Yes

"Resources" MISRA C:2012 Rules

Rule	Category	Compliance	Polyspace Support?
22.1	Required	Compliant	Yes
22.2	Mandatory	Compliant	Yes
22.3	Required	Compliant	Yes
22.4	Mandatory	Compliant	Yes
22.5	Mandatory	Compliant	Yes
22.6	Mandatory	Compliant	Yes

MISRA C:2012 Amendment 1

"Code Design" MISRA C:2012 Amendment 1 Directives

Directive	Category	Compliance	Polyspace Support?
4.14	Required	Compliant	Yes

"Expressions" MISRA C:2012 Amendment 1 Rules

Directive	Category	Compliance	Polyspace Support?
12.5	Mandatory	Compliant	Yes

"Standard Libraries" MISRA C:2012 Amendment 1 Rules

Directive	Category	Compliance	Polyspace Support?
21.13	Mandatory	Compliant	Yes
21.14	Required	Compliant	Yes
21.15	Required	Compliant	Yes
21.16	Required	Compliant	Yes
21.17	Mandatory	Compliant	Yes
21.18	Mandatory	Compliant	Yes
21.19	Mandatory	Compliant	Yes
21.20	Mandatory	Compliant	Yes

"Resources" MISRA C:2012 Amendment 1 Rules

Directive	Category	Compliance	Polyspace Support?
22.7	Required	Compliant	Yes
22.8	Required	Compliant	Yes
22.9	Required	Compliant	Yes
22.10	Required	Compliant	Yes

MISRA C:2012 Amendment 2

"Standard C Environment" MISRA C:2012 Amendment 2 Rules

Directive	Category	Compliance	Polyspace Support?
1.4	Required	Compliant	Yes

"Standard Libraries" MISRA C:2012 Amendment 2 Rules

Directive	Category	Compliance	Polyspace Support?
21.21	Required	Compliant	Yes

Explanatory Notes

These explanatory notes are referenced from the “MISRA C:2012 Compliance Information Summary Tables” on page 24-6.

Explanatory Note for Directive 1.1

Information about the implementation-defined behavior for Embedded Coder is available in "Configure Run-Time Environment Options" on page 10-2 . Compiler documentation is out of scope.

Character set encoding is managed by using the `SavedCharacterEncoding` model parameter. For additional information, see `slCharacterEncoding`.

Configure the integer division method in the Model Configuration Parameters dialog box, on the **Hardware Implementation** pane. For additional information, see "Configure Run-Time Environment Options" on page 10-2

Embedded Coder generates `#pragma` when the user:

- Selects the use of memory sections
- Creates a custom storage class with memory sections on page 40-14

In both instances, you are responsible for documenting the intended use of the `#pragma`. For more information, see "Control Data and Function Placement in Memory by Inserting Pragmas" on page 45-2.

To enable the generation of bitfields:

1. Select at least one of these model configuration parameters:
   - **Pack Boolean data into bitfields**. This parameter is available only for ERT-based system target files on page 50-2.
   - **Use bitsets for storing state configuration**
   - **Use bitsets for storing Boolean data.**
2. Create a custom storage class with defined bitfields. See "Create Storage Classes by Using the Custom Storage Class Designer" on page 40-14 for more information.

If your model uses conversion of one datatype to another, Polyspace Checker shows violation of Dir 1.1 from the generated code of the model. In this case, the violation coming from the model is not an issue if you document this explicit conversion according to the MISRA guidelines.

Explanatory Note for MISRA Directive 3.1

You can link requirements model elements. These links are included in the generated C code to provide traceability from a requirements document, to the model elements, and to the generated code. For additional information, see "View Requirements Toolbox Links Associated with Model Elements" and "Link Blocks and Requirements" (Requirements Toolbox).

Explanatory Note for Directive 4.1

You can use Polyspace Bug Finder to identify run-time errors and Polyspace Code Prover to prove the absence of run-time errors. For information, see:

- "Configure and Run Analysis" (Polyspace Code Prover)
- "Configure and Run Analysis" (Polyspace Bug Finder)

Simulink Design Verifier can be used to detect design errors at the model level. For more information, see "Analyze Models for Design Errors" (Simulink Design Verifier).

To handle the overflow violations, see “Handle Overflows in Simulink Models” (Fixed-Point Designer).

Explanatory Note for Directive 4.3

Embedded Coder does not directly call assembly language code. You can add calls to assembly language functions through S-functions, code replacement libraries, Stateflow, and in MATLAB blocks. These calls are documented as calls to “External C Functions” on page 26-50. In these cases, you are responsible for encapsulation.

For additional information, see:

- “External C Functions” on page 26-50
- “Code Replacement Customization”
- “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2

Explanatory Note for Directive 4.6

Embedded Coder replaces basic data types with typedefs types, which are compatible with Directive 4.6. A guideline is not required because this behavior is default behavior in Embedded Coder. For additional information, see “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24 and Typedefs on page 26-62.

Explanatory Note for Directive 4.11

The requirements of this directive are satisfied by:

“Demonstrate statically that the input parameters can never take invalid values”.

You can use Polyspace Code Prover to analyze parameter ranges and prove the absence of run-time errors caused by out-of-range values. For additional information, see “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Code Prover).

Explanatory Note for Rule 2.1

Embedded Coder does not check user defined code from the Stateflow. You must take care of the violations reported.

Explanatory Note for Rules 5.1, 5.2, 5.4, 5.5, 5.6, 5.7, and 5.8

Embedded Coder is configurable to limit the number of characters imposed by the implementation. For additional information, see **Maximum identifier length**.

To ensure unique names for different types of variables (local scope variables, global scope variables, macros, and so on), implement a naming convention. For additional information, see **Model Configuration Parameters: Code Generation Symbols**.

In case of long identifiers, it is recommended to use shared utilities. For more information, see “Generate Shared Utility Code”

Explanatory Note for Rule 8.12

Embedded Coder supports the use of enumerated data. The file used to define the enumeration can be either manually or automatically generated. Files defining enumerations generated by Embedded Coder are compliant with MISRA C:2012 Rule 8.12 by design. If you manually create the definition

file, you are responsible for ensuring compliance. For additional information, see “Use Enumerated Data in Simulink Models”.

Explanatory Note for Rule 11.8

If your model uses `Simulink.Signal` or `Simulink.Parameter` that has storage class with `Const` or `Volatile` qualifiers, Polyspace Checker may show violation of Rule 11.8 from the generated code of the model.

Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012

The “MISRA C:2012 Compliance Information Summary Tables” on page 24-6 identifies modeling guidelines that are relevant to the compliance of generated C code with MISRA C:2012 coding standards. For a list of these guidelines and their corresponding Model Advisor check, see “High-Integrity System Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012” on page 24-24. For mapping of MISRA C:2012 Model Advisor checks to the MISRA C:2012 rules or directives, see “MISRA C:2012 Model Advisor Checks Rationale” on page 24-20.

MISRA C:2012 Model Advisor Checks Rationale

This table provides MISRA C:2012 rationale for the MISRA C:2012 Model Advisor checks.

Model Advisor Check		MISRA C:2012 Rule or Directive
“Check configuration parameters for MISRA C:2012” (Simulink Check)		
	Set **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`.	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.
	Set **Inf or NaN block output** to `warning` or `error`.	MISRA C:2012 Directive 4.1: Run-time failures shall be minimized
	Set **Model Verification block enabling** to `Disable All`.	General recommendation for embedded systems.
	Set **Undirected event broadcasts** to `error`.	MISRA C:2012 Rule 17.2: Functions shall not call themselves, either directly or indirectly
	Set configuration parameter **Wrap on overflow** to `warning` or `error`.	MISRA C:2012 Directive 4.1: Run-time failures shall be minimized
	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.
	Clear **Shift right on a signed integer as arithmetic shift** .	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.
	Set **Compile-time recursion limit for MATLAB functions** to `0` .	MISRA C:2012 Rule 17.2: Functions shall not call themselves, either directly or indirectly
	Clear **Dynamic memory allocation in MATLAB functions**.	MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used. MISRA C:2012 Rule 21.3: The memory allocation and deallocation functions of <stdlib.h> shall not be used.
	Clear **Enable run-time recursion for MATLAB functions**.	MISRA C:2012 Rule 17.2: Functions shall not call themselves, either directly or indirectly

Model Advisor Check		MISRA C:2012 Rule or Directive
	Set **Bitfield declarator type specifier** to `uint_T`.	MISRA C:2012 Rule 6.1: Bit-fields shall only be declared with an appropriate type MISRA C:2012 Rule 6.2: Single-bit named bit fields shall not be of a signed type
	Set **Casting Modes** to `Standards Compliant`.	MISRA C:2012 Rules 10.x: The essential type model
	Set **Code replacement library** to `None` or `AUTOSAR 4.0`	General recommendation for embedded systems.
	Clear **External mode**.	General recommendation for embedded systems. MISRA C:2012 Directive 4.12 Dynamic memory allocation shall not be used MISRA C:2012 Rule 21.3 The memory allocation and deallocation functions of <stdlib.h> shall not be used MISRA C:2012 Rule 21.6 The Standard Library input/output functions shall not be used
	Clear **Generate shared constants**.	MISRA Rule 8.5: An external object or function shall be declared once in one and only one file
	Clear **MAT-file logging**	General recommendation for embedded systems.
	Set **Maximum identifier length** to the implementation-dependent limit. The default is `31`.	MISRA C:2012 Rules 5.1-9: Identifiers
	Set **Parentheses level** to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.	MISRA C:2012 Rule 12.1: The precedence of operators within expressions should be made explicit
	Select **Preserve static keyword in function declarations**.	MISRA Rule 8.7: Functions and objects should not be defined with external linkage if they are referenced in only one translation unit MISRA Rule 8.8: The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
	Clear **Replace multiplications by powers of two with signed bitwise shifts**.	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type
	Set **Shared code placement** to `Shared location`	Prerequisite of configuration parameter **Generate shared constants**
	Clear **Support continuous time**.	General recommendation for embedded systems.
	Clear **Support non-finite numbers**	MISRA C:2012 Directive 4.1: Run-time failures shall be minimized

Model Advisor Check		MISRA C:2012 Rule or Directive
	Clear **Support non-inlined S-functions**.	General recommendation for embedded systems.
	Set **System-generated identifiers** to `Shortened`.	MISRA C:2012 5.1: External identifiers shall be distinct MISRA C:2012 5.2: Identifiers declared in the same scope and name space shall be distinct MISRA C:2012 5.4: Macro identifiers shall be distinct MISRA C:2012 5.5: Identifiers shall be distinct from macro names
	Set **System target file** to an ERT-based target.	General recommendation for embedded systems.
	Clear **Use dynamic memory allocation for model initialization**. Select only when **Code Interface Packaging** is set to `Reusable Function`.	MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used. MISRA C:2012 Rule 21.3: The memory allocation and deallocation functions of <stdlib.h> shall not be used.
	EnableSignedLeftShifts – off	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type
"Check for blocks not recommended for C/C++ production code deployment"		General recommendation for embedded systems.
"Check for blocks not recommended for MISRA C:2012"		
	Lookup Table blocks using cubic spline interpolation or extrapolation methods. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	MISRA C:2012 Rule 11.3: A cast shall not be performed between a pointer to object type and a pointer to a different object type. MISRA C:2012 Rule 11.5: A conversion should not be performed from pointer to void into pointer to object. MISRA C:2012 Rule 11.8: A cast shall not remove any const or volatile qualification from the type pointed to by a pointer. MISRA C:2012 Rule 11.9: The macro NULL shall be the only permitted form of integer null pointer constant. MISRA C:2012 Rule 12.1: The precedence of operators within expressions should be made explicit.

Model Advisor Check		MISRA C:2012 Rule or Directive
	Deprecated Lookup Table blocks. Specific blocks are: • Lookup Table • Lookup Table (2-D)	MISRA C:2012 Rule 11.3: A cast shall not be performed between a pointer to object type and a pointer to a different object type. MISRA C:2012 Rule 11.5: A conversion should not be performed from pointer to void into pointer to object. MISRA C:2012 Rule 11.8: A cast shall not remove any const or volatile qualification from the type pointed to by a pointer. MISRA C:2012 Rule 11.9: The macro NULL shall be the only permitted form of integer null pointer constant. MISRA C:2012 Rule 12.1: The precedence of operators within expressions should be made explicit. MISRA C:2012 Rule 12.2: The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.
	S-Function Builder blocks	MISRA C:2012 Rule 8.4: A compatible declaration shall be visible when an object or function with external linkage is defined. MISRA C:2012 Rule 8.5: An external object or function shall be declared once in one and only one file.
	From Workspace blocks	MISRA C:2012 Rule 18.4: The +, -, += and -= operators should not be applied to an expression of pointer type.
	String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	MISRA C:2012 Directive 4.7: If a function returns error information, then that error information shall be tested MISRA C:2012 Rule 17.7: The value returned by a function having non-void return type shall be used MISRA C:2012 Rule 21.6: The Standard Library input/output functions shall not be used
“Check for unsupported block names” (Simulink Check)		MISRA C:2012 Rule 3.1: The character sequences /* and // shall not be used within a comment.

Model Advisor Check	MISRA C:2012 Rule or Directive
“Check usage of Assignment blocks” (Simulink Check)	MISRA C:2012 Rule 9.1: The value of an object with automatic storage duration shall not be read before it has been set.
“Check for switch case expressions without a default case” (Simulink Check)	MISRA C:2012 Rule 16.4: Every switch statement shall have a default label.
“Check for missing error ports for AUTOSAR receiver interfaces” (Simulink Check)	MISRA C:2012 Directive 4.7 If a function returns error information, then that error information shall be tested. MISRA C:2012 Rule 17.7: The value returned by a function having non-void return type shall be used.
“Check for bitwise operations on signed integers” (Simulink Check)	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.
“Check for recursive function calls” (Simulink Check)	MISRA C:2012 Rule 17.2: Functions shall not call themselves, either directly or indirectly.
“Check for equality and inequality operations on floating-point values” (Simulink Check)	MISRA C:2012 Directive 1.1: Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.
“Check for missing const qualifiers in model functions” (Simulink Check)	MISRA C:2012 Rule 8.13: A pointer should point to a const-qualified type whenever possible.
“Check integer word length” (Simulink Check)	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.
“Check bus object names that are used as bus element names” (Simulink Check)	MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type. MISRA C:2012 Rule 5.6: A typedef name shall be a unique identifier.

High-Integrity System Modeling Guidelines and Model Advisor Checks for Verifying Compliance with MISRA C:2012

To augment the modeling guidelines developed by the MathWorks Advisory Board (MAB), MathWorks has published a set of modeling guidelines that focus on high-integrity applications.

Many high-integrity modeling guidelines have Model Advisor checks that you can use to verify adherence of your model to the guideline. This table identifies the high-integrity modeling guidelines and provides the corresponding Model Advisor check. Not all modeling guidelines have a corresponding Model Advisor check.

High-Integrity Guideline	Model Advisor Check
“hisl_0001: Usage of Abs block”	“Check usage of Abs blocks” (Simulink Check)
“hisl_0002: Usage of remainder and reciprocal operations”	“Check usage of remainder and reciprocal operations” (Simulink Check)

High-Integrity Guideline	Model Advisor Check
“hisl_0005: Usage of Product blocks”	Not applicable
“hisl_0006: Usage of While Iterator blocks”	“Check usage of While Iterator blocks” (Simulink Check)
“hisl_0008: Usage of For Iterator Blocks”	“Check usage of For Iterator blocks” (Simulink Check)
“hisl_0010: Usage of If blocks and If Action Subsystem blocks”	“Check usage of If blocks and If Action Subsystem blocks” (Simulink Check)
“hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks”	“Check usage of Switch Case blocks and Switch Case Action Subsystem blocks” (Simulink Check)
“hisl_0016: Usage of blocks that compute relational operators”	“Check relational comparisons on floating-point signals” (Simulink Check)
“hisl_0017: Usage of blocks that compute relational operators (2)”	“Check usage of Relational Operator blocks” (Simulink Check)
“hisl_0018: Usage of Logical Operator block”	“Check usage of Logical Operator blocks” (Simulink Check)
“hisl_0019: Usage of bitwise operations”	“Check usage of bit operation blocks” (Simulink Check)
“hisl_0020: Blocks not recommended for MISRA C:2012 compliance”	“Check for blocks not recommended for C/C++ production code deployment” (Simulink Check) “Check for blocks not recommended for MISRA C:2012” (Simulink Check)
“hisl_0029: Usage of Assignment blocks”	“Check usage of Assignment blocks” (Simulink Check)
“hisl_0032: Model element names”	“Check model object names” (Simulink Check)
“hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)”	“Check safety-related optimization settings for logic signals” (Simulink Check)
“hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values”	“Check safety-related optimization settings for data type conversions” (Simulink Check)
“hisl_0054: Configuration Parameters > Code Generation > Optimization > Remove code that protects against division arithmetic exceptions”	“Check safety-related optimization settings for division arithmetic exceptions” (Simulink Check)
“hisl_0060: Configuration parameters that improve MISRA C:2012 compliance”	“Check configuration parameters for MISRA C:2012” (Simulink Check)
“hisl_0061: Unique identifiers for clarity”	“Check Stateflow charts for uniquely defined data objects” (Simulink Check)
“hisl_0062: Global variables in graphical functions”	“Check global variables in graphical functions” (Simulink Check)

High-Integrity Guideline	Model Advisor Check
“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance”	“Check for length of user-defined object names” (Simulink Check)
“hisl_0101: Avoid operations that result in dead logic to improve code compliance”	Not applicable
“hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance”	“Check data type of loop control variables” (Simulink Check)
“hisl_0314: Configuration Parameters > Diagnostics > Data Validity > Signals”	“Check safety-related diagnostic settings for signal data” (Simulink Check)
“hisf_0004: Protect against recursive function calls to improve code compliance”	Not applicable
“hisf_0065: Type cast operations in Stateflow to improve code compliance”	“Check assignment operations in Stateflow Charts” (Simulink Check)
“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance”	“Check Stateflow charts for unary operators” (Simulink Check)

See Also

“MISRA C” on page 23-14 | “Model Advisor Checks for Verifying Compliance with Secure Coding Standards” on page 23-16

Deviations Rationale for MISRA C:2012 Compliance

Instances in which the generated code does not comply with the MISRA C:2012 guidelines are marked as deviations in the “MISRA C:2012 Compliance Information Summary Tables” on page 24-6. In accordance with Section 5.4 of the MISRA C:2012 guidelines,information about the deviations includes:

- The guideline being deviated
- The circumstances in which the deviation is permitted
- Justification for the deviation, including a risk assessment
- Demonstration of how safety is assured
- Potential consequences of non-conformance

Deviations to the MISRA C:2012 directives and rules include:

- “MISRA C:2012 Directive 4.7 Deviation Rationale” on page 24-27
- “MISRA C:2012 Rule 13.5 Deviation Rationale” on page 24-30

MISRA C:2012 Directive 4.7 Deviation Rationale

Directive Definition

If a function returns error information, then that error information shall be tested.

Rationale

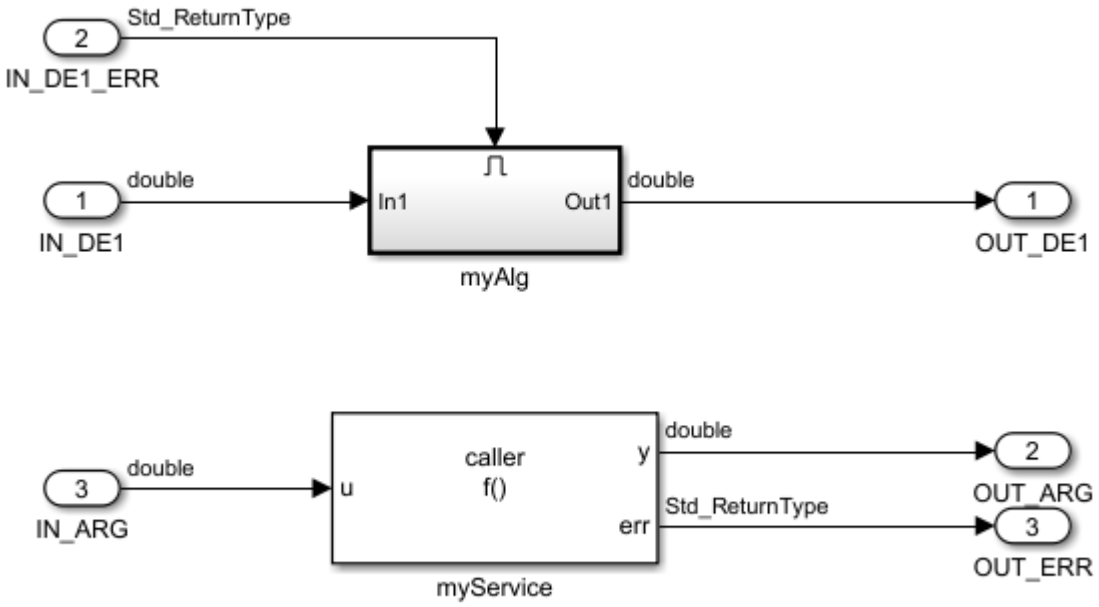
Note This directive is violated only when an AUTOSAR-based code generation target is used.

When a function call provides information that indicates an unsuccessful operation, the calling program has to check for the indication of an error when the function is returned. The function calls are not limited to calls in the standard library.

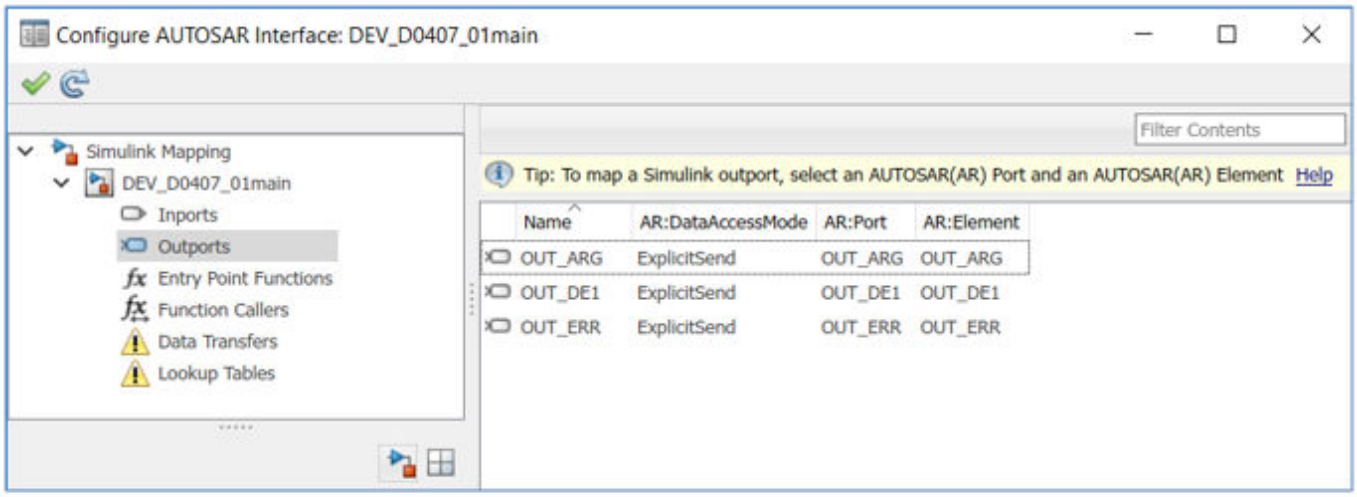
Description

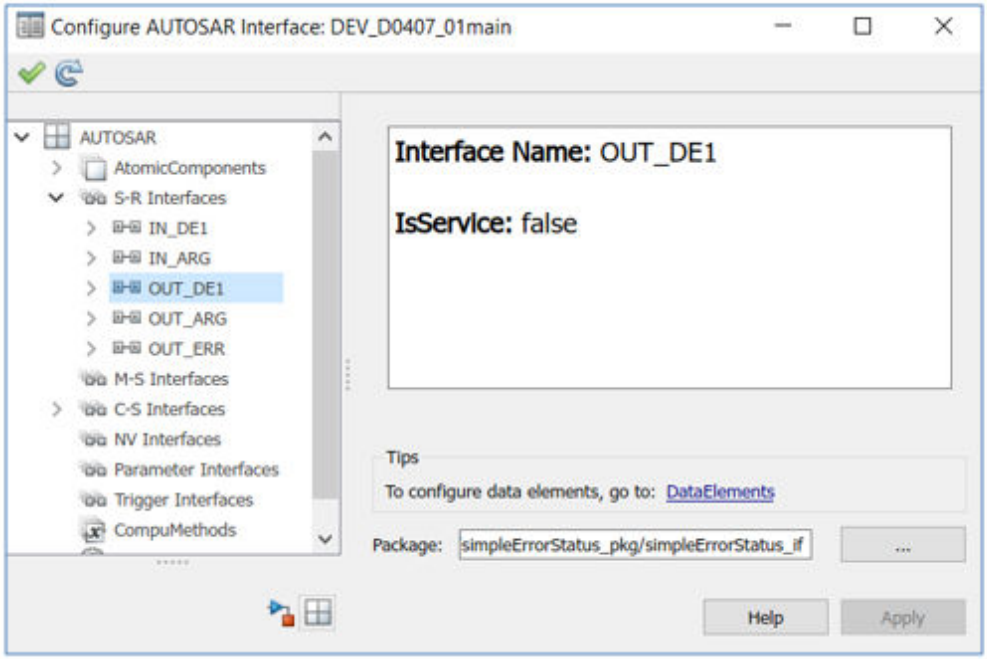
AUTOSAR sender interface function calls, such as `Rte_Write_...`, have return type `Std_ReturnType` that indicates if writing the value was successful. The matching Simulink pattern is a top-level output port that is configured as an AUTOSAR Sender Interface.

Model Example



AUTOSAR Configuration





Generated Code Sample

In the sample code, note the use of AUTOSAR function call `Rte_Write_...` in **`Rte_Write_OUT_DE1_OUT_DE1`**`(rtb_TmpSignalConversionAtIN_DE1);`.

```
void Runnable_Step(void)
{
  /* local block i/o variables */
  real_T rtb_TmpSignalConversionAtIN_DE1;
  …
  rtb_TmpSignalConversionAtIN_D_l = Rte_Read_IN_DE1_IN_DE1
    (&rtb_TmpSignalConversionAtIN_DE1);
  …
  if (rtb_TmpSignalConversionAtIN_D_l > 0) {
    …
```

```
    Rte_Write_OUT_DE1_OUT_DE1(rtb_TmpSignalConversionAtIN_DE1);
  }
  …
}
```

Justification

Failures of `Rte_Write_...` function calls are typically handled on the receiver side because function call returns do not always include all potential errors.

Conditions Under Which the Deviation is Requested

The deviation is requested for all instances of AUTOSAR sender ports.

Consequences of Noncompliance

The calling program cannot react when the sender operation is unsuccessful.

Actions to Control Reporting

When a Polyspace analysis of your code finds known or acceptable coding rule violations, you can suppress the violations in subsequent analyses by adding code annotations (Polyspace Bug Finder) to the Outport block. In future analyses, Polyspace hides results justified through annotations in the **Results List** pane. To review the coding rule violations, see “Filter and Group Results in Polyspace Desktop User Interface” (Polyspace Bug Finder).

If you have already provided annotations for MISRA C:2004 violations, Polyspace imports these justifications when you check your code for MISRA C:2012 violations. For additional information, see “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” (Polyspace Bug Finder).

For additional information, see:

- “Modeling Patterns for AUTOSAR Runnables” (AUTOSAR Blockset)
- “Develop a Model That Complies with the AUTOSAR Standard” on page 23-17
- “MISRA C:2012 Compliance Information Summary Tables” on page 24-6

MISRA C:2012 Rule 13.5 Deviation Rationale

Rule Definition

The right hand operand of a logical && or || operator shall not contain persistent side effects.

Rationale

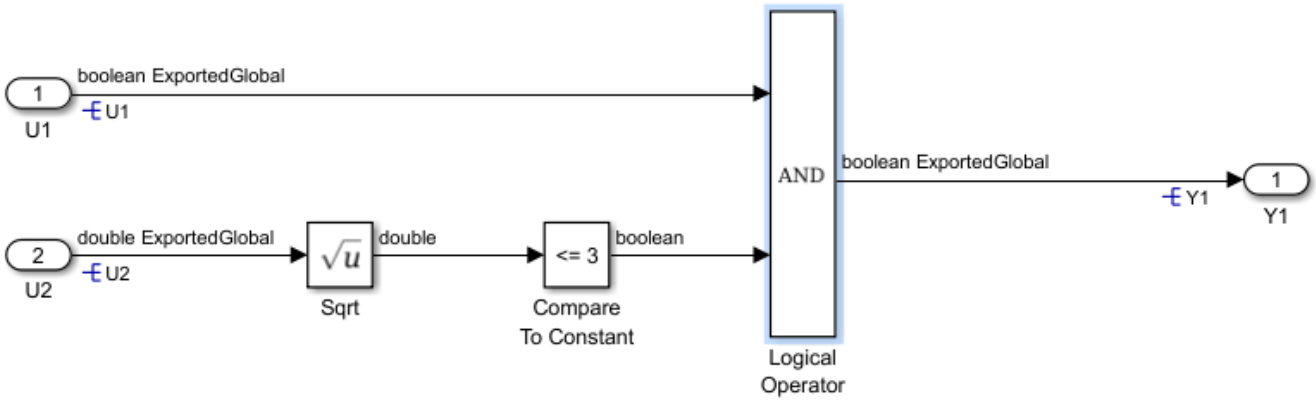
Persistent side effects of the right hand side operand of the && and || operators can occur depending on the left hand side operand, which is contrary to the expectations of the programmer.

Note The term *persistent side effect* is defined in "Appendix J: Glossary" of the *MISRA C:2012 Guidelines for the Use of C Language in Critical Systems* document.

Description

Using math operations as second input to Logical Operator blocks configured as AND or OR operation.

Model Example



Generated Code Sample

In the sample code, note the use of operand && in `Y1 = (U1 && (sqrt(U2) <= 3.0));`.

```
/* Exported block signals */
boolean_T U1;                /* '<Root>/U1' */
real_T U2;                   /* '<Root>/U2' */
boolean_T Y1;                /* '<Root>/Logical Operator' */

/* Model step function */
void DEV_R1305_01main_step(void)
{
  /* Logic: '<Root>/Logical Operator' incorporates:
   *  Constant: '<S1>/Constant'
   *  Inport: '<Root>/U1'
   *  Inport: '<Root>/U2'
   *  RelationalOperator: '<S1>/Compare'
   *  Sqrt: '<Root>/Sqrt'
   */
  Y1 = (U1 && (sqrt(U2) <= 3.0));
}
```

Justification

Some standard math functions have at least one persistent side effect of modifying the global `errno` variable, as defined in the C90 or C99 standard. Since the `errno` variable is not used by Embedded Coder, the second operand can be treated as not having persistent side effects.

Note Polyspace Bug Finder treats every function with no available source code as potentially having side effects.

Conditions Under Which the Deviation is Requested

This deviation is requested for all calls to standard math functions used as right hand side argument of && and || operators.

Consequences of Noncompliance

There are no consequences associated with noncompliance with MISRA C:2012 Rule 13.5 in the circumstances described in this deviation record. There are no additional verification and validation requirements resulting from this deviation.

Actions to Control Reporting

When a Polyspace analysis of your code finds known or acceptable coding rule violations, you can suppress the violations in subsequent analyses by adding code annotations (Polyspace Bug Finder) to the Logical Operator block. In future analyses, Polyspace hides results justified through annotations in the **Results List** pane. To review the coding rule violations, see “Filter and Group Results in Polyspace Desktop User Interface” (Polyspace Bug Finder).

If you have already provided annotations for MISRA C:2004 violations, Polyspace imports these justifications when you check your code for MISRA C:2012 violations. For additional information, see “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” (Polyspace Bug Finder).

For additional information, see “MISRA C:2012 Compliance Information Summary Tables” on page 24-6.

Modeling in Simulink Coder

- “Use Products and Blocks Supported for Code Generation” on page 25-2
- “Consider Modeling Semantics when Generating Code” on page 25-5
- “Modeling Guidelines for Blocks” on page 25-11
- “Modeling Guidelines for Subsystems” on page 25-12
- “Modeling Guidelines for Charts” on page 25-13
- “Modeling Guidelines for MATLAB Functions” on page 25-14
- “Modeling Guidelines for Model Configuration” on page 25-15

Use Products and Blocks Supported for Code Generation

As you construct a model, to prevent issues later in the development process and potential rework, determine whether the code generator supports products and blocks that you want to use.

Related Products

The following table summarizes MathWorks products that extend and complement Simulink CoderEmbedded Coder software. For information about these products and how they support code generation, see the product-specific documentation at `www.mathworks.com`.

Product	Examples of Code Generation Applications
Aerospace Blockset™	Aircraft, spacecraft, rocket, propulsion systems, and unmanned airborne vehicles
Audio Toolbox™	Audio processing systems
Automated Driving Toolbox™	Designing, simulating, and testing ADAS and autonomous driving systems
AUTOSAR Blockset	Modeling and simulation of AUTOSAR Classic and Adaptive ECU software
Communications Toolbox™	Physical layer of communication systems
Computer Vision Toolbox™	Video processing, image processing, and computer vision systems
Control System Toolbox™	Linear control systems
DSP System Toolbox™	Signal processing systems
Embedded Coder	Embedded systems, rapid prototyping boards, and microprocessors in mass production
Fixed-Point Designer	Fixed-point systems
Fuzzy Logic Toolbox™	System designs based on fuzzy logic
HDL Verifier™	Direct programming interface (DPI) component and transaction-level model (TLM) generation from Simulink
IEC Certification Kit	ISO 26262 and IEC 61508 certification
Model-Based Calibration Toolbox™	Developing processes for systematically identifying optimal balance of engine performance, emissions, and fuel economy, and reusing statistical models for control design, hardware-in-the-loop (HIL) testing, or powertrain simulation
Model Predictive Control Toolbox™	Model predictive controllers
Deep Learning Toolbox™	Neural networks
Parallel Computing Toolbox™	Parallel builds for large Simulink models
Phased Array System Toolbox™	Sensor array systems in radar, sonar, wireless communications, and medical imaging applications
Polyspace Bug Finder	MISRA-C compliance and static analysis of generated code

Product	Examples of Code Generation Applications
Polyspace Code Prover	Formal analysis of generated code
Powertrain Blockset™	Real-time testing of powertrain applications
Robotics System Toolbox™	Design, simulate, and test robotics applications
ROS Toolbox	Design, simulate, and deploy ROS-based applications
Simscape™	Systems spanning mechanical, electrical, hydraulic, and other physical domains as physical networks
Simscape Driveline™	Driveline (drivetrain) systems
Simscape Electrical™	Electronic, electromechanical, and electrical power systems
Simscape Fluids™	Hydraulic power and control systems
Simscape Multibody™	Three-dimensional mechanical systems
Simulink 3D Animation™	Systems with 3D visualizations
Simulink Check	Model standards compliance checking and metrics
Simulink Code Inspector™	Automated reviews of generated code
Simulink Control Design™	Autotuning of PID controllers and online frequency response estimation
Simulink Coverage	Model and code structural coverage analysis
Simulink Design Optimization™	Tuning adaptive lookup tables and checking signal characteristics
Simulink Desktop Real-Time™	Rapid prototyping or hardware-in-the-loop (HIL) simulation of control system and signal processing algorithms
Simulink Real-Time™	Rapid control prototyping, hardware-in-the-loop (HIL) simulation, and other real-time testing applications
Simulink Report Generator	Automatically generating project documentation in a standard format
Requirements Toolbox	Authoring and tracing requirements to design and code
Simulink Test	Software-in-the-loop (SIL), processor-in-the-loop (PIL), and real-time hardware-in-the-loop (HIL) testing of generated code
SoC Blockset™	Designing, evaluating, and implementing SoC hardware and software architectures
Stateflow	State machines and flow charts
System Identification Toolbox™	Systems identified from measured input-output data
Vehicle Dynamics Blockset™	Modeling and simulation of vehicle dynamics in 3D environment
Vehicle Network Toolbox™	CAN blocks for Accelerator and Rapid Accelerator simulations and code deployment on Windows

Blocks Supported for Code Generation

You can generate efficient code for a subset of Simulink built-in blocks and product-specific blockset blocks. The following tables list these blocks. In the tables, an information icon before the name of a block indicates that specific usage notes or limitations apply to code generation for that block. To view the usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom of the page and expand the **C/C++ Code Generation** section.

- Blocks Supported for C/C++ Code Generation (Category List)
- Blocks Supported for C/C++ Code Generation (Alphabetical List)

Simulink also provides a web page that summarizes characteristics of Simulink built-in blocks. The content, which is sourced from the Simulink block library, identifies blocks that are not recommended for production code generation and gives other usage guidelines. To view the page, in the MATLAB Command Window, type `showblockdatatypetable`.

For detailed information about a block, see the block reference page.

For modeling guidelines regarding blocks, see “Modeling Guidelines for Generated Code” on page 23-5.

For a list of blocks and block patterns that are not supported for models configured with a service code interface, see “Service Interface Constraints and Limitations” on page 3-29.

Block Set Support for Code Generation

Several products that include blocks are available for you to consider for code generation. However, before using the blocks for one of these products, consult the documentation for that product to confirm which blocks support code generation.

Consider Modeling Semantics when Generating Code

In this section...

"Data Propagation" on page 25-5
"Sample Time Propagation" on page 25-6
"Latches for Subsystem Blocks" on page 25-7
"Block Execution Order" on page 25-8
"Algebraic Loops" on page 25-9

Data Propagation

The first stage of code generation is compilation of the block diagram. This stage is analogous to that of a C or C++ program. The compiler carries out type checking and preprocessing. Similarly, the Simulink engine verifies that input/output data types of block ports are consistent, line widths between blocks are of expected thickness, and the sample times of connecting blocks are consistent.

The Simulink engine propagates data from one block to the next along signal lines. The data propagated consists of

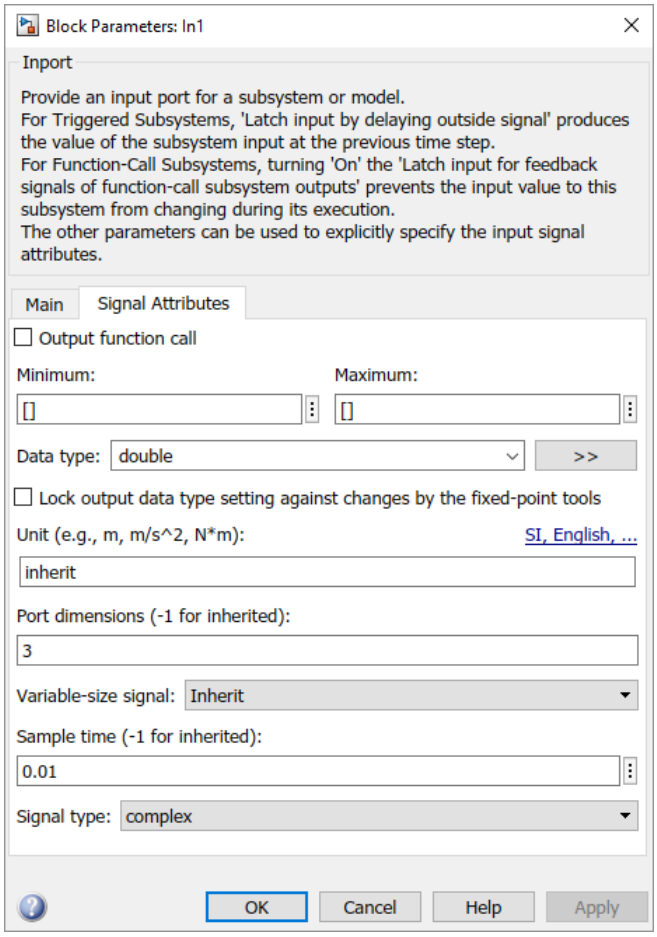
- Data type
- Line widths
- Sample times

You can verify what data types a Simulink block supports by typing

```
showblockdatatypetable
```

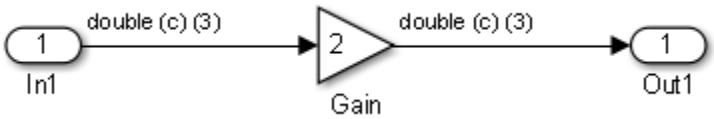
at the MATLAB prompt, or (from the Help browser) clicking the command above.

The Simulink engine typically derives signal attributes from a source block. For example, the Inport block's parameters dialog box specifies the signal attributes for the block.



In this example, the Inport block has a port width of 3, a sample time of .01 seconds, the data type is double, and the signal is complex.

This figure shows the propagation of the signal attributes associated with the Inport block through a simple block diagram.



In this example, the Gain and Outport blocks inherit the attributes specified for the Inport block.

Sample Time Propagation

Inherited sample times in source blocks (for example, a root inport) can sometimes lead to unexpected and unintended sample time assignments. Since a block may specify an inherited sample time, information available at the outset is often insufficient to compile a block diagram completely.

In such cases, the Simulink engine propagates the known or assigned sample times to those blocks that have inherited sample times but that have not yet been assigned a sample time. Thus, the engine continues to fill in the blanks (the unknown sample times) until sample times have been assigned to

as many blocks as possible. Blocks that still do not have a sample time are assigned a default sample time.

For a completely deterministic model (one where no sample times are set using the above rules), you should explicitly specify the sample times of your source blocks. Source blocks include root inport blocks and blocks without input ports. You do not have to set subsystem input port sample times. You might want to do so, however, when creating modular systems.

An unconnected input implicitly connects to ground. For ground blocks and ground connections, the sample time is always constant (`inf`).

All blocks have an inherited sample time (T_s = -1). They are assigned a sample time of (T_f - T_i)/50.

Blocks Whose Outputs Have Constant Values

When you display sample time colors, by default, Constant blocks appear magenta in color to indicate that the block outputs have constant values during simulation. Downstream blocks whose output values are also constant during simulation, such as Gain blocks, similarly appear magenta if they use an inherited sample time. The code generated for these blocks depends in part on the tunability of the block parameters.

If you set model configuration parameter **Default parameter behavior** to `Inlined`, the block parameters are not tunable in the generated code. Because the block outputs are constant, the code generator eliminates the block code due to constant folding. If the code generator cannot fold the code, or if you select settings to disable constant folding, the block code appears in the model initialization function. The generated code is more efficient because it does not compute the outputs of these blocks during execution.

However, if you configure a block or model so that the block parameters appear in the generated code as tunable variables, the code generator represents the blocks in a different way. Block parameters are tunable if, for example:

- You set parameter **Default parameter behavior** to `Tunable`. By default, numeric block parameters appear as tunable fields of a global parameter structure.
- You use a tunable parameter, such as a `Simulink.Parameter` object that uses a storage class other than `Auto`, as the value of one or more numeric block parameters. These block parameters are tunable regardless of the setting that you choose for model configuration parameter **Default parameter behavior**.

If a block parameter is tunable, the generated code must compute the block outputs during execution. Therefore, the block code appears in the model `step` function. If the model uses multiple discrete rates, the block code appears in the output function for the fastest downstream rate that uses the block outputs.

Latches for Subsystem Blocks

When an Inport block is the signal source for a triggered or function-call subsystem, you can use latch options to preserve input values while the subsystem executes. The Inport block latch options include:

For	Use
Triggered subsystems	**Latch input by delaying outside signal**

For	Use
Function-call subsystems	**Latch input for feedback signals of function-call subsystem outputs**

When you select subsystem block parameter **Latch input for feedback signals of function-call subsystem outputs** for a function-call subsystem, the code generator

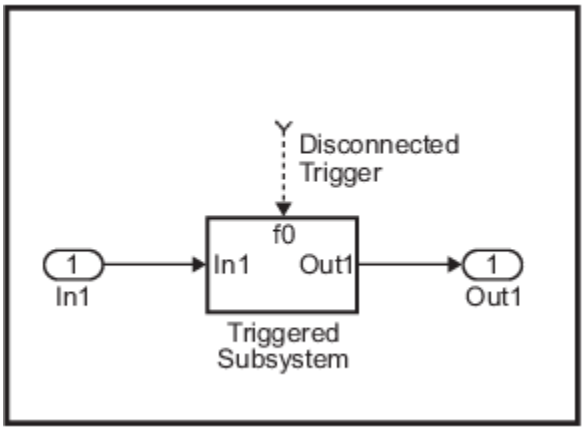
- Preserves latches in generated code regardless of optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

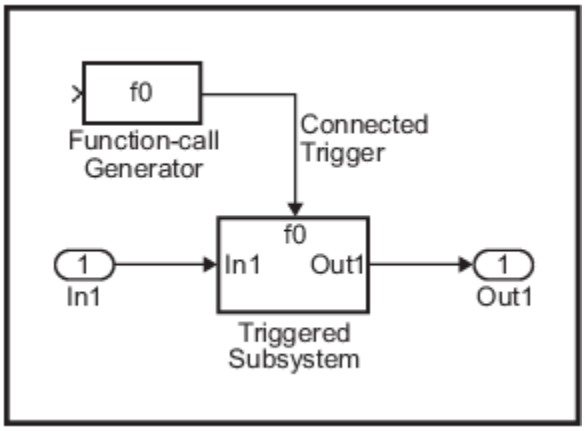
For more information on these options, see the block description of Inport.

Block Execution Order

Once the Simulink engine compiles the block diagram, it creates a *`model`*`.rtw` file (analogous to an object file generated from a C or C++ file). The *`model`*`.rtw` file contains the connection information of the model, as well as the signal attributes. Thus, the timing engine in can determine when blocks with different rates should be executed.

You cannot override this execution order by directly calling a block (in handwritten code) in a model. For example, in the next figure the `disconnected_trigger` model on the left has its trigger port connected to ground, which can lead to the blocks inheriting a constant sample time. Calling the trigger function, `f()`, directly from user code does not work. Instead, you should use a function-call generator to specify the rate at which `f()` should be executed, as shown in the `connected_trigger` model on the right.





Instead of the function-call generator, you could use another block that can drive the trigger port. Then, you should call the model's main entry point to execute the trigger function.

For multirate models, a common use of the code generator is to generate code for individual models separately and then manually code the I/O between the generated code modules. This approach places the burden of data consistency between models on the developer of the models. Another approach is to let Simulink and the code generator maintain data consistency between rates and generate multirate code for use in a multitasking environment. The Rate Transition block is able to interface periodic and asynchronous signals. For a description of the Simulink Coder block libraries, see “Asynchronous Events”. For more information on multirate code generation, see “Modeling for Multitasking Execution”.

Algebraic Loops

Algebraic loops are circular dependencies between variables. This prevents the straightforward direct computation of their values. For example, in the case of a system of equations

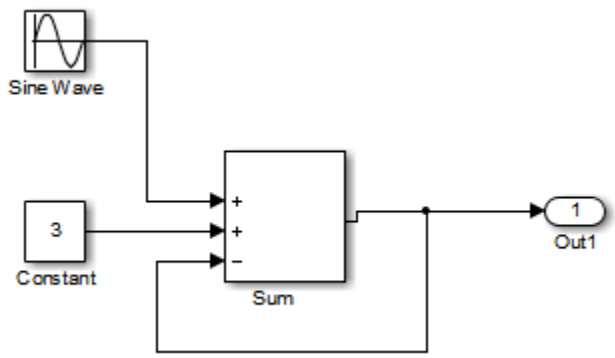
- `x = y + 2`
- `y = -x`

the values of `x` and `y` cannot be directly computed.

To solve this, either repeatedly try potential solutions for `x` and `y` (in an intelligent manner, for example, using gradient based search) or "solve" the system of equations. In the previous example, solving the system into an explicit form leads to

- `2x = 2`
- `y = -x`
- `x = 1`
- `y = -1`

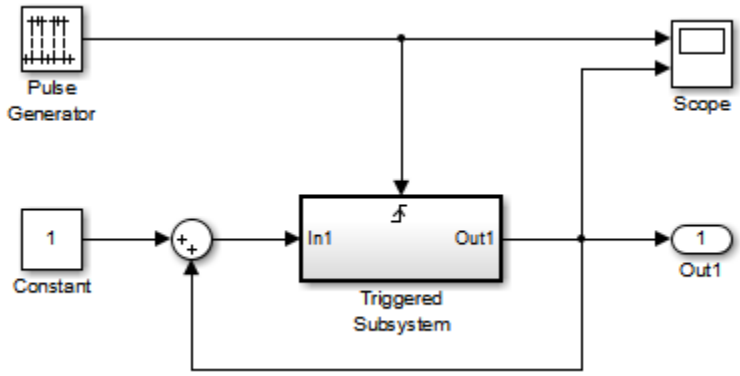
An algebraic loop exists whenever the output of a block having direct feedthrough (such as Gain, Sum, Product, and Transfer Fcn) is fed back as an input to the same block. The Simulink engine is often able to solve models that contain algebraic loops, such as the next diagram.



The code generator does not produce code that solves algebraic loops. This restriction includes models that use Algebraic Constraint blocks in feedback paths. However, the Simulink engine can often eliminate algebraic loops that arise, by grouping equations in certain ways in models that contain them. It does this by separating the update and output functions to avoid circular dependencies. For details, see "Algebraic Loop Concepts".

Algebraic Loops in Triggered Subsystems

While the Simulink engine can minimize algebraic loops involving atomic and enabled subsystems, a special consideration applies to some triggered subsystems. An example for which code can be generated is shown in the following model and triggered subsystem.



The default Simulink behavior is to combine output and update methods for the subsystem, which creates an apparent algebraic loop, even though the Unit Delay block in the subsystem has no direct feedthrough.

You can allow the Simulink engine to solve the problem by splitting the output and update methods of triggered and enabled-triggered subsystems when feasible. If you want the code generator to take advantage of this feature, select subsystem block parameter **Minimize algebraic loop occurrences**. Select this parameter to avoid algebraic loop warnings in triggered subsystems involved in loops.

Note If you check this box, the generated code for the subsystem might contain split output and update methods, even if the subsystem is not actually involved in a loop. Also, if a direct feedthrough block (such as a Gain block) is connected to the inport in the above triggered subsystem, the Simulink engine cannot solve the problem, and the code generator is unable to generate code.

A model configuration parameter **Minimize algebraic loop occurrences** is available also. Selecting it enables the code generator to produce code for models containing Model blocks that are involved in algebraic loops.

Modeling Guidelines for Blocks

Code generation modeling guidelines include recommended model settings, block usage, and block parameters. When you develop models for code generation, use these guidelines.

Code Generation Modeling Guidelines	“cgsl_0101: Zero-based indexing” “cgsl_0102: Evenly spaced breakpoints in lookup tables” “cgsl_0103: Precalculated signals and parameters” “cgsl_0201: Redundant Unit Delay and Memory blocks”

For more information, see “Model Advisor Checks for High-Integrity Systems Modeling Guidelines” and “Using Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

See Also

“Modeling Guidelines for Subsystems” on page 25-12 | “Modeling Guidelines for Charts” on page 25-13 | “Modeling Guidelines for MATLAB Functions” on page 25-14 | “Modeling Guidelines for Model Configuration” on page 25-15

Modeling Guidelines for Subsystems

When you develop models and generate code for subsystems, use the modeling guideline recommendations.

Code Generation Modeling Guidelines	“cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks”
High-Integrity Systems Modeling Guidelines	hisl_0007: Usage of For Iterator or While Iterator subsystems “hisl_0010: Usage of If blocks and If Action Subsystem blocks” “hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” “hisl_0023: Verification of variant blocks”
MathWorks Advisory Board (MAB) Control Algorithm Guidelines	db_0042: Usage of Inport and Outport blocks db_0081: Unconnected signals and blocks db_0143: Usable block types in model hierarchy db_0144: Use of subsystems db_0146: Block layout in conditional subsystems jc_0201: Usable characters for subsystem names jc_0231: Usable characters for block names jc_0281: Trigger signal names jm_0002: Block resizing na_0008: Display of labels on signals na_0009: Entry versus propagation of signal labels

For more information, see “Model Advisor Checks for High Integrity System Modeling Guidelines” (Simulink Check) and “Using Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

Modeling Guidelines for Charts

When you develop models and generate code for charts, use the modeling guideline recommendations.

High-Integrity Systems Modeling Guidelines	“hisf_0001: State Machine Type” “hisf_0002: User-specified state/transition execution order” “hisf_0011: Stateflow debugging settings” “hisf_0004: Protect against recursive function calls to improve code compliance” “hisf_0013: Usage of transition paths (crossing parallel state boundaries)” “hisf_0014: Usage of transition paths (passing through states)” “hisf_0015: Strong data typing (casting variables and parameters in expressions)”
MathWorks Advisory Board (MAB) Modeling Guidelines	db_0127: Limitation on MATLAB commands in Stateflow blocks jc_0451: Use of unary minus on unsigned integers jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow jc_0501: Format of entries in a State block jc_0511: Return values from a graphical function jc_0531: Default transition jm_0011: Pointers in Stateflow na_0001: Standard usage of Stateflow operators

For more information, see “Model Advisor Checks for High Integrity System Modeling Guidelines” (Simulink Check) and “Using Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

Modeling Guidelines for MATLAB Functions

When you develop models and generate code for MATLAB Functions, use the modeling guideline recommendations.

High-Integrity Systems Modeling Guidelines	“himl_0001: Usage of standardized MATLAB function headers” “himl_0002: Strong data typing at MATLAB function boundaries” “himl_0003: Complexity of user-defined MATLAB Functions”

For more information, see “Model Advisor Checks for High Integrity System Modeling Guidelines” (Simulink Check) and “Using Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

Modeling Guidelines for Model Configuration

When you develop models and generate code, use the modeling guideline configuration recommendations.

Code Generation Modeling Guidelines	“cgsl_0301: Prioritization of code generation objectives for code efficiency”
High-Integrity Systems Modeling Guidelines	“hisl_0043: Configuration Parameters > Diagnostics > Solver” “hisl_0044: Configuration Parameters > Diagnostics > Sample Time” “hisl_0301: Configuration Parameters > Diagnostics > Compatibility” “hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” “hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks” “hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization” “hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging” “hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” “hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” “hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls” “hisl_0309: Configuration Parameters > Diagnostics > Type Conversion” “hisl_0310: Configuration Parameters > Diagnostics > Model Referencing” “hisl_0311: Configuration Parameters > Diagnostics > Stateflow”

For more information, see “Model Advisor Checks for High Integrity System Modeling Guidelines” (Simulink Check) and “Using Model Advisor Checks for JMAAB Modeling Guidelines” (Simulink Check).

See Also

“Modeling Guidelines for Blocks”

Patterns for C Code in Embedded Coder

- “Prepare a Model for Code Generation” on page 26-2
- “Definition, Initialization, and Declaration of Parameter Data” on page 26-5
- “Definition and Declaration of Signal Data” on page 26-7
- “Data Type Conversion” on page 26-9
- “Type Qualifiers” on page 26-12
- “Relational and Logical Operators” on page 26-14
- “Bitwise Operations” on page 26-17
- “Enumeration” on page 26-20
- “If-Else” on page 26-23
- “Switch” on page 26-27
- “For Loop” on page 26-31
- “While Loop” on page 26-36
- “Do While Loop” on page 26-41
- “Function Call” on page 26-45
- “Function Prototyping” on page 26-47
- “External C Functions” on page 26-50
- “Macro Definitions (#define)” on page 26-56
- “Conditional Inclusions (#if / #endif)” on page 26-58
- “Typedef” on page 26-62
- “Structures of Parameters” on page 26-64
- “Structures of Signals” on page 26-67
- “Nested Structures of Signals” on page 26-70
- “Bitfields” on page 26-74
- “Arrays for Parameters” on page 26-76
- “Arrays for Signals” on page 26-78
- “Pointers” on page 26-80

Prepare a Model for Code Generation

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

Model examples in “Modeling Patterns for C Code Constructs” have the following naming conventions:

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

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

Configure Input Ports, Output Ports, and Arbitrary Signals

1. Create a model in Simulink. For more information, see “Interactive Model Editing”.
2. On the **Modeling** tab, click **Model Data Editor**. You can also use the keyboard shortcut **Ctrl** + **Shift** + **E**.
3. Set the **Change view** drop-down list to `Design`.
4. Use the **Storage Class** column to apply a storage class to the signal.

   For example, apply the storage class `ExportedGlobal`, which makes the signal appear in the generated code as a separate global variable. The variable has the same name as the signal in the model.

Initialize and Configure States

1. In the Model Data Editor, on the **States** tab, use the **Initial Value** column to specify initial values for a block state (such as the state of a Unit Delay block).
2. Set the **Change view** drop-down list to `Code`.
3. Use the **Name** column to give the state a name.
4. Use the **Storage Class** column to apply a storage class to the state.

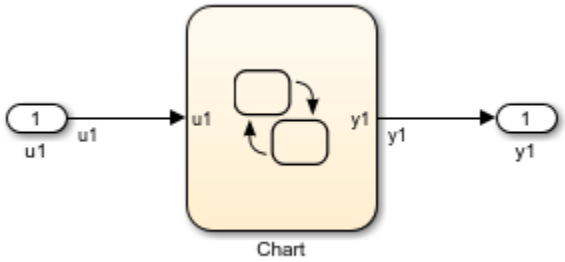
Set Up Configuration Parameters for Code Generation

1. On the **C Code** tab, click **Settings** to open the Configuration Parameters dialog box. You can also use the keyboard shortcut **Ctrl** + **E**.

Open the Configuration Parameter dialog box by selecting **Simulation** > **Model Configuration parameters**.

2. Open the **Solver** pane and select
   - **Solver type**: `Fixed-Step`
   - **Solver**: `discrete (no continuous states)`
3. Open the **Optimization** pane, and set **Default parameter behavior** to `Inlined`.
4. Open the **Code Generation** pane, and specify `ert.tlc` as the **System Target File**.
5. Clear **Generate makefile**.
6. Select **Generate code only**.
7. Enable the HTML report generation by opening the **Code Generation** > **Report** pane and selecting **Create code generation report** and **Open report automatically**. Click the horizontal ellipsis and, under **Advanced parameters**, select **Code-to-model**. Enabling the HTML report generation is optional.
8. Click **Apply** and then **OK** to exit.

Set Up an Example Model With a Stateflow Chart

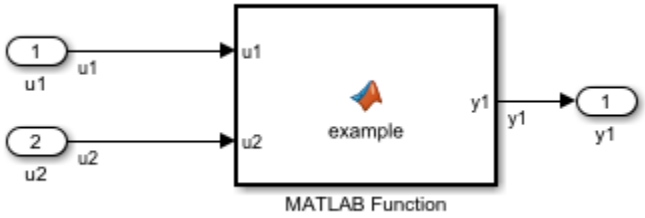


Follow this general procedure to create a simple model containing a Stateflow chart.

1. From the **Stateflow > Chart** library, add a Stateflow chart to your model .
2. Add Inport blocks and Outport blocks according to the example model.
3. Open the **Stateflow Editor** by performing one of the following:
   - Double-click the Stateflow chart.
   - Press **Ctrl+R**.
4. Add inputs to you chart as described in “Add Stateflow Data” (Stateflow).
5. Specify the **Name** (`u1, u2, ...`) and the **Type** (`Inherit: Same as Simulink`) for each input, unless specified differently in the example.
6. Add outputs to you chart as described in “Add Stateflow Data” (Stateflow).
7. Specify the **Name** (`y1, y2, ...`) and **Type** (`Inherit: Same as Simulink`) for each output, unless specified differently in the example.
8. In the **Stateflow Editor**, create the Stateflow diagram specific to the example.
9. The inputs and outputs appear on the chart in your model.
10. Connect the Inport and Outport blocks to the Stateflow Chart.

11 Configure the input and output signals; see “Configure Input Ports, Output Ports, and Arbitrary Signals” on page 26-2.

Set Up an Example Model With a MATLAB Function Block



1 Add the number of Inport and Outport blocks according to a C construct example included in this chapter.
2 From the Simulink User-defined Functions library drag a MATLAB Function block into the model.
3 Double-click the block. The MATLAB Function Block Editor opens. Edit the function to implement your application.
4 On the **Editor** tab, click **Save** and close the MATLAB Function Block Editor.
5 Connect the Inport and Outport blocks to the MATLAB Function block. See “Configure Input Ports, Output Ports, and Arbitrary Signals” on page 26-2.
6 Save your model.

Definition, Initialization, and Declaration of Parameter Data

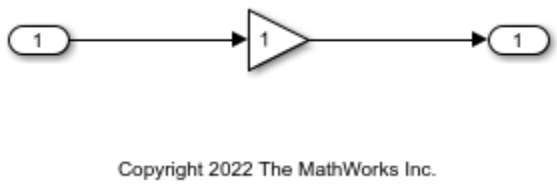
This example shows how to export the definition, initialization, and declaration of a global variable that the generated code uses as a parameter.

C Construct

```
int32 myParam = 3;

extern int32 myParam;
```

Procedure

1. Open the example model `ex_defn_decl`.



2. In the model, select the Gain block. In the Property Inspector, set the value of the **Gain** parameter to `myParam`.

3. Next to the parameter value, click the action button (the button with three vertical dots) and select **Create**.

4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(3)`. Click **Create**. A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the Gain parameter, in this case, 3.

5. In the Simulink.Parameter property dialog box, set **Data type** to `int32`.

6. Set **Storage class** to `ExportToFile`.

7. Set **Header File** to `myDecls.h`.

8. Set **Definition File** to `myDefns.c`. Click **OK**.

9. To build the model and generate code, press **Ctrl+B** .

Results

The generated header file `myDecls.h` declares the global variable `myParam` by using the `extern` keyword.

```
/* Declaration for custom storage class: ExportToFile */
extern int32_T myParam;                /* Referenced by: '<Root>/Gain' */
```

The generated source file `myDefns.c` defines and initializes `myParam`.

```
/* Definition for custom storage class: ExportToFile */
int32_T myParam = 3;                    /* Referenced by: '<Root>/Gain' */
```

See Also

Related Examples

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2

Definition and Declaration of Signal Data

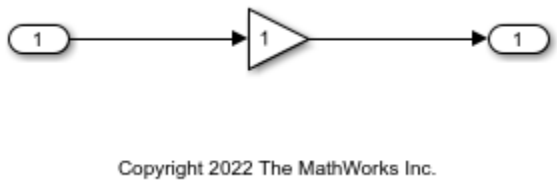
This example shows how to export the definition and declaration of a global variable that the generated code uses as a signal.

C Construct

```
float mySig;

extern float mySig;
```

Procedure

1. Open the example model `ex_defn_decl`.



2. In the Model Data Editor, open the **Inports/Outports** tab.

3. In the Model Data Editor, for the Inport block, set **Data Type** to `single`.

4. Open the **Embedded Coder** app.

5. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

6. In the Code Mappings editor, on the **Inports** tab, set **Storage class** for the Inport block to `ExportToFile`.

7. In the Property Inspector, set **Identifier** to `mySig`, **Header File** to `myDecls.h`, and **Definition File** to `myDefns.c`.

8. Generate code from the model.

Results

The generated header file `myDecls.h` declares the global variable `mySig` by using the `extern` keyword.

```
/* Declaration for custom storage class: ExportToFile */
extern real32_T mySig;                 /* '<Root>/In1' */
```

The generated source file `myDefns.c` defines and initializes `mySig`.

```
/* Definition for custom storage class: ExportToFile */
real32_T mySig;                        /* '<Root>/In1' */
```

See Also

Related Examples

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Signal Data for C Code Generation” on page 33-50

Data Type Conversion

This example shows how to create a data type conversion by using a Data Type Conversion block, Stateflow® Chart, or MATLAB® Function block.

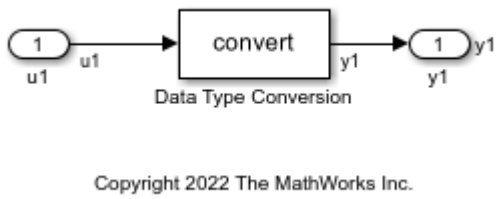
C Construct

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

Modeling Pattern for Data Type Conversion — Simulink Block

To create a data type conversion, use a Data Type Conversion block from the **Simulink > Commonly Used Blocks library**.

1. Open example model `ex_data_type_SL`.



2. Click the Inport block. In the Property Inspector, under **Signal Attributes**, specify **Data type** as `int32`.

3. Click the Data Type Conversion block. In the Property Inspector, specify the **Output data type** parameter as `double`.

4. In the Configuration Parameters dialog box, set the **Casting modes** parameter to `Explicit`. By default, Simulink® sets the **Casting modes** parameter to `Nominal` that generates minimal casting. For more details, see “Control Cast Expressions in Generated Code” on page 55-42.

5. To build the model and generate code, press **Ctrl+B** .

The generated code appears in `ex_data_type_SL.c`:

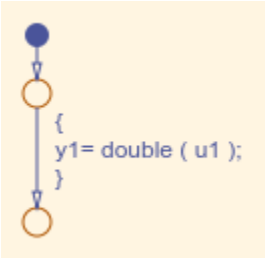
```
int32_T u1;                            /* '<Root>/u1' */
real_T y1;                             /* '<Root>/y1' */

/* Model step function */
void ex_data_type_SL_step(void)
{
  /* Outport: '<Root>/y1' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/u1'
   */
  y1 = (real_T)u1;
}
```

The code generator type definition for `double` is `real_T`.

Modeling Pattern for Data Type Conversion — Stateflow Chart

You can use a Stateflow chart in place of a Data Type Conversion block, to create a data type conversion.

1. Open example model `ex_data_type_SF`.



2. Click the Inport block. In the Property Inspector, under **Signal Attributes**, specify **Data type** as `int32`.

3. In the Configuration Parameters dialog box, set the **Casting modes** parameter to `Explicit`.

4. To build the model and generate code, press **Ctrl+B** .

The generated code appears in `ex_data_type_SF.c`:

```
int32_T u1;                             /* '<Root>/u1' */
real_T y1;                              /* '<Root>/Type_Conversion' */

/* Model step function */
void ex_data_type_SF_step(void)
{
  /* Chart: '<Root>/Type_Conversion' incorporates:
   *  Inport: '<Root>/u1'
   */
  y1 = (real_T)u1;
}
```

Modeling Pattern for Data Type Conversion — MATLAB Function Block

1. Open example model `ex_data_type_ML`.

2. The MATLAB Function Block contains this function:

```
function y1 = typeconv(u1)
```

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

3. Click the Inport block. In the Property Inspector, under **Signal Attributes**, specify **Data type** as `int32`.

4. In the Configuration Parameters dialog box, set the **Casting modes** parameter to `Explicit`.

5. To build the model and generate code, press **Ctrl+B** .

The generated code appears in `ex_data_type_ML.c`:

```
int32_T u1;                             /* '<Root>/u1' */
real_T y1;                              /* '<Root>/MATLAB Function' */

/* Model step function */
void ex_data_type_ML_step(void)
{
  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   */
  y1 = (real_T)u1;
}
```

Other Type Conversions in Modeling

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

See Also

Data Type Conversion

Related Examples

- “Parameter Data Types in the Generated Code” on page 36-131

Type Qualifiers

This example shows how to apply the `const` and `volatile` keywords to a global variable that represents parameter data.

C Construct

```
const volatile double myParam = 9.8;
```

Procedure

1. Open the example model `ex_const_volatile`.

```
open_system('ex_const_volatile');
```

2. In the model, select the Gain block. In the Property Inspector, set the value of the **Gain** parameter to `myParam`.

3. Next to the parameter value, click the action button (the button with three vertical dots) and select **Create**.

```
myParam = Simulink.Parameter(9.8);
set_param('ex_const_volatile/Gain','Gain','myParam');
```

4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(9.8)`. Click **Create**. A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the Gain parameter, in this case, 9.8.

5. On the **Code Generation** tab, click the **Configure in Coder App** button. In the Code Mappings editor, set **Storage Class** to `ConstVolatile`. Alternatively, to apply only one of the keywords, use the storage classes `Const` or `Volatile`.

```
myParam.StorageClass = 'ConstVolatile';
```

6. To build the model and generate code, press **Ctrl+B** .

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
evalc('slbuild(''ex_const_volatile'')');
```

Results

The generated source file `ex_const_volatile.c` defines `myParam` by using the `const` and `volatile` keywords.

```
file = fullfile('ex_const_volatile_ert_rtw','ex_const_volatile.c');
rtwdemodbtype(file,'/* Definition for custom', ...
    'const volatile real_T myParam = 9.8;',1,1);
```

```
/* Definition for custom storage class: ConstVolatile */
const volatile real_T myParam = 9.8;   /* Referenced by: '<Root>/Gain' */
```

See Also

Related Examples

- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75

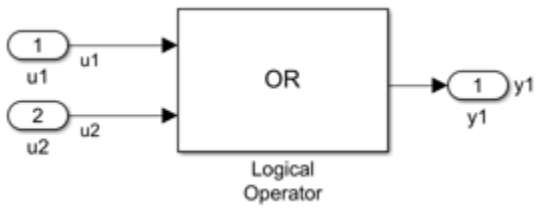
Relational and Logical Operators

This example shows how to implement relational and logical operators by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function blocks.

Modeling Pattern for Relational or Logical Operators — Simulink Blocks

To include a logical operation in your model, use the Logical Operator block from the **Logic and Bit Operations** library.

1. Open example model `ex_data_type_SL`.



The Logical Operator block performs an OR operation in the model. To change the operation, double-click the block and set the **Operator** field to any of the operations in the menu.

You can implement relational operators by replacing the Logical Operator block with a Relational Operator block.

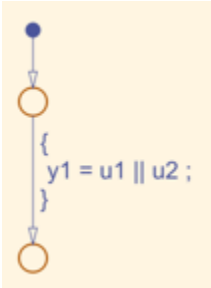
2. To build the model and generate code, press **Ctrl+B**.

The code implementing the logical operator OR is in the `ex_logical_SL_step` function in `ex_logical_SL.c`.

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

/* Model step function */
void ex_logical_SL_step(void)
{
  /* Outport: '<Root>/y1' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Logic: '<Root>/Logical Operator'
   */
  y1 = (u1 || u2);
}
```

Modeling Pattern for Relational and Logical Operators — Stateflow Chart

1. Open example model `ex_data_type_SF`.



In the Stateflow chart, the relational or logical operation actions are on the transition from one junction to another. Relational statements specify conditions to conditionally allow a transition. In that case, the statements are within square brackets.

2. To build the model and generate code, press **Ctrl+B**.

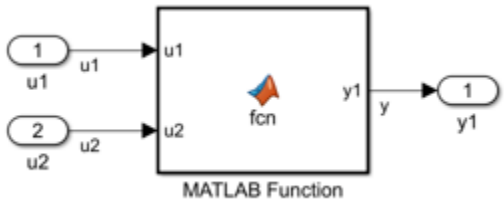
The code implementing the logical operator OR is in the `ex_logical_SF_step` function in `ex_logical_SF.c`.

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

/* Model step function */
void ex_logical_SF_step(void)
{
  /* Chart: '<Root>/Logical Operator' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  y1 = (u1 || u2);
}
```

Modeling Pattern for Relational and Logical Operators — MATLAB Function Block

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

1. Open example model `ex_logical_ML`.



2. The MATLAB Function Block contains this function:

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

```
end
```

3. To build the model and generate code, press **Ctrl+B**.

The generated code appears in `ex_data_type_ML.c`:

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

/* Model step function */
void ex_logical_ML_step(void)
{
  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  y = (u1 > u2);
}
```

See Also

Logical Operator

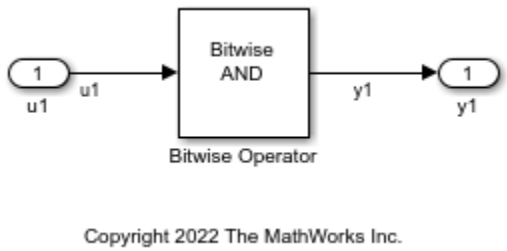
Bitwise Operations

This example shows how to implement bitwise operations by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function blocks.

Simulink Bitwise Operator Block

To include a logical operation in your model, use the Bitwise Operator block from the **Logic and Bit Operations** library.

1. Open example model `ex_bit_logic_SL`.



The Logical Operator blocks perform an AND operation in the model. To change the operation, double-click the block and set the **Operator** field to any of the operations in the menu.

2. Double-click the block to open the Block Parameters dialog box.

3. To perform bitwise operations with a bit-mask, select **Use bit mask**.

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

4. In the **Bit Mask** field, enter a decimal number. Use `bin2dec` or `hex2dec` to convert the input from binary or hexadecimal. In this example, enter `hex2dec('D9')`.

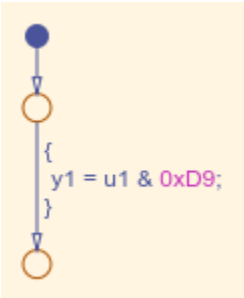
5. To build the model and generate code, press **Ctrl+B**.

The code implementing the bitwise operator AND is in the `ex_bit_logic_SL_step` function in `ex_bit_logic_SL.c`:

```
/* Exported block signals */
uint8_T u1;                            /* '<Root>/u1' */
uint8_T y1;                            /* '<Root>/Bitwise Operator' */

/* Model step function */
void ex_bit_logic_SL_step(void)
{
  /* S-Function (sfix_bitop): '<Root>/Bitwise Operator' incorporates:
   *  Inport: '<Root>/u1'
   */
  y1 = (uint8_T)(u1 & 217);
}
```

Stateflow Chart

1. Open example model `ex_bit_logic_SF`.



2. Right-click the Stateflow chart to open the chart **Properties**.

3. Verify that the **Enable C-bit operations** check box is selected. For more information, see “Enable C-bit operations” (Stateflow).

4. To build the model and generate code, press **Ctrl+B**.

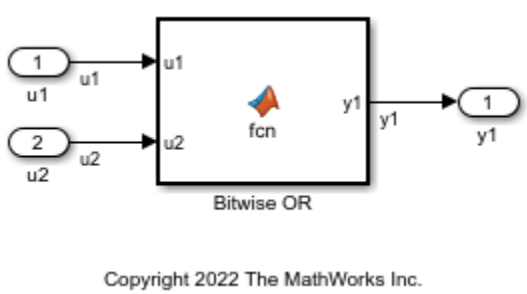
The code implementing the bitwise operator AND is in the `ex_bit_logic_SF_step` function in `ex_bit_logic_SF.c`:

```
/* Exported block signals */
uint8_T u1;                            /* '<Root>/u1' */
uint8_T y1;                            /* '<Root>/Bit_Logic' */

/* Model step function */
void ex_bit_logic_SF_step(void)
{
  /* Chart: '<Root>/Bit_Logic' incorporates:
   *  Inport: '<Root>/u1'
   */
  y1 = (uint8_T)(u1 & 0xD9);
}
```

MATLAB Function Block

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

1. Open example model `ex_bit_logic_ML`.



2. The MATLAB Function Block contains this function:

```
function y1 = fcn(u1, u2)
%#eml

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

3. To build the model and generate code, press **Ctrl+B**.

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

```
/* Exported block signals */
uint8_T u1;                            /* '<Root>/u1' */
uint8_T u2;                            /* '<Root>/u2' */
uint8_T y1;                            /* '<Root>/Bitwise OR' */

/* Model step function */
void ex_bit_logic_ML_step(void)
{
  /* MATLAB Function: '<Root>/Bitwise OR' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  y1 = (uint8_T)(u1 | u2);
}
```

See Also

Bitwise Operator

Enumeration

To generate an enumerated data type, define an enumeration class in a MATLAB® file. Then, use the enumeration class as the data type of signals, block parameters, and states in your model.

C Construct

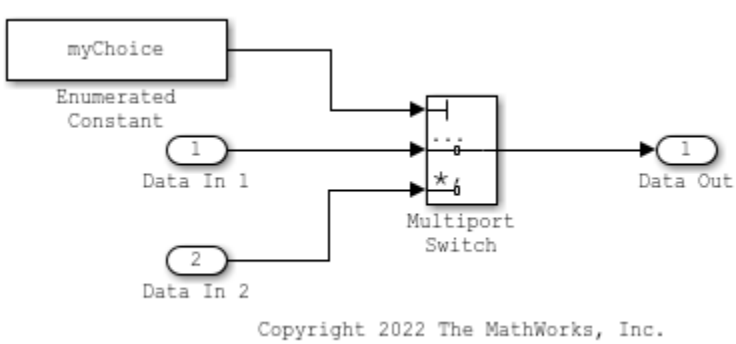
```
typedef enum {
  Choice1 = 0,
  Choice2
} myEnumType;
```

Procedure

In your current folder, create the MATLAB file `ex_myEnumType.m`. The file defines an enumeration class `ex_myEnumType`.

```
classdef ex_myEnumType < Simulink.IntEnumType

    enumeration
        Choice1(0)
        Choice2(1)
    end %enumeration

    methods (Static)
        function retVal = getHeaderFile()
            retVal = 'myEnumHdr.h';
        end %function

        function retVal = getDataScope()
            retVal = 'Exported';
        end %function
    end %methods

end %classdef
```

1. Open the model `ex_pattern_enum` that has an Enumerated Constant block and a Multiport Switch block.



2. In the Model Data Editor, select the **Parameters** tab.

3. In the model, click the Enumerated Constant block.

4. In the Model Data Editor, use the **Value** column to set the constant value to `myChoice`. Next to `myChoice`, click the action button (with three vertical dots) and select **Create**.

5. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(ex_myEnumType.Choice1)` and click **Create**. A `Simulink.Parameter` object named `myChoice` appears in the base workspace. The object stores the enumerated value `Choice1` of the type `ex_myEnumType`.

6. On the **Code Generation** tab, set **Storage Class** to `ExportedGlobal`. With this setting, the object appears in the generated code as a global variable.

7. In the Model Data Editor, select the **Signals** tab.

8. In the model, select the output signal of the Enumerated Constant block.

9. In the Model Data Editor, use the **Data Type** column to set the signal data type to `Enum: ex_myEnumType`.

10. Select the Multiport Switch block.

11. In the Property Inspector, set:

- **Data port order** to `Specify indices`.
- **Data port indices** to `enumeration('ex_myEnumType')`. This expression returns all of the enumeration members of `ex_myEnumType`.

12. Set the model configuration parameter **File packaging format** to `Modular`. With this setting, in the generated code, the definition of `ex_myEnumType` can appear in the specified header file, `myEnumHdr.h`.

13. To build the model and generate code, press **Ctrl+B** .

Results

View the generated header file `myEnumHdr.h`. The file defines the enumerated data type.

```
typedef enum {
  Choice1 = 0,                          /* Default value */
  Choice2
} ex_myEnumType;
```

View the source file `ex_pattern_enum.c`. The file defines the variable `myChoice`. The algorithm in the `step` function uses `myChoice` to route one of the input signals to the output signal.

```
ex_myEnumType myChoice = Choice1;      /* Variable: myChoice

/* Model step function */
void ex_pattern_enum_step(void)
{
  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Constant: '<S1>/Constant'
   */
```

```
  if (myChoice == Choice1) {
    /* Outport: '<Root>/Data Out' incorporates:
     *  Inport: '<Root>/Data In 1'
     */
    rtY.DataOut = rtU.DataIn1;
  } else {
    /* Outport: '<Root>/Data Out' incorporates:
     *  Inport: '<Root>/Data In 2'
     */
    rtY.DataOut = rtU.DataIn2;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */
}
```

See Also

`enumeration`

Related Examples

- “Use Enumerated Data in Simulink Models”
- “Use Enumerated Data in Generated Code” on page 36-86
- “Define Enumerated Data Types” (Stateflow)
- “Define Enumerations for MATLAB Function Blocks”

If-Else

This example shows how to implement an if-else construct by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function block.

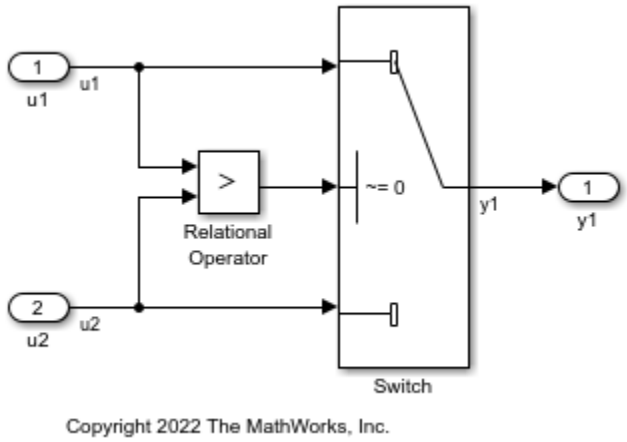
C Construct

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

Modeling Pattern for If-Else: Switch block

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

1. Open example model `ex_if_else_SL`.



The model contains the Switch block with the block parameter **Criteria for passing first input** of `u2~=0`. The software selects `u1` if `u2` is `TRUE`, otherwise `u2` passes.

2. To build the model and generate code, press **Ctrl+B**.

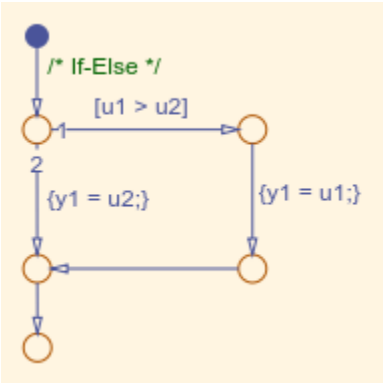
The code implementing the `if-else` construct is in the `ex_if_else_SL_step` function in `ex_if_else_SL.c`:

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

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

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

  /* End of Switch: '<Root>/Switch' */
}
```

Modeling Pattern for If-Else: Stateflow Chart

1. Open example model `ex_if_else_SF`.



The model contains an `If-Else` decision pattern that you add by right clicking inside the chart > **Add Pattern in Chart** > **Decision** > **If-Else**.

2. To build the model and generate code, press **Ctrl+B**.

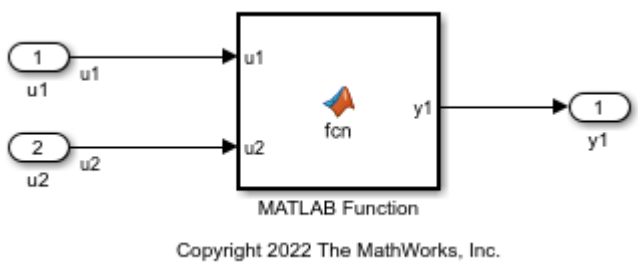
The code implementing the `if-else` construct is in the `ex_if_else_SF_step` function in `ex_if_else_SF.c`:

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

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

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

```
  /* Chart: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  /*  If-Else  */
  if (U.u1 >= U.u2) {
    /* Outport: '<Root>/y1' */
    Y.y1 = U.u1;
  } else {
    /* Outport: '<Root>/y1' */
    Y.y1 = U.u2;
  }

  /* End of Chart: '<Root>/Chart' */
}
```

Modeling Pattern for If-Else: MATLAB Function Block

1. Open example model `ex_if_else_ML`.



2. The MATLAB Function Block contains this function:

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

2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `if-else` construct is in the `ex_if_else_ML_step` function in `ex_if_else_ML.c`:

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

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

/* Model step function */
void ex_if_else_ML_step(void)
{
  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
```

```
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  if (U.u1 >= U.u2) {
    /* Outport: '<Root>/y1' */
    Y.y1 = U.u1;
  } else {
    /* Outport: '<Root>/y1' */
    Y.y1 = U.u2;
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

See Also

Related Examples

- “Switch” on page 26-27

Switch

This example shows how to implement a `switch` construct by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function block.

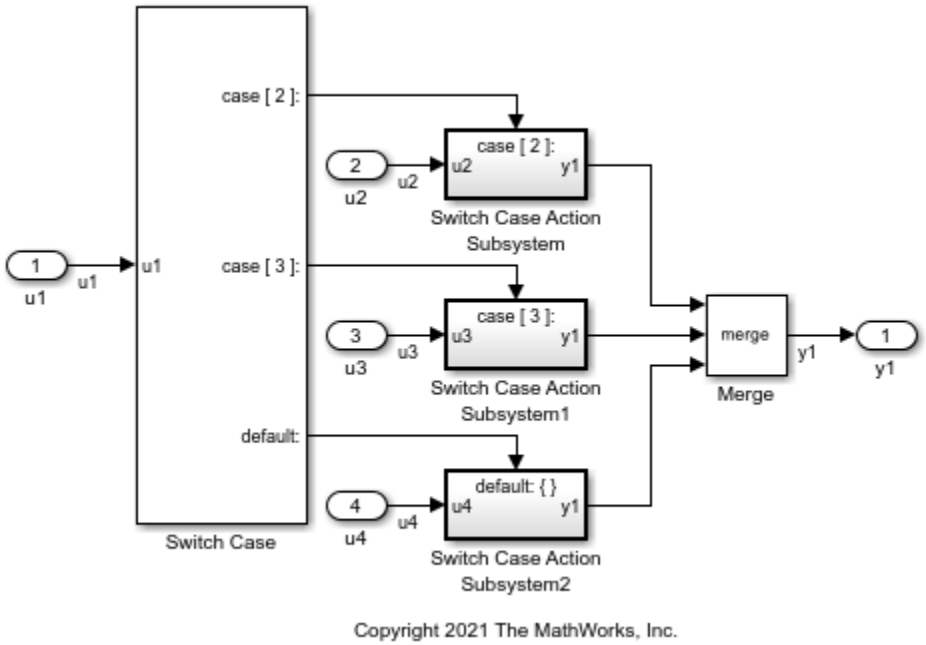
C Construct

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

Modeling Pattern for Switch: Switch Case block

One method to create a `switch` statement is to use a Switch Case block from the **Simulink > Signal Routing** library.

1. Open example model `ex_switch_SL`.



The model contains a Switch Case Action Subsystem. The Switch Case block takes an integer input, therefore, the input signal `u1` is type cast to `int32`.

2. To build the model and generate code, press **Ctrl+B**.

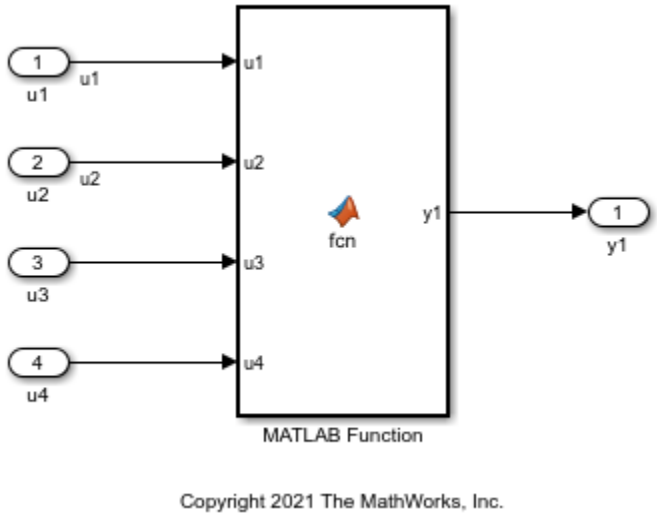
The code implementing the `switch` construct is in the `ex_switch_SL_step` function in `ex_switch_SL.c`:

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

/* External inputs (root inport signals with default storage) */
ExternalInputs rtU;

/* External outputs (root outports fed by signals with default storage) */
ExternalOutputs rtY;

/* Model step function */
void ex_switch_SL_step(void)
{
  /* SwitchCase: '<Root>/Switch Case' incorporates:
   *  Inport: '<Root>/u1'
   */
  switch (u1) {
   case 2:
    /* Outputs for IfAction SubSystem: '<Root>/Switch Case Action Subsystem' incorporates:
     *  ActionPort: '<S1>/Action Port'
     */
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u2'
     *  SignalConversion generated from: '<S1>/u2'
     */
    rtY.y1 = rtU.u2;

    /* End of Outputs for SubSystem: '<Root>/Switch Case Action Subsystem' */
    break;

   case 3:
    /* Outputs for IfAction SubSystem: '<Root>/Switch Case Action Subsystem1' incorporates:
     *  ActionPort: '<S2>/Action Port'
     */
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u3'
     *  SignalConversion generated from: '<S2>/u3'
     */
    rtY.y1 = rtU.u3;

    /* End of Outputs for SubSystem: '<Root>/Switch Case Action Subsystem1' */
    break;

   default:
    /* Outputs for IfAction SubSystem: '<Root>/Switch Case Action Subsystem2' incorporates:
     *  ActionPort: '<S3>/Action Port'
     */
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u4'
     *  SignalConversion generated from: '<S3>/u4'
     */
    rtY.y1 = rtU.u4;

    /* End of Outputs for SubSystem: '<Root>/Switch Case Action Subsystem2' */
    break;
```

```
  }

  /* End of SwitchCase: '<Root>/Switch Case' */
}
```

Modeling Pattern for Switch: MATLAB Function Block

1. Open example model `ex_switch_ML`.



The MATLAB Function Block contains this function:

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

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

2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `switch` construct is in the `ex_switch_ML_step` function in `ex_switch_ML.c`:

```
/* External inputs (root inport signals with default storage) */
ExternalInputs rtU;

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

```
/* Model step function */
void ex_switch_ML_step(void)
{
  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   */
  switch (rtU.u1) {
   case 2:
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u2'
     */
    rtY.y1 = rtU.u2;
    break;

   case 3:
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u3'
     */
    rtY.y1 = rtU.u3;
    break;

   default:
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u4'
     */
    rtY.y1 = rtU.u4;
    break;
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

Convert If-Elseif-Else to Switch statement

If a MATLAB Function block or a Stateflow chart uses `if-elseif-else` decision logic, you can convert the block or chart to a `switch` statement by using a configuration parameter. Select the **Configuration Parameters > Code Generation > Code Style > Convert if-elseif-else patterns to switch-case statements** parameter. For more information, see “Converting If-Elseif-Else Code to Switch-Case Statements” on page 55-105. For more information on this conversion by using a Stateflow chart, see “Enhance Readability of Code for Flow Charts” on page 55-95.

See Also

Switch Case

Related Examples

- “If-Else” on page 26-23
- “Enumeration” on page 26-20
- “Create Flow Charts by Using Pattern Wizard” (Stateflow)
- “Implement MATLAB Functions in Simulink with MATLAB Function Blocks”

For Loop

This example shows how to implement a `for` loop construct by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function blocks.

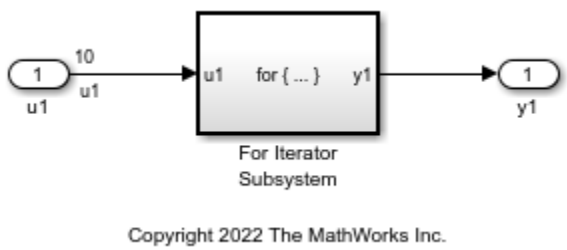
C Construct

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

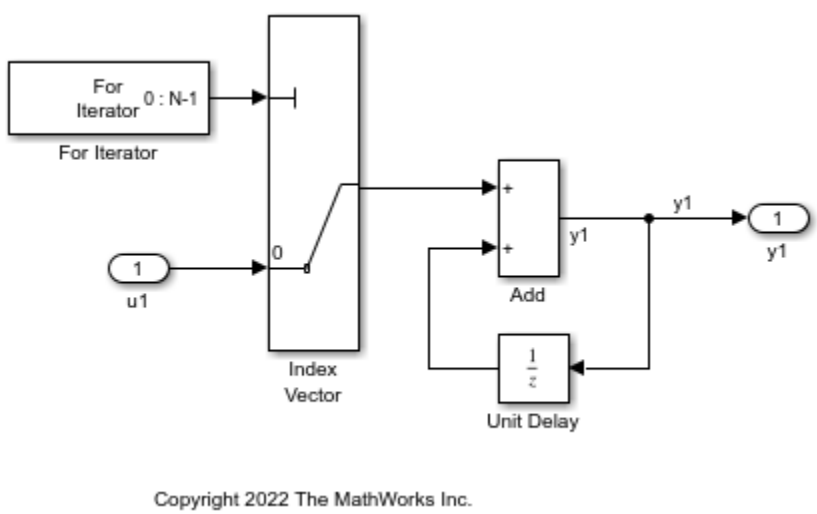
Modeling Pattern for For Loop: For-Iterator Subsystem block

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

1. Open example model `ex_for_loop_SL`.



The model contains a For Iterator Subsystem block that repeats execution of the contents of the subsystem during a simulation time step.



Observe the following settings in the model:

- Open the For Iterator block. In the Block Parameters dialog box, the **Index-mode** parameter is `Zero-based` and the **Iteration limit** parameter is 10.
- Open the Unit Delay block. In the Block Parameters dialog box, the **Initial Conditions** parameter is 0. This parameter initializes the state to zero.

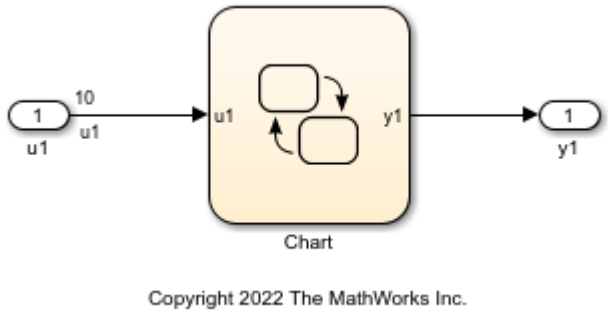
2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `for` loop is in the `ex_for_loop_SL_step` function in `ex_for_loop_SL.c`:

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

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

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

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

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

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

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

Modeling Pattern for For Loop: Stateflow Chart

1. Open example model `ex_for_loop_SF`.



The chart contains a `For` loop decision pattern that you add by right clicking inside the chart > **Add Pattern in Chart** > **Loop** > **For**.

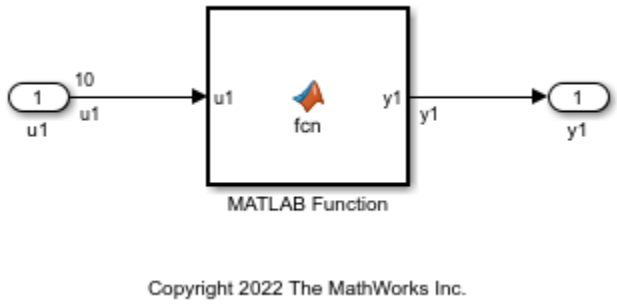
2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `for` loop is in the `ex_for_loop_SF_step` function in `ex_for_loop_SF.c`:

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

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

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

  /* Chart: '<Root>/Chart' */
  for (inx = 0; inx < 10; inx++) {
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u1'
     */
    Y.y1 += U.u1[inx];
  }

  /* End of Chart: '<Root>/Chart' */
}
```

Modeling Pattern for For Loop: MATLAB Function block

1. Open example model `ex_for_loop_ML`.



The MATLAB Function Block contains this function:

```
function y1 = fcn(u1)

y1 = 0;

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

2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `for` loop is in the `ex_for_loop_ML_step` function in `ex_for_loop_ML.c`:

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

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

  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   */
  y1 = 0.0;
  for (inx = 0; inx < 10; inx++) {
    y1 += u1[inx];
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

See Also

For Iterator Subsystem

Related Examples

- “Create Flow Charts by Using Pattern Wizard” (Stateflow)

More About

- “Implement MATLAB Functions in Simulink with MATLAB Function Blocks”

While Loop

This example shows how to implement a `while` loop construct by using Simulink® blocks, Stateflow® Charts, and MATLAB® Function blocks.

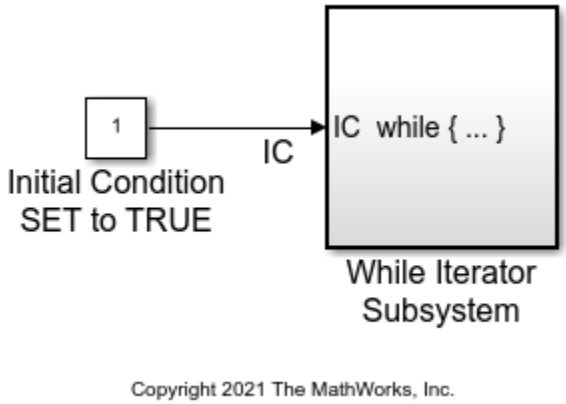
C Construct

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

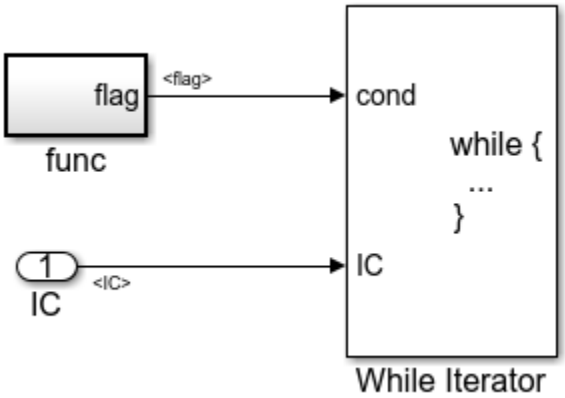
Modeling Pattern for While Loop: While Iterator Subsystem block

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

1. Open example model `ex_while_loop_SL`.



The model contains a While Iterator Subsystem block that repeats execution of the contents of the subsystem during a simulation time step.



Observe the following settings in the model:

- The Constant block provides an initial condition to the While Iterator Subsystem. For the Constant block, the **Constant value** is `1` and the **Output data type** is `boolean`. The initial condition can be dependent on the input to the block.
- In the While Iterator Subsystem, the `func` subsystem block has an output `flag` of `0` or `1` depending on the result of the algorithm in `func( )`. `func()` is the **Function name** in `func` subsystem.
- In the While Iterator Subsystem, for the While Iterator block, the **Maximum number of iterations** is `100`.
- For the While Iterator block, the **While loop type** is `while`.

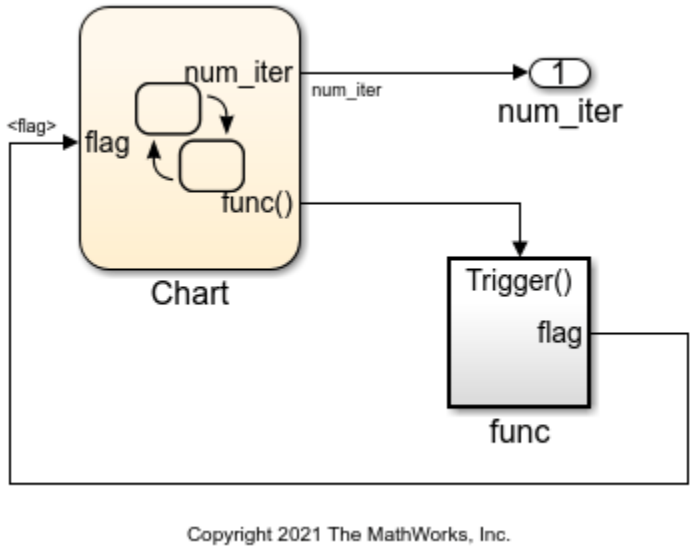
2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `while` loop is in the `ex_while_loop_SL_step` function in `ex_while_loop_SL.c`:

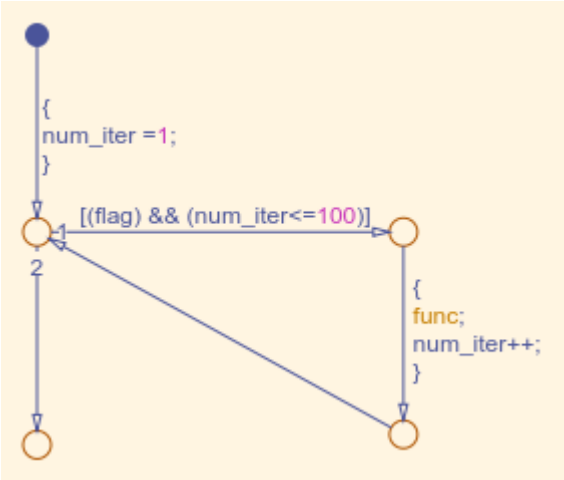
```
/* Model step function */
void ex_while_loop_SL_step(void)
{
  int32_T s1_iter;
  boolean_T loopCond;

  /* Outputs for Iterator SubSystem: '<Root>/While Iterator Subsystem' incorporates:
   *  WhileIterator: '<S1>/While Iterator'
   */
  s1_iter = 1;

  /* SystemReset for Atomic SubSystem: '<S1>/func' */
  func_Reset();

  /* End of SystemReset for SubSystem: '<S1>/func' */
  loopCond = true;
  while (loopCond && (s1_iter <= 100)) {
    /* Outputs for Atomic SubSystem: '<S1>/func' */
    func();

    /* End of Outputs for SubSystem: '<S1>/func' */
    loopCond = flag;
    s1_iter++;
  }

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

Modeling Pattern for While Loop: Stateflow Chart

1. Open example model `ex_while_loop_SF`.



In the model, the `ex_while_loop_SF/Chart` executes the `while` loop.



The chart contains a `While` loop decision pattern that you add by right clicking inside the chart > **Add Pattern in Chart** > **Loop** > **While**.

2. To build the model and generate code, press **Ctrl+B**.

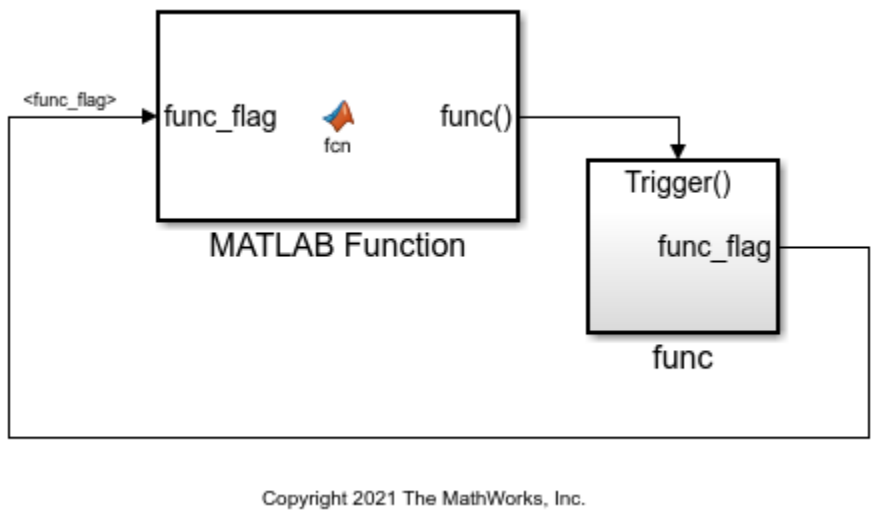
The code implementing the `while` loop is in the `ex_while_loop_SF_step` function in `ex_while_loop_SF.c`:

```
/* Model step function */
void ex_while_loop_SF_step(void)
{
  /* Chart: '<Root>/Chart' */
  num_iter = 1;
  while (flag && (num_iter <= 100)) {
    /* Outputs for Function Call SubSystem: '<Root>/func' */
```

```
    func();

    /* End of Outputs for SubSystem: '<Root>/func' */
    num_iter++;
  }

  /* End of Chart: '<Root>/Chart' */
}
```

Modeling Pattern for While Loop: MATLAB Function block

1. Open example model `ex_while_loop_ML`.



The MATLAB Function Block contains this function:

```
function fcn(func_flag)

flag = true;
num_iter = 1;

while(flag && (num_iter<=100))
    func;
    flag = func_flag;
    num_iter = num_iter + 1;
end
```

2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `while` loop is in the `ex_while_loop_ML_step` function in `ex_while_loop_ML.c`:

```
/* Model step function */
void ex_while_loop_ML_step(void)
{
  int32_T num_iter;
```

```
  boolean_T flag;
  boolean_T func_flag_0;

  /* MATLAB Function: '<Root>/MATLAB Function' */
  func_flag_0 = func_flag;
  flag = true;
  num_iter = 1;
  while (flag && (num_iter <= 100)) {
    /* Outputs for Function Call SubSystem: '<Root>/func' */
    func();

    /* End of Outputs for SubSystem: '<Root>/func' */
    flag = func_flag_0;
    num_iter++;
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

See Also

While Iterator Subsystem

Related Examples

- “Do While Loop” on page 26-41
- “Create Flow Charts by Using Pattern Wizard” (Stateflow)

More About

- “Implement MATLAB Functions in Simulink with MATLAB Function Blocks”

Do While Loop

This example shows how to implement a `do while` loop construct by using Simulink® blocks and Stateflow® Charts.

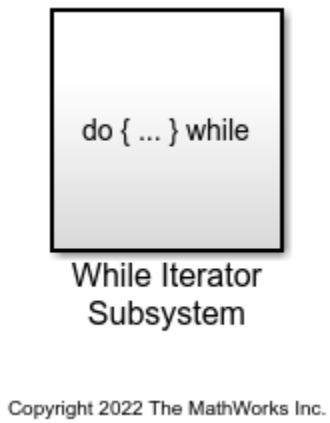
C Construct

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

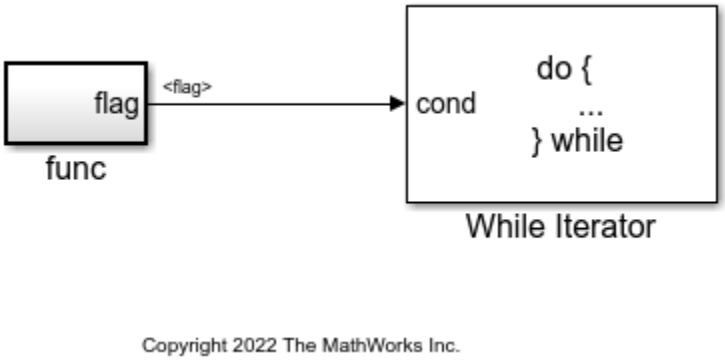
Modeling Pattern for Do While Loop: While Iterator Subsystem block

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

1. Open example model `ex_do_while_loop_SL`.



The model contains a While Iterator Subsystem block that repeats execution of the contents of the subsystem during a simulation time step.



Observe the following settings in the While Iterator Subsystem:

- The `func` subsystem block has an output `flag` of `0` or `1` depending on the result of the algorithm in `func( )`. `func()` is the **Function name** in `func` subsystem.
- For the While Iterator block, the **Maximum number of iterations** is `100`.
- For the While Iterator block, the **While loop type** is `do-while`.

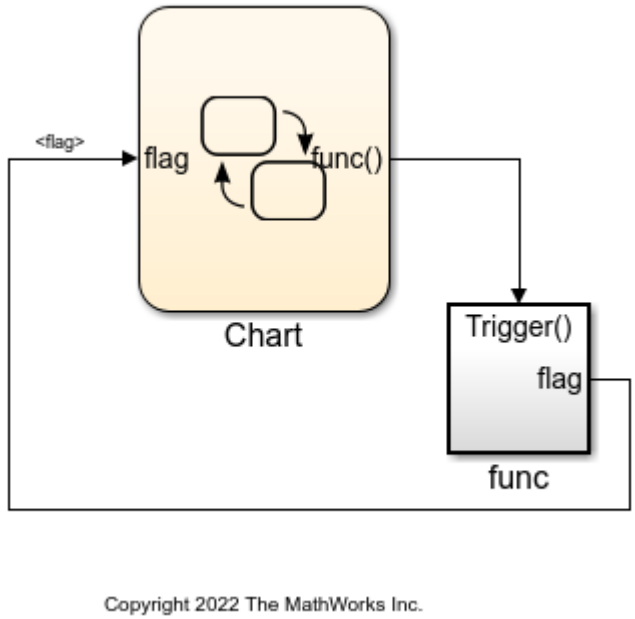
2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `do while` loop is in the `ex_do_while_loop_SL_step` function in `ex_do_while_loop_SL.c`:

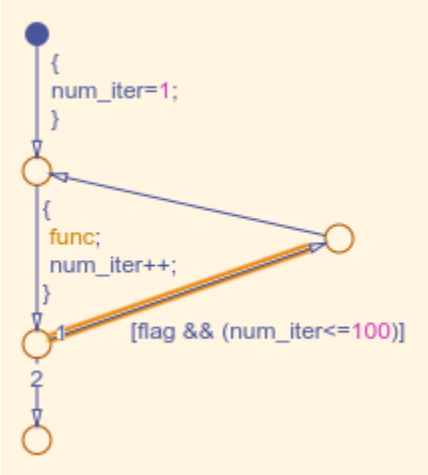
```
/* Model step function */
void ex_do_while_loop_SL_step(void)
{
  int32_T s1_iter;

  /* Outputs for Iterator SubSystem: '<Root>/While Iterator Subsystem' incorporates:
   *  WhileIterator: '<S1>/While Iterator'
   */
  s1_iter = 1;

  /* SystemReset for Atomic SubSystem: '<S1>/func' */
  func_Reset();

  /* End of SystemReset for SubSystem: '<S1>/func' */
  /* End of Outputs for SubSystem: '<Root>/While Iterator Subsystem' */
  do {
    func();
    s1_iter++;
  } while (flag && (s1_iter <= 100));
}
```

Modeling Pattern for Do While Loop: Stateflow Chart

1. Open example model `ex_do_while_loop_SF`.



In the model, the `ex_do_while_loop_SF/Chart` executes the `do while` loop.



The chart contains a `While` loop decision pattern that you add by selecting **Chart** > **Add Pattern in Chart** > **Loop** > **While**.

2. To build the model and generate code, press **Ctrl+B**.

The code implementing the `do while` loop is in the `ex_do_while_loop_SF_step` function in `ex_do_while_loop_SF.c`:

```
/* Model step function */
void ex_do_while_loop_SF_step(void)
{
  int32_T num_iter;
```

```
  /* Chart: '<Root>/Chart' */
  num_iter = 1;
  do {
    func();
    num_iter++;
  } while (flag && (num_iter <= 100));
}
```

See Also

While Iterator Subsystem

Related Examples

- “While Loop” on page 26-36
- “Create Flow Charts by Using Pattern Wizard” (Stateflow)

Function Call

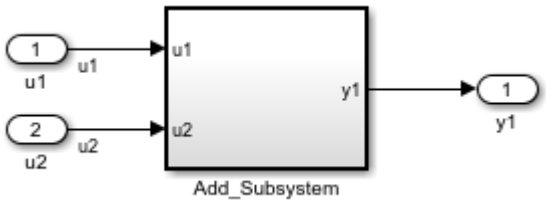
This example shows how to generate a function call by adding a subsystem, which implements the operations that you want.

C Construct

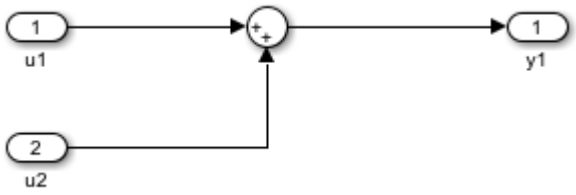
```
void add_function(void)
{
   y1 = u1 + u2;
}
```

Procedure

1. Open example model `ex_function_call`.



The subsystem has two inputs and returns one output.



Selecting the **Treat as atomic unit** parameter enables parameters on the **Code Generation** tab. The **Code Generation** tab provides these customizations:

- **Function packaging** set to `Nonreusable function`
- **Function name options** set to `User specified`
- **Function name** specified as `add_function`
- **File name options** set to `Use function name`

2. To build the model and generate code, press **Ctrl+B**.

Results

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

```
/* Model step function */
void ex_function_call_step(void)
```

```
{
  /* Outputs for Atomic SubSystem: '<Root>/Add_Subsystem' */
  add_function();

  /* End of Outputs for SubSystem: '<Root>/Add_Subsystem' */
}
```

The function prototype is externally declared through the subsystem file, `add_function.h`:

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

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

```
void add_function(void)
{
  /* Outport: '<Root>/y1' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Sum: '<S1>/Sum'
   */
  rtY.y1 = u1 + u2;
}
```

See Also

Function-Call Subsystem

Related Examples

- “Generate Reentrant Code from Subsystems”
- “Conditionally Executed Subsystems Overview”

Function Prototyping

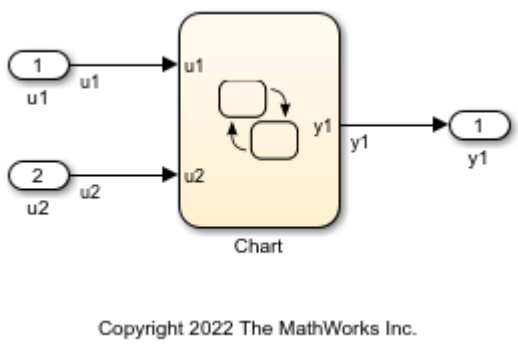
This example shows how to create and customize your function prototype.

C Construct

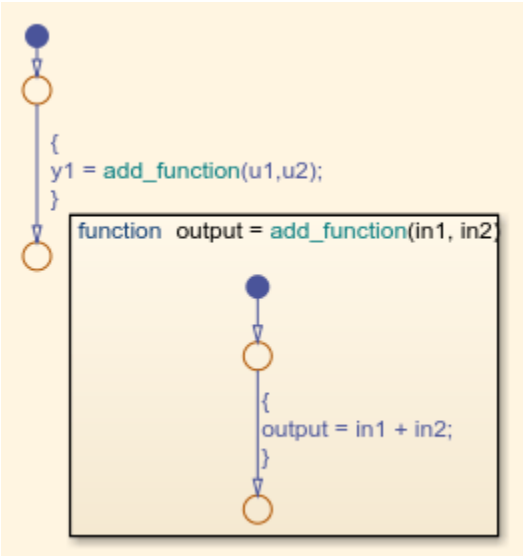
```
double add_function(double u1, double u2)
{
   return u1 + u2;
}
```

Function Call by Using Graphical Functions

1. Open example model `ex_func_SF`. This example model contains two Inport blocks and one Outport block.



The Stateflow® chart contains a graphical function that you create by clicking the **fx** button and placing a graphical function into the Stateflow chart.



The graphical function contains this signature:

```
output = add_function(u1, u2)
```

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

2. Open the Model Explorer. From the Model Hierarchy tree, select **ex_func_SF > Chart > add_function**. On the right pane, make sure that the **Function Inline Option** is set as `Function`. matlab 3. From the Model Hierarchy tree, click **Chart**. On the right pane make sure that the **Export Chart Level Functions** parameter is selected.

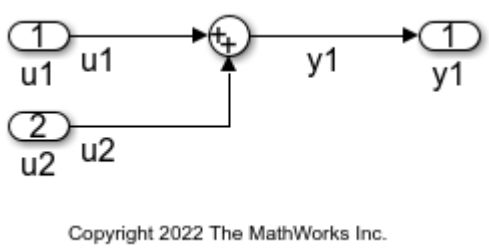
4. To build the model and generate code, press **Ctrl+B**.

`ex_func_SF.c` contains the generated code:

```
/* Function for Chart: '<Root>/Chart' */
void add_function(real_T in1, real_T in2, real_T *output)
{
  *output = in1 + in2;
}

/* Model step function */
void ex_func_SF_step(void)
{
  /* Chart: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  add_function(u1, u2, &y1);

  /* Outport: '<Root>/y1' */
  rtY.y1_e = y1;
}
```

Control Function Prototype of the *model*_step Function

1. Open example model `ex_control_step_function`.



2. Open the Code perspective. In the Code Mappings editor, select the **Entry-Point Functions** tab.

3. In the step function row, under the **Function Preview** column, click the prototype hyperlink. The configuration dialog box used for customization opens. The first field in the dialog box, **C function prototype**, automatically updates to preview the changes to the step function as you make them.

4. Change the function name. If you do not specify a name, the code generator names the function based on a default naming rule. For this example, rename the function by setting **C Step Function Name** to `ex_control_run`. The function preview changes to reflect the new step function name.

5. Configure step function arguments. If an application requires a `void-void` interface, clear **Configure arguments for Step function prototype**. For this example, select that option.

6. Display current default settings for the entry-point function arguments by clicking **Get default**. An updated diagram appears.

7. Configure the function return argument by setting **C return argument** to `void` or one of the listed outport arguments. For this example, select `void`. Check your change in the preview.

8. Configure the function input and output arguments. For each root inport and outport, you can change the C type qualifier and argument name. For details, see “Configure Name and Arguments for Individual Step Functions” on page 33-120.

For this example:

- For `u1`, set **C Type Qualifier** to `Value` and **C Identifier Name** to `arg_u1`.
- For `u2`, set **C Type Qualifier** to `Pointer to const` and **C Identifier Name** to `arg_u2`.
- For the outport, set **C Identifier Name** to `arg_y1`.

Review your changes in the preview.

9. To validate your changes, click **Validate**.

10. Apply your changes. Click **Apply**. Then, click **OK**.

11. To build the model and generate code, press **Ctrl+B**.

`ex_control_step_function.c` contains the generated code:

```
void ex_control_run(real_T arg_u1, const real_T *arg_u2, real_T *arg_y1)
```

See Also

Related Examples

- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Choose an Approach for Configuring Generated C Function Interfaces” on page 33-118
- “Reuse Logic Patterns by Defining Graphical Functions” (Stateflow)

External C Functions

This example shows several methods for integrating legacy C functions into generated code. These methods create an S-function or make a call to an external C function. For more information on S-functions, see “S-Functions and Code Generation” on page 13-2.

C Construct

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

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

Create S-Functions by Using Legacy Code Tool

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

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

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

2. Create a C source file named `add.c` that contains the function body:

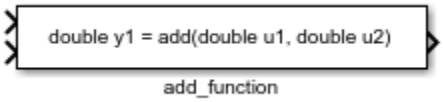
```
double add(double u1, double u2)
{
   double y1;
   y1 = u1 + u2;
   return (y1);
}
```

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

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

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

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

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

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

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

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

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

%% Create a Masked S-function Block
legacy_code('slblock_generate', def);
```

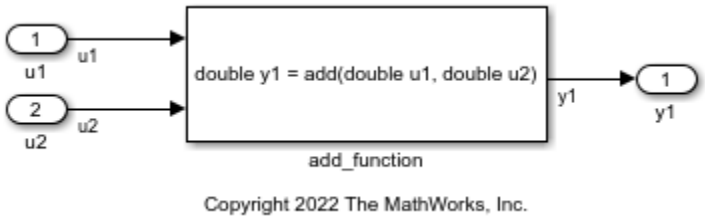
```
### Start Compiling add_function
    mex('-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\32\tp9bac2ac6\ecoder-ex61094511', '-c', '-outdi
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
    mex('add_function.c', '-IC:\TEMP\Bdoc23a_2213998_3568\ib570499\32\tp9bac2ac6\ecoder-ex610945
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling add_function
### Exit
```



The output of this script produces:

- A new model containing the S-function block.
- A TLC file named `add_function.tlc`.
- A C source file named `add_function.c`.
- A mexw32 dll file named `add_function.mexw32`.

4. Add inport blocks and an outport block and make the connections, as shown in the model.



5. Name and save your model. In this example, the model is named `ex_function_call_lct`.

6. To build the model and generate code, press **Ctrl+B**.

This code is generated in `ex_function_call_lct.c`:

```
/* Exported block signals */
real_T u1;                              /* '<Root>/u1' */
real_T u2;                              /* '<Root>/u2' */

/* External outputs (root outports fed by signals with default storage) */
ExternalOutputs rtY;

/* Model step function */
void ex_function_call_lct_step(void)
{
  /* Outport: '<Root>/y1' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  S-Function (add_function): '<Root>/add_function'
   */
  rtY.y1 = add(u1, u2);
}
```

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

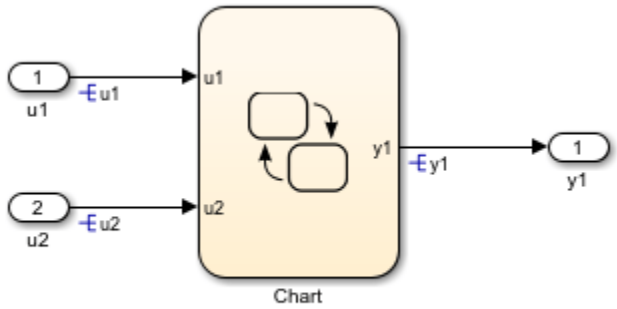
```
#include "add.h"
```

Call C Functions by Using a Stateflow Chart

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

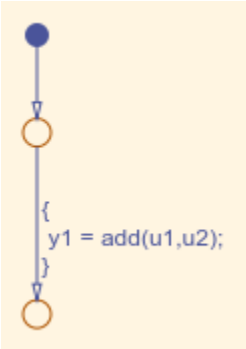
2. Create a C source file named `add.c` that contains the function body. Refer to the preceding example.

3. Add inport blocks and an outport block and make the connections, as shown in the model. In this example, the model is named `ex_function_call_SF`.



Copyright 2022 The MathWorks, Inc.

4. Double-click the Stateflow® chart and edit the chart as shown. Place the call to the `add` function within a transition action.



5. Select the **Configuration Parameters** > **Code Generation** > **Custom Code** pane. On the **Code information** tab, select **Include headers**. In the **Include headers** field, enter the `#include` statement:

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

6. Select **Source files**. In the **Source files** field, enter:

```
add.c
```

7. Select the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Use the same custom code settings as Simulation Target** parameter.

8. To build the model and generate code, press **Ctrl+B**.

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

```
/* Definition for custom storage class: Global */
real_T u1;                              /* '<Root>/u1' */
real_T u2;                              /* '<Root>/u2' */
real_T y1;                              /* '<Root>/Chart' */

/* Model step function */
void ex_function_call_SF_step(void)
{
  /* Chart: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  y1 = (real_T)add(u1, u2);
}
```

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

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

Use MATLAB Function Block to Call C Functions

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

2. Create a C source file named `add.c` that contains the function body. Refer to the preceding example.

3. In the Simulink Library Browser, click **Simulink > User Defined Functions**. Drag a MATLAB Function block into your model.

4. Double-click the MATLAB Function block. Edit the function to include:

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

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

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

end
```

5. Select the **Configuration Parameters > Code Generation > Custom Code** pane. On the **Code information** tab, select **Include headers**. In the **Include headers** field, enter the `#include` statement:

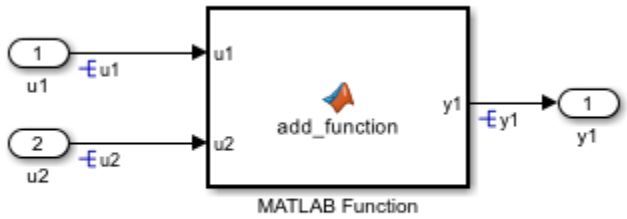
```
#include "add.h"
```

6. In the **Code information** tab, select **Source files**. In the **Source files** field, enter:

```
add.c
```

7. Add two Inport blocks and one Outport block to the model and connect it to the MATLAB Function block.

8. Save the model as `ex_function_call_ML`.



Copyright 2022 The MathWorks, Inc.

9. To build the model and generate code, press **Ctrl+B**.

`ex_exfunction_call_ML.c` contains the following code in the step function:

```
/* Definition for custom storage class: Global */
real_T u1;                              /* '<Root>/u1' */
real_T u2;                              /* '<Root>/u2' */
real_T y1;                              /* '<Root>/MATLAB Function' */

/* Model step function */
```

```
void ex_function_call_ML_step(void)
{
  /* MATLAB Function: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  y1 = add(u1, u2);
}
```

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

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

See Also

Related Examples

- “Call C Library Functions in C Charts” (Stateflow)
- “When to Generate Code from MATLAB Algorithms”
- “Legacy Code Tool and Code Generation”

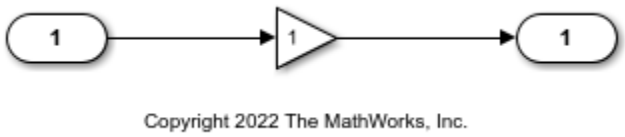
Macro Definitions (#define)

C Construct

```
#define myParam 9.8;
```

Export Generated Macro Definition

1. Open example model `ex_param_macro`.



2. In the model, select the Gain block. In the Property Inspector, set the value of the **Gain** parameter to `myParam`.

3. Next to the parameter value, click the action button (the button with three vertical dots) and select **Create**.

4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(9.8)`. Click **Create**. A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the Gain parameter, in this case, `9.8`.

5. On the **Code Generation** tab, click the **Configure in Coder App** button. In the Code Mappings editor, set **Storage Class** to `Define`. Click **OK**.

6. To build the model and generate code, press **Ctrl+B** .

The generated header file `ex_param_macro.h` defines `myParam` as a macro.

```
/* Definition for custom storage class: Define */
#define myParam                        9.8                        /* Referenced by: '<Root>/Gain'
```

Reuse Macro from Handwritten Code

1. In the Code Mappings editor, on the **Parameters** tab, click the **Update Code Mappings** button.

2. Change **Storage Class** of `myParam` from `Define` to `ImportedDefine`.

3. In the Property Inspector, under **Code** section, set **Header File** to `external_params.h`. The generated code imports the macro definition from a custom header file named `external_params.h`.

4. In your current folder, create the C header file `external_params.h`, which contains the `#define` statement.

```
#ifndef _EXTERNAL_PARAMS
#define _EXTERNAL_PARAMS
```

```
#define myParam 9.8

#endif

/* EOF */
```

5. To build the model and generate code, press **Ctrl+B** .

The generated header file `ex_param_macro.h` does not define the macro. Instead, the file uses `#include` to include the custom header file `external_params.h`.

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

The source file `ex_param_macro.c` contains a guard to check that a definition for `myParam` exists.

```
#include "rtwtypes.h"
#include "external_params.h"

/*
 * Check that imported macros with storage class "ImportedDefine" are defined
 */
#ifndef myParam
#error The value of parameter "myParam" is not defined
#endif
```

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75

Conditional Inclusions (#if / #endif)

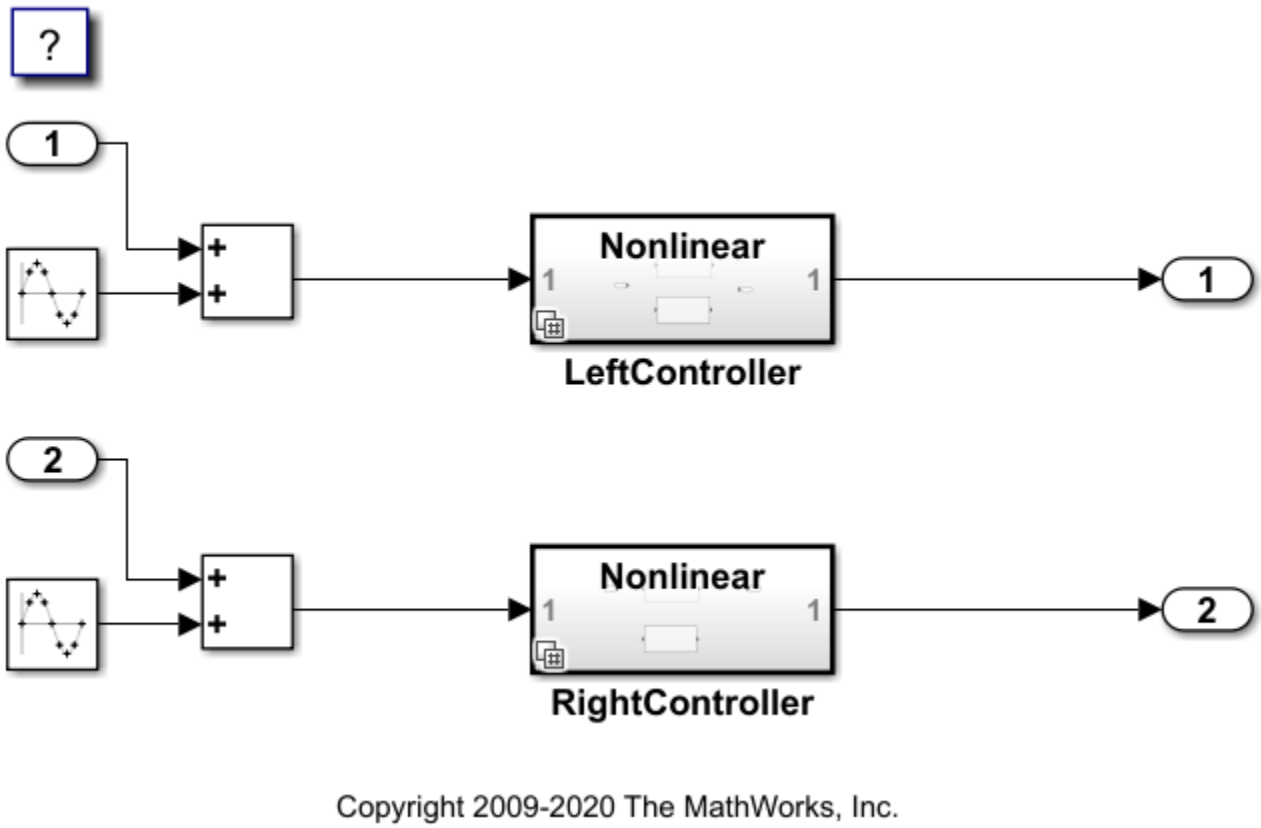
You can generate preprocessor conditional directives in your code by implementing variants in your model. In the generated code, the preprocessor conditional directives select a section of code to execute at compile time.

This example uses `Simulink.Variant` objects to control Variant Subsystem blocks. For information about other ways to control variants, see “Variant Control Modes in Variant Blocks”. For information about types of variants other than Variant Subsystem blocks, see “What Are Variants and When to Use Them”.

Inspect Model

Open the example model `rtwdemo_preprocessor_subsys`.

```
model = 'rtwdemo_preprocessor_subsys';
open_system(model);
```



This model contains two variant subsystems, `LeftController` and `RightController`, each with two child subsystems, `Linear` and `Nonlinear`. The `PostLoadFcn` callback function of the model loads `rtwdemo_preprocessor_data.mat`. To view the callback function, from the Simulink toolstrip, click the **Modeling** tab and, in the **Setup** section, click **Model Settings > Model Properties**. In the Model Properties dialog box, open the **Callbacks** tab and select **PostLoadFcn**. Alternatively, view the callback function programmatically.

```
get_param(model,'PostLoadFcn')
```

```
ans =
'load rtwdemo_preprocessor_data.mat'
```

The callback function instantiates two `Simulink.Variant` objects, `LINEAR` and `NONLINEAR`, and one `Simulink.Parameter`, `VSSMODE`. The `LINEAR` variant is active when `VSSMODE` is equal to `0`. The `NONLINEAR` variant is active when `VSSMODE` is equal to `1`. `VSSMODE` is initialized to `1`. Verify these settings programmatically.

```
LINEAR.Condition
```

```
ans =
'VSSMODE==0'
```

```
NONLINEAR.Condition
```

```
ans =
'VSSMODE==1'
```

```
VSSMODE.Value
```

```
ans = int32
    1
```

`LeftController` and `RightController` use `LINEAR` and `NONLINEAR` as the variant control expressions for their respective `Linear` and `Nonlinear` subsystems. To view this configuration, right-click `LeftController` and click **Block Parameters (Subsystem)**. Alternatively, view the variant control expressions programmatically.

```
choices = get_param(append(model,'/LeftController'),'VariantChoices');
choices(1)
```

```
ans = struct with fields:
         Name: 'LINEAR'
    BlockName: 'rtwdemo_preprocessor_subsys/LeftController/Linear'
```

```
choices(2)
```

```
ans = struct with fields:
         Name: 'NONLINEAR'
    BlockName: 'rtwdemo_preprocessor_subsys/LeftController/Nonlinear'
```

Configure Model

Open the Configuration Parameters dialog box. In the left pane, click **Code Generation**, then confirm **System target file** is set to `ert.tlc`. Alternatively, set the parameter programmatically.

```
set_param(model,'SystemTargetFile','ert.tlc');
```

In the left pane, click **Interface.** Confirm the Ignore custom storage classes parameter is cleared. Alternatively, clear the parameter programmatically.

```
set_param(model,'IgnoreCustomStorageClasses','off');
```

To generate preprocessor conditionals, use custom storage classes.

Open the Block Parameters dialog box for the `LeftController` subsystem. Verify the “Variant activation time” parameter is set to `code compile`. Alternatively, set the parameter programmatically.

```
set_param(append(model,'/LeftController'),'VariantActivationTime','code compile');
```

Open the `LeftController` subsystem. For each child subsystem, click the subsystem and, in the **Subsystem Block** tab, confirm **Atomic Subsystem** is selected. Alternatively, set the parameters programmatically.

```
set_param(append(model,'/LeftController/Linear'),'TreatAsAtomicUnit','on');
set_param(append(model,'/LeftController/Nonlinear'),'TreatAsAtomicUnit','on');
```

Generate code.

```
[~,cgDir] = rtwdemodir();
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_preprocessor_subsys
### Successful completion of build procedure for: rtwdemo_preprocessor_subsys

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_preprocessor_subsys  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.432s
```

View the generated code. The generated code includes `#if` and `#endif` preprocessor directives that use `LINEAR` and `NONLINEAR` to select one variant to compile.

```
cfile = fullfile(cgDir,'rtwdemo_preprocessor_subsys_ert_rtw','rtwdemo_preprocessor_subsys.c');
rtwdemodbtype(cfile,"  /* Outputs for Atomic SubSystem: '<Root>/LeftController' */", "  /* End o
```

```
  /* Outputs for Atomic SubSystem: '<Root>/LeftController' */
#if LINEAR

  /* Outputs for Atomic SubSystem: '<S1>/Linear' */
  /* VariantMerge generated from: '<S1>/Out1' incorporates:
   *  DiscreteFilter: '<S3>/Discrete Filter'
   */
  rtY.Out1 = rtb_Add - 0.5 * rtDWork.DiscreteFilter_states_a;

  /* Update for DiscreteFilter: '<S3>/Discrete Filter' */
  rtDWork.DiscreteFilter_states_a = rtY.Out1;

  /* End of Outputs for SubSystem: '<S1>/Linear' */
#elif NONLINEAR

  /* Outputs for Atomic SubSystem: '<S1>/Nonlinear' */
  /* VariantMerge generated from: '<S1>/Out1' incorporates:
   *  DiscreteFilter: '<S4>/Discrete Filter'
   *  Lookup_n-D: '<S4>/Lookup Table'
   *  Sum: '<Root>/Add'
```

```
   */
  rtY.Out1 = look1_binlxpw(rtb_Add, rtCP_LookupTable_bp01Data,
    rtCP_LookupTable_tableData, 4U) - 0.5 * rtDWork.DiscreteFilter_states_o;

  /* Update for DiscreteFilter: '<S4>/Discrete Filter' */
  rtDWork.DiscreteFilter_states_o = rtY.Out1;

  /* End of Outputs for SubSystem: '<S1>/Nonlinear' */
#endif
```

See Also

Related Examples

- “Create a Simple Variant Model”
- “Generate Preprocessor Conditionals for Variant Systems” on page 27-32

Typedef

Create data type aliases by generating `typedef` statements.

C Construct

```
typedef float float_32;
```

Procedure

To create a data type alias in Simulink®, use a `Simulink.AliasType`. The code generator creates a `typedef` statement in the generated code.

The built-in Simulink data type `single` corresponds to the C data type `float`.

1. At the command prompt, create a `Simulink.AliasType` object named `float_32` that represents an alias of `single`.

```
float_32 = Simulink.AliasType('single');
```

2. Open the example model `ex_typedef`.

3. In the Model Data Editor, open the **Inports/Outports** tab.

4. From the **Change View** menu, select `Design`.

5. In the model, select the Inport block.

6. In the Model Data Editor, for the Inport block, set **Data Type** to `float_32`. Click `Refresh data type` if `float_32` is not available in the drop down options.

7. Open the **Embedded Coder** app.

8. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

9. In the Code Mappings editor, on the **Inports** tab, set **Storage Class** for the Inport block to `ExportedGlobal`. The Inport block appears in the generated code as a separate global variable.

10. In the Property Inspector, for the Inport block, set the **Code** > **Identifier** property to `mySig`.

11. To build the model and generate code, press **Ctrl+B** .

Results

The generated header file `ex_typedef.h` defines the data type alias `float_32`.

```
#ifndef DEFINED_TYPEDEF_FOR_float_32_
#define DEFINED_TYPEDEF_FOR_float_32_
```

```
typedef real32_T float_32;

#endif
```

By default, the code generator also creates the alias `real32_T`, which corresponds to the C data type `float`. You can see the `typedef` statement in the generated header file `rtwtypes.h`.

```
typedef float real32_T;
```

The generated source file `ex_typedef.c` uses `float_32` to define the global variable mySig.

```
/* Exported block signals */
float_32 mySig;                         /* '<Root>/In1' */
```

See Also

`Simulink.AliasType`

Related Examples

- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “Structures of Signals” on page 26-67

Structures of Parameters

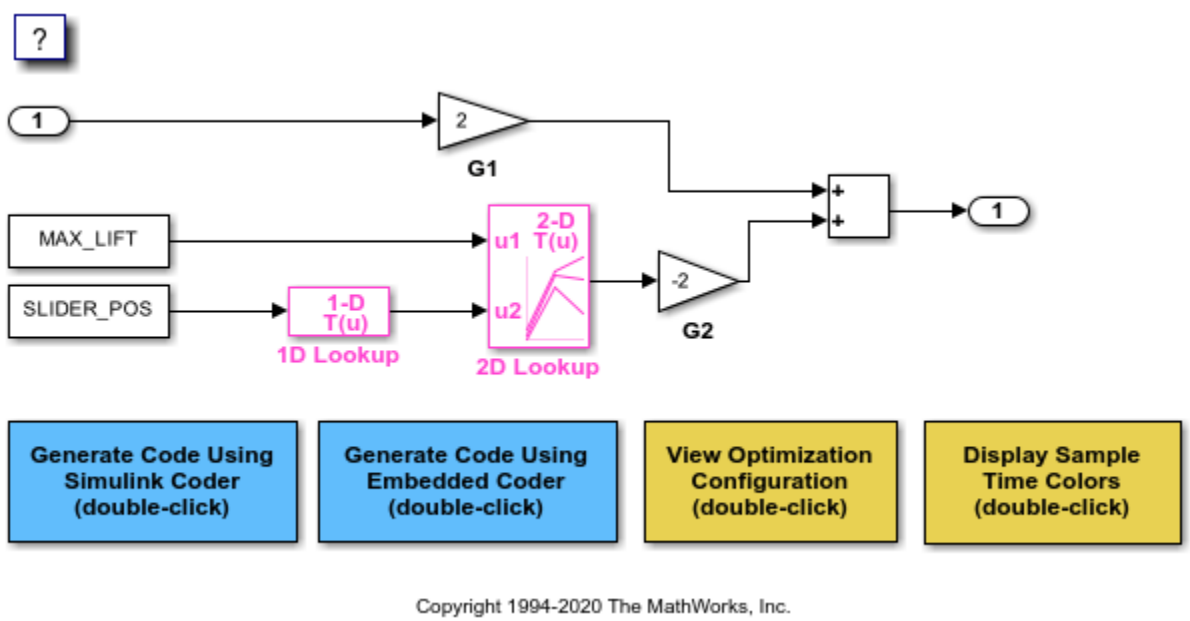
Create a structure in the generated code. The structure stores parameter data.

C Construct

```
typedef struct {
  double G1;
  double G2;
} myStructType;

myStructType myStruct = {
  2.0,
  -2.0
} ;
```

Procedure

1. Open the example model `rtwdemo_paraminline`.



2. On the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, select the **Parameters** tab.

3. In the model, click the Gain block labeled `G1`. In the Model Data Editor, use the **Value** column to set the value of the **Gain** parameter to `myStruct.G1`.

4. Set the value of the **Gain** parameter in the `G2` block to `myStruct.G2`.

5. Next to `myStruct.G2`, click the action button (with three vertical dots) and select **Create**.

6. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(struct)` and click **Create**. A `Simulink.Parameter` object named `myStruct` appears in the base workspace.

7. In the Simulink.Parameter property dialog box, next to the **Value** property, click the action button and select **Open Variable Editor**.

8. Right-click the white space under the **Field** column and select **New**. Name the new structure field `G1`. Use the **Value** column to set the value of the field to `2`.

9. Add a field `G2` whose value is `-2`, and then close the Variable Editor.

10. In the Simulink.Parameter property dialog box, on the **Code Generation** tab, set **Storage class** to `ExportedGlobal`. The structure `myStruct` appears in the generated code as a global variable.

11. Generate code from the model.

Results

The generated header file `rtwdemo_paraminline_types.h` defines a structure type that has a random name.

```
typedef struct {
  real_T G1;
  real_T G2;
} struct_6h72eH5WFuEIyQr5YrdGuB;
```

The source file `rtwdemo_paraminline.c` defines and initializes the structure variable `myStruct`.

```
/* Exported block parameters */
struct_6h72eH5WFuEIyQr5YrdGuB myStruct = {
  2.0,
  -2.0
} ;                                     /* Variable: myStruct
                                         * Referenced by:
                                         *   '<Root>/G1'
                                         *   '<Root>/G2'
                                         */
```

Specify Name of Structure Type

1. Optionally, specify a name to use for the structure type definition (`struct`). At the command prompt, use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object that represents the structure type.

2. The default name of the object is `slBus1`. Change the name by copying the object into a new MATLAB® variable.

3. In the Model Data Editor, click the **Show/refresh additional information** button.

4. In the data table, find the row that corresponds to `myStruct`. Use the **Data Type** column to set the data type of `myStruct` to `Bus: myStructType`.

5. Generate code from the model.

The code generates the definition of the structure type `myStructType` and uses this type to define the global variable `myStruct`.

```
myStructType myStruct = {
  2.0,
  -2.0
} ;                                     /* Variable: myStruct
```

See Also

Related Examples

- “Organize Data into Structures in Generated Code” on page 36-147
- “Configure Parameters for C Code Generation” on page 33-61
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Structures of Signals

This example shows how to create a structure of signal data in the generated code.

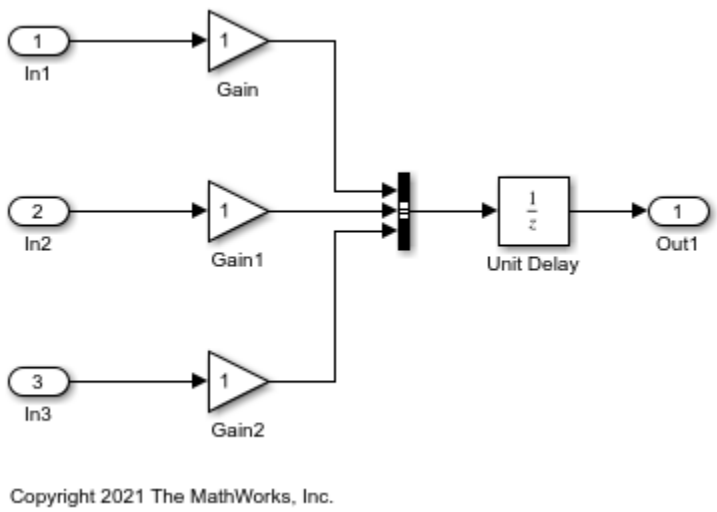
C Construct

```
typedef struct {
   double signal1;
   double signal2;
   double signal3;
} my_signals_type;
```

Procedure

To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of buses in your model.

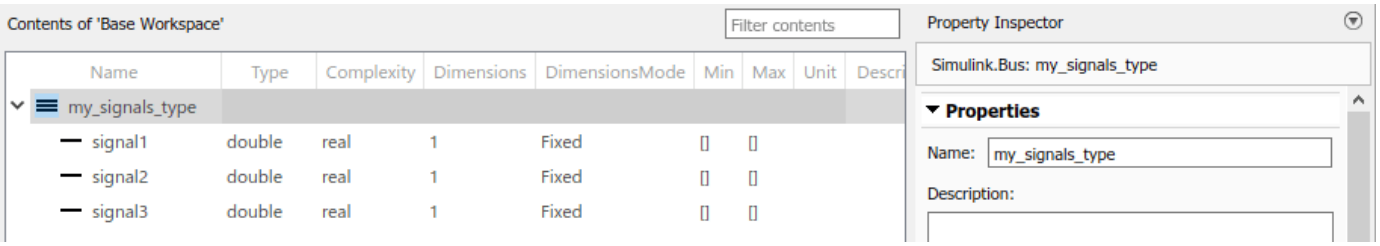
1. Create the `ex_signal_struct` model by using Gain blocks, a Bus Creator block, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



2. To configure the Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

3. In the toolstrip, on the **Modeling** tab, under **Design**, click **Type Editor**.

4. In the Type Editor toolstrip, in the **Add** gallery, click **Bus** to create a `Simulink.Bus` object. Then, name it `my_signals_type`. With `my_signals_type` selected, in the **Add** gallery, click **Bus Element** three times to add three elements. Name them `signal1`, `signal2`, and `signal3`. For more information, see Type Editor.



5. Save the bus objects in your current folder as `ex_signal_struct_data.mat`.

This bus object represents the structure type that you want the generated code to use.

6. In the Bus Creator block dialog box, set **Output data type** to `Bus: my_signals_type`.

7. Select **Output as nonvirtual bus**. Click **OK**. A nonvirtual bus appears in the generated code as a structure.

8. Open the **Simulink Coder** app. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

9. Open the **Signals/States** tab. In the model, select the output signal of the Bus Creator block and click the **Add selected signals to code mappings** button in the Code Mappings editor.

11. For the added signal, set **Storage Class** to `ExportedGlobal`.

12. In the Property Inspector, set the **Code** > **Identifier** property to `sig_struct_var`. The output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

13. Generate code from the model.

Results

The generated header file `ex_signal_struct_types.h` defines the structure type `my_signals_type`.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} my_signals_type;
```

The source file `ex_signal_struct.c` allocates memory for the global variable `sig_struct_var`, which represents the output of the Bus Creator block.

```
/* Exported block signals */
my_signals_type sig_struct_var;        /* '<Root>/Bus Creator' */
```

In the same file, in the model `step` function, the algorithm accesses `sig_struct_var` and the fields of `sig_struct_var`.

See Also

`Simulink.Bus`

Related Examples

- “Organize Data into Structures in Generated Code” on page 36-147
- “Group Nonvirtual Buses in Arrays of Buses”
- “Configure Signal Data for C Code Generation” on page 33-50
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Nested Structures of Signals

You can create nested structures of signal data in the generated code.

C Construct

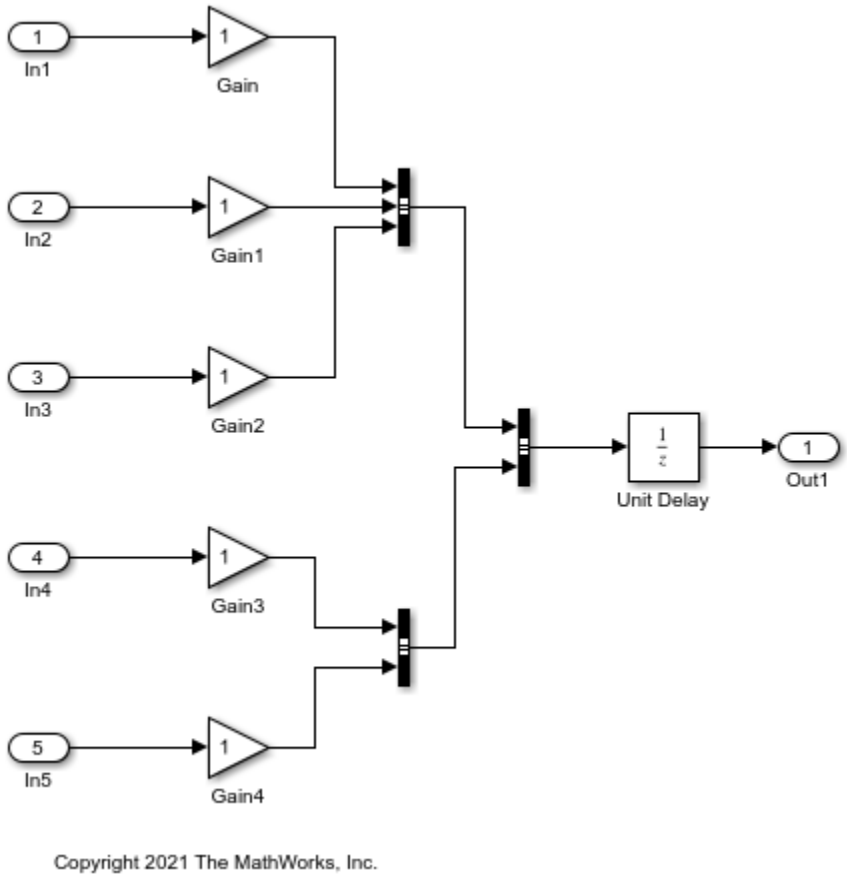
```
typedef struct {
  double signal1;
  double signal2;
  double signal3;
} B_struct_type;

typedef struct {
  double signal1;
  double signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
  C_struct_type subStruct_C;
} A_struct_type;
```

Procedure

To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of buses in your model.

To nest a structure inside another structure, use a bus object as the data type of an element in another bus object.

1. Create the `ex_signal_nested_struct` model with Gain blocks, Bus Creator blocks, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



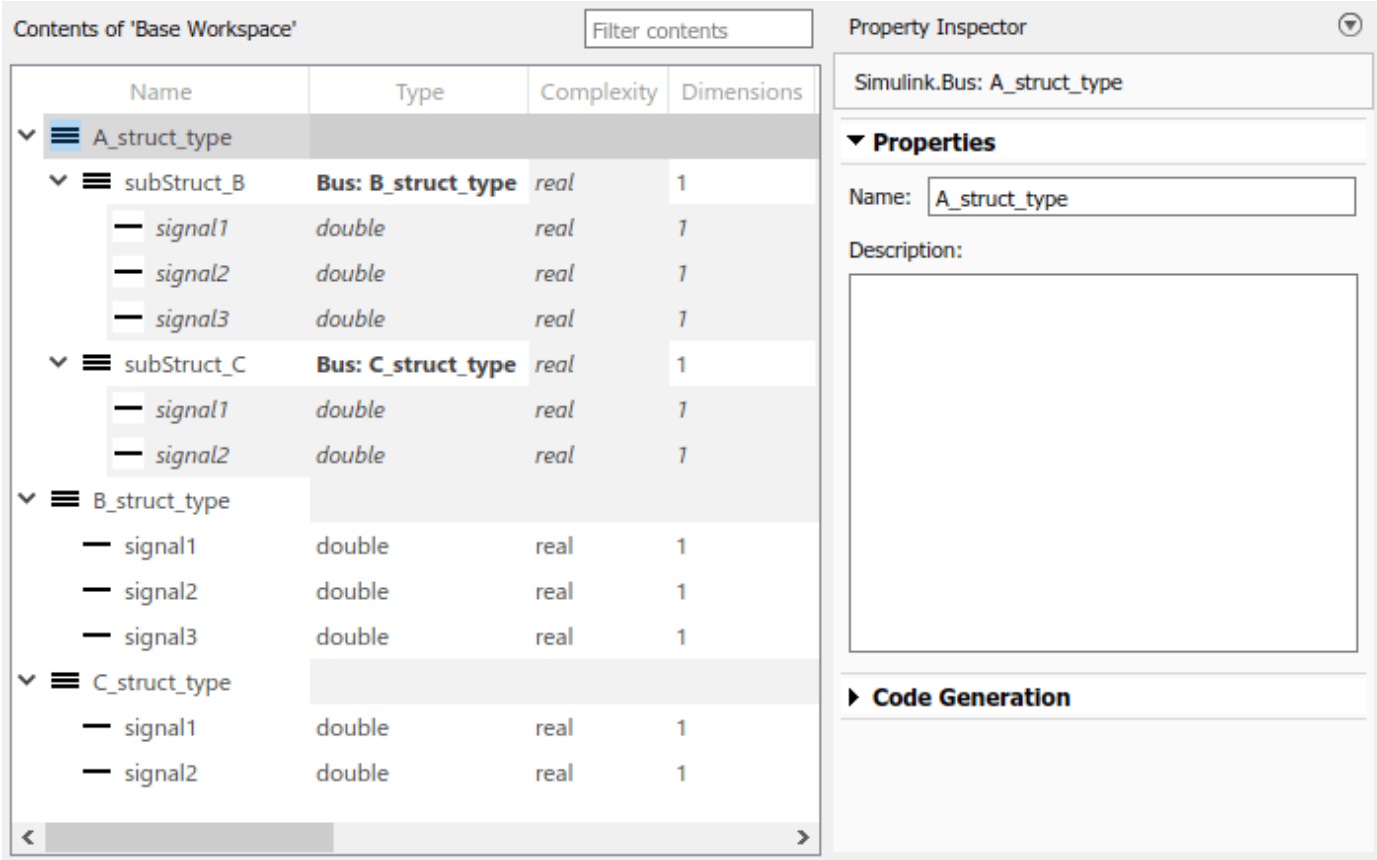
2. To configure a Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

3. In the toolstrip, on the **Modeling** tab, under **Design**, click **Type Editor**.

In the Type Editor toolstrip, in the **Add** gallery, click **Bus** to create a `Simulink.Bus` object. Then, name it `A_struct_type`. With `A_struct_type` selected, in the **Add** gallery, click **Bus Element** twice to add two elements. Name the elements `subStruct_B` and `subStruct_C`. For more information, see Type Editor. This bus object represents the top-level structure type that you want the generated code to use.

4. Similarly, create `Simulink.Bus` objects `B_struct_type` (with three elements) and `C_struct_type` (with two elements).

5. In the `A_struct_type` object, for the `subStruct_B` element, set **DataType** to `Bus: B_struct_type`. Similarly, for the `subStruct_C` element, set **DataType** to `Bus: C_struct_type`.



Each element in `A_struct_type` uses another bus object as a data type. These elements represent substructures.

6. In the dialog box of the Bus Creator block that collects the three Gain signals, set **Output data type** to `Bus: B_struct_type`. Click **Apply**.

7. Select **Output as nonvirtual bus** and click **OK**.

8. In the dialog box of the other subordinate Bus Creator block, set **Output data type** to `Bus: C_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

9. In the last Bus Creator block dialog box, set **Output data type** to `Bus: A_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

10. Open the **Simulink Coder** app. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

11. Open the **Signals/States** tab. In the model, select the output signal of the `A_struct_type` Bus Creator block, which feeds the Unit Delay block. Click the **Add selected signals to code mappings** button in the Code Mappings editor.

12. For the added signal, set **Storage Class** to `ExportedGlobal`.

13. In the Property Inspector, set the **Code** > **Identifier** property to `sig_struct_var`. The output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

14. Generate code from the model.

Results

The generated header file `ex_signal_nested_struct_types.h` defines the structure types. Each structure type corresponds to a `Simulink.Bus` object.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} B_struct_type;

typedef struct {
  real_T signal1;
  real_T signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
  C_struct_type subStruct_C;
} A_struct_type;
```

The generated source file `ex_signal_nested_struct.c` allocates memory for the global structure variable `sig_struct_var`. By default, the name of the `A_struct_type` Bus Creator block is `Bus Creator2`.

```
/* Exported block signals */
A_struct_type sig_struct_var;          /* '<Root>/Bus Creator2' */
```

In the same file, in the model `step` function, the algorithm accesses `sig_struct_var` and the fields of `sig_struct_var`.

See Also

`Simulink.Bus`

Related Examples

- “Organize Data into Structures in Generated Code” on page 36-147
- “Group Nonvirtual Buses in Arrays of Buses”
- “Configure Signal Data for C Code Generation” on page 33-50
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

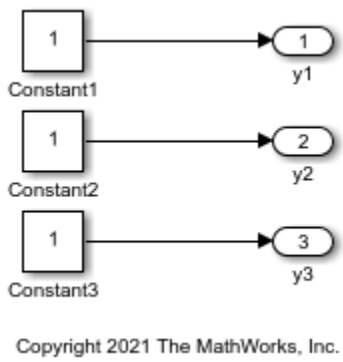
Bitfields

C Construct

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

Procedure

1. Open example model `ex_struct_bitfield_CSC`. The model contains three Constant blocks and three Outport blocks.



2. Open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**.

3. In the Model Data Editor, on the **Parameters** tab, in the **Value** column, observe that the value of the first Constant block is `p1`. Next to the parameter value, click the action button (the button with three vertical dots) and select **Create**.

4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(false)`. Click **Create**. A `Simulink.Parameter` object, `p1`, appears in the base workspace. The object stores a Boolean value, `false`, and uses the data type `boolean`.

5. In the `Simulink.Parameter` property dialog box, click **OK**.

6. Use the Model Data Editor to configure the other Constant blocks to refer to new parameter objects named `p2` and `p3`.

7. Open the **Embedded Coder** app.

8. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

9. In the Code Mappings editor, on the **Parameters** tab, click the **Update Code Mappings** button.

10. Use the **Storage Class** column to apply the storage class `BitField` to all parameter objects.

10. In the Property Inspector, use the **Struct Name** column to configure each object to use the same structure type, `my_struct`.

11. Generate code from the model.

Results

The generated header file `ex_struct_bitfield_CSC.h` defines the structure type `my_struct_type`.

```
/* Type definition for custom storage class: BitField */
typedef struct myStruct_tag {
  uint_T p1 : 1;                       /* Referenced by: '<Root>/Constant1' */
  uint_T p2 : 1;                       /* Referenced by: '<Root>/Constant2' */
  uint_T p3 : 1;                       /* Referenced by: '<Root>/Constant3' */
} myStruct_type;
```

The generated source file `ex_struct_bitfield_CSC.c` defines and initializes the structure variable `my_struct`.

```
/* Definition for custom storage class: BitField */
myStruct_type myStruct = {
  /* p1 */
  false,

  /* p2 */
  false,

  /* p3 */
  false
};
```

See Also

Related Examples

- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- Pack Boolean data into bitfields

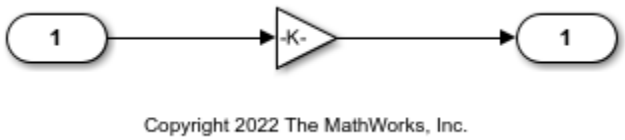
Arrays for Parameters

C Construct

```
float myParams[5]= {1.0F,2.0F,3.0F,4.0F,5.0F};
```

Procedure

1. Create the `ex_param_array` model by using a Gain block.



2. In the model, select the Gain block. In the Property Inspector, set the value of the **Gain** parameter to `myParam`.

3. Next to the parameter value, click the action button (the button with three vertical dots) and select **Create**.

4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter ([1 2 3 4 5])`. Click **Create**. A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the Gain parameter.

5. In the `Simulink.Parameter` property dialog box, set **Storage class** to `ExportedGlobal`. Click OK.

With this setting, `myParams` appears in the generated code as a separate global variable.

6. Set **Data type** to `single`. Click **OK**.

7. To build the model and generate code, press **Ctrl+B** .

Results

The generated source file `ex_param_array.c` defines and initializes the global variable `myParams`.

```
/* Exported block parameters */
real32_T myParam[5] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F } ;/* Variable: myParam
                                                         * Referenced by: '<Root>/Gain'
                                                         */
```

See Also

Related Examples

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Configure Parameters for C Code Generation” on page 33-61

- “Code Generation of Matrices and Arrays” on page 52-63

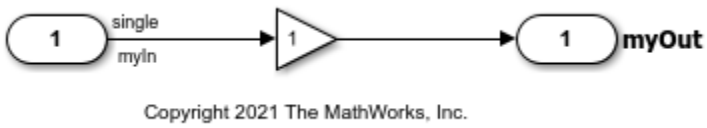
Arrays for Signals

C Construct

```
double myIn[5];
double myOut[5];
```

Procedure

1. Open example model `ex_signal_array`.



2. In the Model Data Editor, select the **Inports/Outports** tab.

3. From the **Change View** drop-down list, select `Design`.

Observe these settings:

- For the Inport block, the **Signal Name** is `myIn` and Dimensions is `[5 1]`.
- For the Outport block, the **Signal Name** is `myOut`.

4. Open the **Embedded Coder** app.

5. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

6. In the Code Mappings editor, on the **Inports** tab, for the Inport block and the Outport block, the **Storage Class** is `ExportedGlobal`. With this setting, the blocks appear in the generated code as separate global variables.

7. To build the model and generate code, press **Ctrl+B** .

Results

The generated source file `ex_signal_array.c` defines the global variables `myIn` and `myOut` as arrays with 5 elements each.

```
/* Exported block signals */
real32_T myIn[5];                       /* '<Root>/In1' */
real32_T myOut[5];                      /* '<Root>/Out1' */
```

See Also

Related Examples

- “Determine Signal Dimensions”

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Configure Signal Data for C Code Generation” on page 33-50
- “Code Generation of Matrices and Arrays” on page 52-63

Pointers

When your handwritten code allocates memory for signal, state, or parameter data, you can generate code that accesses that data through a pointer. Apply a storage class such as `ImportedExternPointer` to a data item in the model. Your handwritten code provides the pointer definition.

C Construct

```
extern double *myIn;
```

Procedure

1. Open example model `ex_pointer`.

2. Open the **Embedded Coder** app.

3. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

4. In the Code Mappings editor, on the **Inports** tab, set **Storage Class** for the Inport block to `ImportedExternPointer`.

5. Generate code from the model.

Results

The generated header file `ex_pointer.h` declares the pointer.

```
/* Imported (extern) pointer block signals */
extern real_T *In1;                     /* '<Root>/In1' */
```

In the generated source file `ex_pointer.c`, in the model step function, the algorithm dereferences the pointer, `In1`.

```
/* Model step function */
void ex_pointer_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
```

```
  rtY.Out1 = *In1;
}
```

See Also

Related Examples

- “C Code Generation Configuration for Model Interface Elements”
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75

Variant Systems in Embedded Coder

- “Implement Symbolic Dimensions for Array Sizes in Generated Code” on page 27-2
- “Code Generation for Variant Blocks” on page 27-15
- “Represent Subsystem and Variant Models in Generated Code” on page 27-22
- “Generate Preprocessor Conditionals for Variant Systems” on page 27-32
- “Represent Variant Source and Sink Blocks in Generated Code” on page 27-37
- “Configure Symbolic Dimensions for S-Function Blocks” on page 27-45
- “Options to Represent Variant Parameters in Generated Code” on page 27-53
- “Generate Code for Variant Subsystem with Child Subsystems of Different Output Signal Dimensions” on page 27-62
- “Use Variant Subsystem to Generate Code That Uses C Preprocessor Conditionals” on page 27-65
- “Use Variant Models to Generate Code That Uses C Preprocessor Conditionals” on page 27-70
- “Compile Code Conditionally for All Values of Variant Parameters with Same and Different Dimensions” on page 27-76

Implement Symbolic Dimensions for Array Sizes in Generated Code

Dimension Variants

Use symbolic dimensions to simulate various sets of dimension choices without regenerating code for every set. Set up your model with dimensions that you specify as symbols in blocks and data objects. These symbols propagate throughout the model during simulation, and then go into the generated code. Modeling constraints for symbols during simulation (for example, `C=A+B`) are output as preprocessor conditionals in either the `model.h` or the *`model`* `_types.h` file.

You can directly specify dimension information as a symbol or a numeric constant for these blocks and data objects:

- Inport
- Outport
- Signal Specification
- Data Store Memory
- Interpreted MATLAB® Function
- `Simulink.Signal`
- `Simulink.Parameter`
- `Simulink.BusElement`
- `AUTOSAR.Parameter`

The Data Store Memory and Interpreted MATLAB Function blocks also support variable dimension signals. For these blocks, the symbolic dimensions control the maximum allowed size.

You must provide the initialization code for `Simulink.Parameter` objects that contain symbolic dimensions. To prevent the generated code from initializing these parameters, you must either:

- Configure the parameters to use a storage class with the **Data scope** property set to `Imported`, such as the `ImportedExtern` or `ImportedExternPointer` built-in storage classes.
- Configure the parameters to use a storage class with the **Data initialization** property set to `None`.

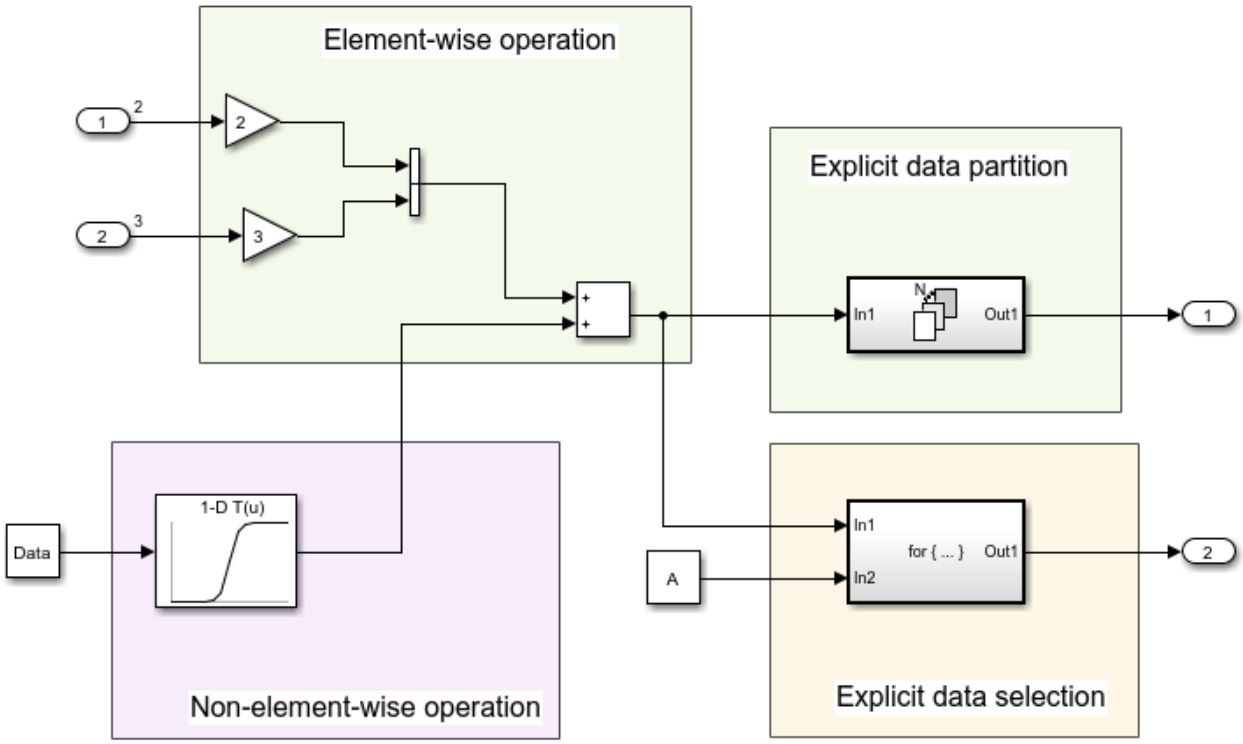
You use `Simulink.Parameter` objects to specify dimension information as symbols. For more information on signal dimensions, see “Determine Signal Dimensions”.

You must fix symbolic dimension values at compile-time. You can not resolve a symbolic dimension to another variable and then vary it during run-time because you define the behavior during simulation.

Note: The dimension variants feature is on by default. You can turn off this feature by clearing the “Allow symbolic dimension specification” parameter in the Configuration Parameters dialog box.

Define Symbolic Dimensions

This example uses the model `rtwdemo_dimension_variants` to show how to implement symbolic dimensions. This model has four modeling patterns involving vectors and matrices.



1. To show block names, on the **Debug** tab, on the **Information Overlays** menu, clear the **Hide Automatic Block Names** parameter.
2. Open the **Embedded Coder** app. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.
3. In the Code Mappings editor, on the **Parameters** tab, click the refresh button. Seven `Simulink.Parameter` objects appear. Four of these objects are for specifying symbolic dimensions. These `Simulink.Parameter` objects have the names `A`, `B`, `C`, and `D`. Notice that these parameters have a storage class of `CompilerFlag`.

To use a `Simulink.Parameter` object for dimension specification, it must be defined in the base workspace and have one of these storage classes:

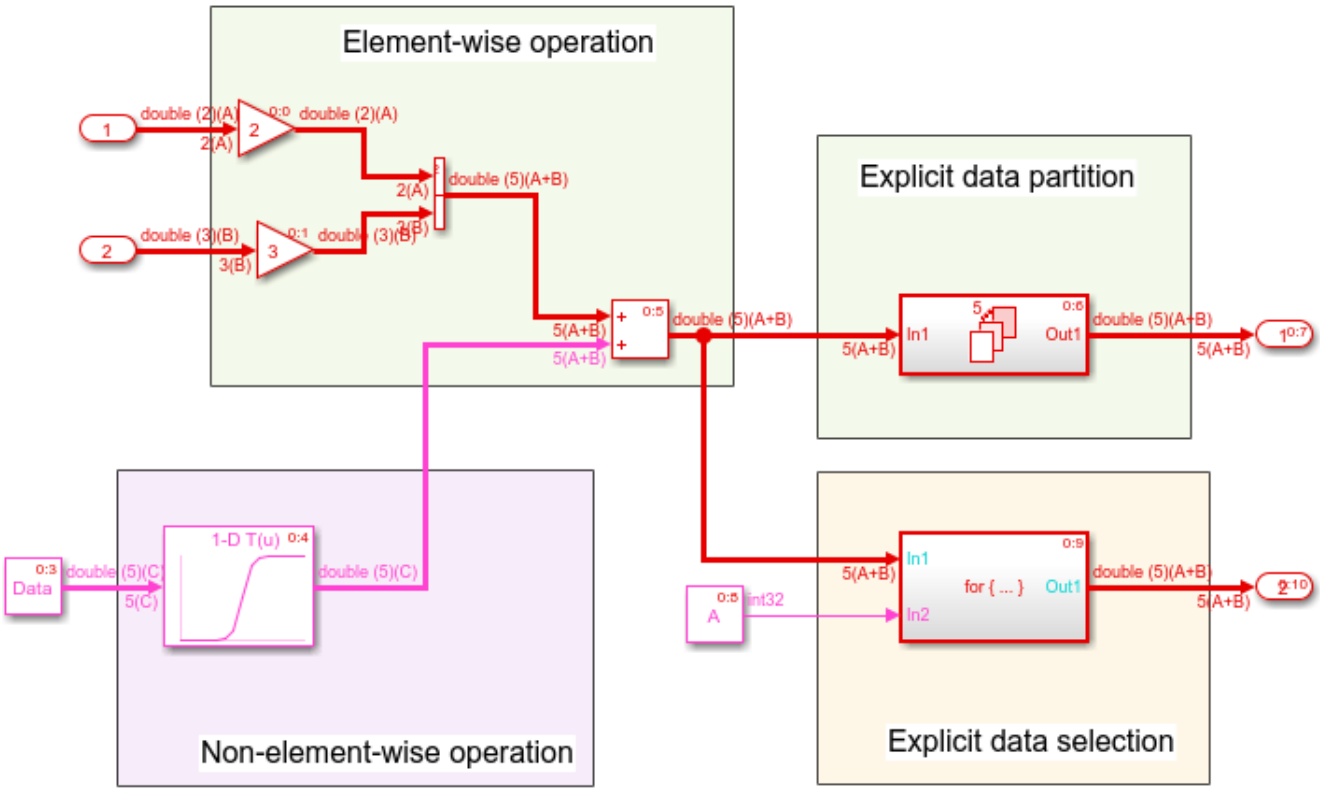
- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- User-defined custom storage class that defines data as a macro in a specified header file

For `Simulink.Parameter` objects with an `ImportedDefine` custom storage class, provide a header file on the MATLAB path. Insert the name of the header file in the **HeaderFile** field in the Simulink.Parameter dialog box.

You can use MATLAB expressions to specify symbolic dimensions. For a list of supported MATLAB expressions, see the section Operators and Operands in Variant Condition Expressions in “Introduction to Variant Controls”.

Specify Symbolic Dimensions for Blocks and Data Objects

1. Click Inport Block `In2`. On the Property Inspector, on the **Parameters** tab, the **Port Dimensions** field contains the `Simulink.Parameter` object `A`. For Inport blocks, you specify symbolic dimensions in the **Port Dimensions** field.
2. Click Inport block `In3`. The **Port Dimensions** field contains the `Simulink.Parameter` object `B`.
3. Click the Constant block. The **Constant value** parameter has a value of `Data`. In the Code Mappings editor, on the **Parameters** tab, click the `Simulink.Parameter` object `Data`. On the Property Inspector, the **Dimension** field has the character vector `'[1,C]'`, which is equivalent to `'[1,5]'` because `C` has a value of `5`. The **Value** field contains an array with `5` values, which is consistent with its dimensions. The dimensions of the data object must be consistent with the value of the `Simulink.Parameter` object that is in the **Dimensions** field. Notice that `Data` has a **Storage class** of `ImportedExtern`.
4. Open the `1-D Lookup Table1` block parameters dialog box. The **Table data** field contains the `Simulink.Parameter`, `PT`. The **Breakpoints 1** field contains the `Simulink.Parameter`, `PB`.
5. In the Code Mappings editor, click `PB` and `PT` and view their properties. These parameters contain the character vector `'[1,D]'` in their **Dimensions** field and are arrays consisting of 15 values. The value of `D` is consistent with the dimension of the `PB` and `PT` parameters because `D` has a value of `15` .
6. Simulate the model. Simulink® propagates the dimensions symbolically in the diagram. During propagation, Simulink establishes modeling constraints among symbols. Simulink then checks for consistency with these constraints based on current numerical assignments. One modeling constraint for `rtwdemo_dimension_variants` is that `C=A+B`. The **Diagnostic Viewer** produces a warning for any violations of constraints.
7. Change the dimension specification to a different configuration and simulate the model again.



Though not shown in this example, you can specify an n-D dimension expression with one or more of the dimensions being a symbol (for example, `'[A,B,C]'` or `'[1,A,3]'`). To preserve multi-dimensional information, you must specify an n-D dimension expression with each dimension that you want to preserve being a symbol.

Generate Code for a Model with Dimension Variants

Once you have verified dimension specifications through model simulation, generate code for `rtwdemo_dimension_variants`.

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
model='rtwdemo_dimension_variants';
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_dimension_variants
### Successful completion of build procedure for: rtwdemo_dimension_variants

Build Summary

Top model targets built:
```

```
Model                        Action                        Rebuild Reason
==============================================================================================
rtwdemo_dimension_variants  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.244s
```

View the generated code. In the `rtwdemo_dimension_variants.h` file, symbolic dimensions are in data declarations.

```
hfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.h');
rtwdemodbtype(hfile,'/* External inputs', '/* Real-time', 1, 0);
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  real_T In2[A];                        /* '<Root>/In2' */
  real_T In3[B];                        /* '<Root>/In3' */
} ExtU;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1[(A + B)];                 /* '<Root>/Out1' */
  real_T Out2[(A + B)];                 /* '<Root>/Out2' */
} ExtY;
```

The `rtwdemo_dimension_variants.h` file contains data definitions and preprocessor conditionals that define constraints established among the symbols during simulation. One of these constraints is that the value of a symbolic dimension must be greater than 1. This file also includes the user-provided header file for any `Simulink.Parameter` objects with an `ImportedDefine` custom storage class.

```
hfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.h');
rtwdemodbtype(hfile,'#ifndef A', '/* Macros for accessing', 1, 0);
```

```
#ifndef A
#error The value of parameter "A" is not defined
#endif

#ifndef B
#error The value of parameter "B" is not defined
#endif

#ifndef C
#error The value of parameter "C" is not defined
#endif

#ifndef D
#error The value of parameter "D" is not defined
#endif

/*
 * Constraints for division operations in dimension variants
 */
#if (1 == 0) || (((A+B) % 1) != 0)
# error "The preprocessor definition '1' must not be equal to zero and     the division of '(A+B
#endif
```

```
/*
 * Registered constraints for dimension variants
 */
/* Constraint 'C == (A+B)' registered by:
 * '<Root>/1-D Lookup Table1'
 */
#if C != (A+B)
# error "The preprocessor definition 'C' must be equal to '(A+B)'"
#endif

#if A <= 1
# error "The preprocessor definition 'A' must be greater than '1'"
#endif

#if B <= 1
# error "The preprocessor definition 'B' must be greater than '1'"
#endif

/* Constraint 'D > 1' registered by:
 * '<Root>/1-D Lookup Table1'
 */
#if D <= 1
# error "The preprocessor definition 'D' must be greater than '1'"
#endif

/* Constraint 'C > 3' registered by:
 * '<S2>/Assignment'
 */
#if C <= 3
# error "The preprocessor definition 'C' must be greater than '3'"
#endif

#if A >= 11
# error "The preprocessor definition 'A' must be less than '11'"
#endif

#if B >= 11
# error "The preprocessor definition 'B' must be less than '11'"
#endif

/* Constraint 'D < 21' registered by:
 * '<Root>/1-D Lookup Table1'
 */
#if D >= 21
# error "The preprocessor definition 'D' must be less than '21'"
#endif

/* Constraint 'C < 11' registered by:
 * '<S2>/Assignment'
 */
#if C >= 11
# error "The preprocessor definition 'C' must be less than '11'"
#endif
```

In the `rtwdemo_dimension_variants.c` file, symbolic dimensions participate in loop bound calculations, array size and index offset calculations, and a parameterized utility function (for example, Lookup Table block) calculation.

```
cfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_dimension_variants_step(void)
{
  real_T rtb_VectorConcatenate[A + B];
  real_T rtb_VectorConcatenate_0;
  int32_T ForEach_itr;
  int32_T i;
  int32_T s2_iter;

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In2'
   */
  for (ForEach_itr = 0; ForEach_itr <= (int32_T)(A - 1); ForEach_itr++) {
    rtb_VectorConcatenate[ForEach_itr] = 2.0 * rtU.In2[ForEach_itr];
  }

  /* End of Gain: '<Root>/Gain' */

  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In3'
   */
  for (ForEach_itr = 0; ForEach_itr <= (int32_T)(B - 1); ForEach_itr++) {
    rtb_VectorConcatenate[(int32_T)(A + ForEach_itr)] = 3.0 *
      rtU.In3[ForEach_itr];
  }

  /* End of Gain: '<Root>/Gain1' */

  /* Outputs for Iterator SubSystem: '<Root>/For Each Subsystem' incorporates:
   *  ForEach: '<S1>/For Each'
   */
  for (ForEach_itr = 0; ForEach_itr <= (int32_T)((int32_T)(A + B) - 1);
       ForEach_itr++) {
    /* Sum: '<Root>/Add' incorporates:
     *  Constant: '<Root>/Constant'
     *  Lookup_n-D: '<Root>/1-D Lookup Table1'
     */
    rtb_VectorConcatenate_0 = look1_binlx(Data[ForEach_itr], PB, PT, (uint32_T)
      (int32_T)(D - 1)) + rtb_VectorConcatenate[ForEach_itr];
    rtb_VectorConcatenate[ForEach_itr] = rtb_VectorConcatenate_0;

    /* ForEachSliceAssignment generated from: '<S1>/Out1' incorporates:
     *  ForEachSliceSelector generated from: '<S1>/In1'
     *  MATLAB Function: '<S1>/MATLAB Function'
     */
    /* MATLAB Function 'For Each Subsystem/MATLAB Function': '<S3>:1' */
    /* '<S3>:1:4' y = 2*u; */
    rtY.Out1[ForEach_itr] = 2.0 * rtb_VectorConcatenate_0;
  }
```

```
  /* End of Outputs for SubSystem: '<Root>/For Each Subsystem' */

  /* Outputs for Iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
   *  ForIterator: '<S2>/For Iterator'
   */
  /* Constant: '<Root>/Constant1' */
  ForEach_itr = ((int32_T)A);
  if (((int32_T)A) < 0) {
    ForEach_itr = 0;
  }

  /* End of Constant: '<Root>/Constant1' */
  for (s2_iter = 0; s2_iter < ForEach_itr; s2_iter++) {
    /* Assignment: '<S2>/Assignment' incorporates:
     *  Constant: '<S2>/Constant'
     *  Outport: '<Root>/Out2'
     *  Product: '<S2>/Product'
     *  Selector: '<S2>/Selector'
     */
    if (s2_iter == 0) {
      for (i = 0; i <= (int32_T)((int32_T)(A + B) - 1); i++) {
        rtY.Out2[i] = rtb_VectorConcatenate[i];
      }
    }

    rtY.Out2[s2_iter] = rtb_VectorConcatenate[s2_iter] * 2.0;

    /* End of Assignment: '<S2>/Assignment' */
  }

  /* End of Outputs for SubSystem: '<Root>/For Iterator Subsystem' */
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Set Parameter Value Based on Variant Choice

When you specify the dimensions of a parameter in a model by using a symbol, you must make sure that the parameter value is consistent with the dimension value. To simulate different choices for the dimension value, you must manually correct the parameter value.

For example, in the `rtwdemo_dimension_variants` model , the `Simulink.Parameter` object `Data` stores a vector value, `[1 2 3 4 5]`, and uses a symbolic dimension `C` with initial value `5`. If you change the value of `C`, to simulate the model, you must make sure the number of elements in the vector value matches the new value of `C`.

To reduce the effort of maintenance when you change the value of `C`, you can set the value of `Data` to an expression involving `C`.

1 Open the model.

```
rtwdemo_dimension_variants
```

2. At the command prompt, inspect the initial values of `Data` and `C`. The value of `Data` is a vector of integers from `1` to `C`.

```
Data.Value
```

```
ans =

     1     2     3     4     5
```

```
C.Value
```

```
ans =

     5
```

3. In MATLAB code syntax, the value of `Data` is `1:C`. To preserve this relationship between the parameter objects, set the value of `Data` by using the `slexpr` function.

```
Data.Value = slexpr('1:C');
```

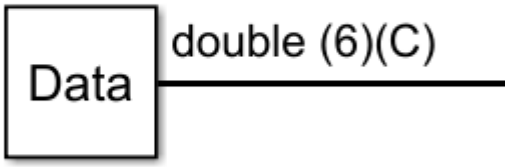
4. To prevent data type propagation errors, set the data type of `Data` explicitly to `double`, which is the data type that the parameter acquired before you set the parameter value to an expression.

```
Data.DataType = 'double';
```

5. Set the value of `C` to a different number, such as `6`. Due to dimension constraints in the model, you must set the value of another dimension symbol, `A`, to `3`.

```
C.Value = 6;
A.Value = 3;
```

6. Simulate the model. The block diagram shows that the value of `Data` now has six elements.



For more complicated applications, you can write your own MATLAB function that returns parameter values based on dimension symbols. Set the value of the parameter data to an expression that calls your function.

For general information about using an expression to set the value of a `Simulink.Parameter` object, see “Set Variable Value by Using a Mathematical Expression”.

Code Generation Optimization Considerations

When you create a model with symbolic dimensions, be aware of the following optimization considerations:

- The code generator reuses buffers only if dimension propagation establishes equivalence among buffers.
- Two loops with symbolic loop bound calculations are fused together only if they share equivalent symbolic expression.
- Optimizations do not eliminate a symbolic expression or condition check based on the current value of a symbolic dimension.

Backward Compatibility

If an existing model uses `Simulink.Parameter` objects to specify dimensions, it can be incompatible with dimension variants. Here are two common scenarios:

- Only a subset of blocks accepts symbolic dimension specifications. If a block is not compatible with symbolic dimensions, it causes an update diagram error.
- `Simulink.Parameter` objects that you use to define symbolic dimensions or have symbolic dimensions must have one of the storage classes described in this example. If these specifications are not met, the build procedure for the model fails during code generation.

You can address these backward compatibility issues by doing the following:

- Turn off dimension variants feature by clearing the **Allow symbolic dimension specification** parameter in the Configuration Parameters dialog box.
- Update `Simulink.Parameter` objects that define symbolic dimensions or have symbolic dimension specifications.
- Update the model so that only supported blocks have symbolic dimensions or propagate symbolic dimensions.

Supported Blocks

For a list of supported blocks, see the Block Support Table. To access the information in this table, enter `showblockdatatypetable` at the MATLAB command prompt. Unsupported blocks (for example, MATLAB Function) can still work in a model containing symbolic dimensions as long as these blocks do not directly interact with symbolic dimensions.

In the following cases, supported blocks do not propagate symbolic dimensions.

- For Assignment and Selector blocks, you set the **Block Parameters** > **Index Option** parameter to `Index vector (dialog)`. For Selector and Assignment blocks, if you specify a symbolic dimension for the Index parameter, the code generator does not honor the symbolic dimension in the generated code.
- For the Product block, you specify a value of `1` for the **Block Parameters** > **Number of inputs** parameter, and you set the **Multiply over** parameter to `Specified dimensions`.
- For the ForEach block, you specify a symbolic dimension for the **Partition Width** parameter.

Note that the following modeling patterns are among those modeling patterns that can cause Simulink to error out:

- For Switch blocks, an input signal or the **Threshold** parameter has symbolic dimensions, and you select **Allow different data input sizes (Results in variable-size output signal)**.
- A Data Store Read block selects elements of a `Simulink.Bus` signal that has symbolic dimensions.

- For Lookup Table blocks, on the **Block Parameters** > **Algorithm** tab, you select the parameter **Use one input port for all input data**.

Limitations

The following products and software capabilities support dimension variants in that they act on the numeric value of a symbolic dimension. These features do not support the propagation of symbolic dimensions during model simulation and the preservation of symbolic dimensions in the generated code.

- Code Replacement for Lookup Tables
- Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) simulations
- Accelerator and rapid accelerator simulation modes
- Scope and simulation observation (for example, logging, SDI, and so on)
- Model coverage
- Simulink Design Verifier
- Fixed-Point Designer
- Data Dictionary
- Simulink PLC Coder
- HDL Coder

The following do not support dimension variants:

- System Object
- Stateflow charts that use MATLAB as the action language
- Physical modeling
- Discrete-event simulation
- Frame data
- MATLAB functions

The following limitations also apply to models that utilize symbolic dimensions.

- For simulation, the size of a symbolic dimension can equal 1. For code generation, the size of a symbolic dimension must be greater than `1`.
- If a symbolic dimension is a MATLAB expression that contains an arithmetic expression and either a relational or logical expression, you must add `+0` after the relational or logical part of the MATLAB expression. If you do not add `+0`, the model errors out during simulation because you cannot mix a `boolean` data type with integer or `double` data types. Adding `+0` converts the data type of the relational or logical part of the expression from a `boolean` to a `double`.

  For example, suppose in the Inport block parameters dialog box, the **Port dimensions** parameter has the expression `[(C==8)*D+E,3]`. The **Data type** parameter is set to `double`. Since `C==8` is a relational expression, you must change the expression to `[((C==8)+0)*D+E,3]` to prevent the model from producing an error during simulation.
- Simulink propagates symbolic dimensions for an entire structure or matrix, but not for a part of a structure or matrix. For example, the `Simulink.Parameter` `P` is a `2x3` matrix with symbolic dimensions `[Dim,Dim1]`.

```
p=Simulink.Parameter(struct('A',[1 2 3;4 5 6]))
p.DataType='Bus:bo'
bo=Simulink.Bus
bo.Elements(1).Name='A'
bo.Elements(1).Dimensions='[Dim,Dim1]'
Dim=Simulink.Parameter(2)
Dim1=Simulink.Parameter(3)
p.CoderInfo.StorageClass='Custom'
p.CoderInfo.CustomStorageClass='Define'
Dim.CoderInfo.StorageClass='Custom'
Dim.CoderInfo.CustomStorageClass='Define'
Dim1.CoderInfo.StorageClass='Custom'
Dim1.CoderInfo.CustomStorageClass='Define'
```

If you specify `p.A` for a dimensions parameter, Simulink propagates the symbolic dimensions `[Dim,Dim1]`. If you specify `p.A(1,:)`, Simulink propagates the numeric dimension `3` but not the symbolic dimension, `Dim1`.

- The MATLAB expression `A(:)` does not maintain symbolic dimension information. Use `A` instead.
- The MATLAB expression `P(2:A)` does not maintain symbolic dimension information. Use the Selector block instead.
- The MATLAB expression `P(2,:)` is not a tunable expression, so it does not maintain symbolic dimension information.
- Suppose that you set the value of a mask parameter, `myMaskParam`, by using a field of a structure or by using a subset of the structures in an array of structures. You store the structure or array of structures in a `Simulink.Parameter` object so that you can use a `Simulink.Bus` object to apply symbolic dimensions to the structure fields. Under the mask, you configure a block parameter to use one of the fields that have symbolic dimensions. The table shows some example cases.

Description	Value of mask parameter (`myMaskParam`)	Value of block parameter
`myStruct` is a structure with field `gains`, which uses symbolic dimensions.	`myStruct.gains`	`myMaskParam`
`myStruct` is a structure with field hierarchy `myStruct.subStruct.gains`. The field `gains` uses symbolic dimensions.	`myStruct.subStruct`	`myMaskParam.gains`
`myStructs` is an array of structures. Each structure has a field `gains`, which uses symbolic dimensions.	`myStructs(2)`	`myMaskParam.gains`

In these cases, you cannot generate code from the model. As a workaround, choose one of these techniques:

- Use the entire structure (`myStruct`) or array of structures (`myStructs`) as the value of the mask parameter. Under the mask, configure the block parameter to dereference the target field from the mask parameter by using an expression such as `myMaskParam.subStruct.gains`.

- Use literal dimensions instead of symbolic dimensions for the target field (`gains`).

This limitation also applies when you use a field of a structure or a subset of the structures in an array of structures as the value of a model argument in a Model block.

See Also

Related Examples

- "Configure Symbolic Dimensions for S-Function Blocks" on page 27-45

Code Generation for Variant Blocks

The code generator produces code from a Simulink model containing one or more Variant Subsystem, Variant Model, Variant Source, and Variant Sink blocks. To learn how to create a model containing variant blocks, see “Create a Simple Variant Model”. To learn how to create a custom Model Advisor check that evaluates the generated code from active and inactive variant paths in a variant system model, see “Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model” (Simulink Check).

Note During code generation, a warning related to 'Code generation function packaging' for the Variant block might be displayed. In the block parameters dialog of the Variant block, click **Code Generation** and then select `Reusable function` from the **Function packaging** drop-down list to solve this issue.

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

- Select `Expression` as **Variant control mode**.
- Select `code compile` as **Variant activation time**.

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

To construct variant models and generate preprocessor directives in the generated code, see the example “Use Variant Models to Generate Code That Uses C Preprocessor Conditionals” on page 27-70.

To construct variant subsystems and generate preprocessor directives in the generated code, see the example “Use Variant Subsystem to Generate Code That Uses C Preprocessor Conditionals” on page 27-65.

To construct models with variant sources and sinks and generate preprocessor directives in the generated code, see the example “Represent Variant Source and Sink Blocks in Generated Code” on page 27-37.

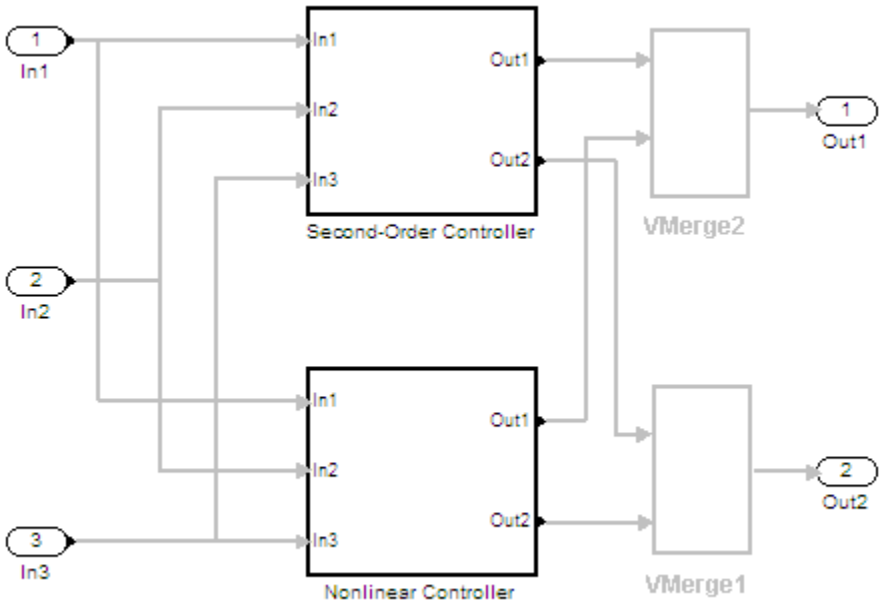
Restrictions on Variant Subsystem Code Generation

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

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

When compared to a generic Merge block the `VariantMerge` block can have only one parameter which is the number of Inputs. The `VariantMerge` block is used for code generation in variant

subsystems internally, and is not available externally to be used in models. The number of inputs for `VariantMerge` is determined and wired as shown in the figure below.



The child subsystems of the Variant Subsystem block must be atomic subsystems. Select **Treat as atomic unit** parameter in the Subsystem block parameters dialog, to make the subsystems atomic. The `VariantMerge` blocks are inserted at the outport of the subsystems if more than one child subsystems are present. If the source block of a `VariantMerge` block input is nonvirtual, an error message will be displayed during code generation. You must make the source block contiguous, by inserting Signal Conversion blocks inside the variant choices. The signals that enter a Variant Subsystem block must have the same signal properties (for example, signal dimensions, port width, and storage class). The `VariantMerge` block does not support different signal properties because the input ports and output ports share the same memory. You can use symbolic dimensions to generate code for a variant subsystem with child subsystems of different output signal dimensions.

Generated Code Components Not Compiled Conditionally

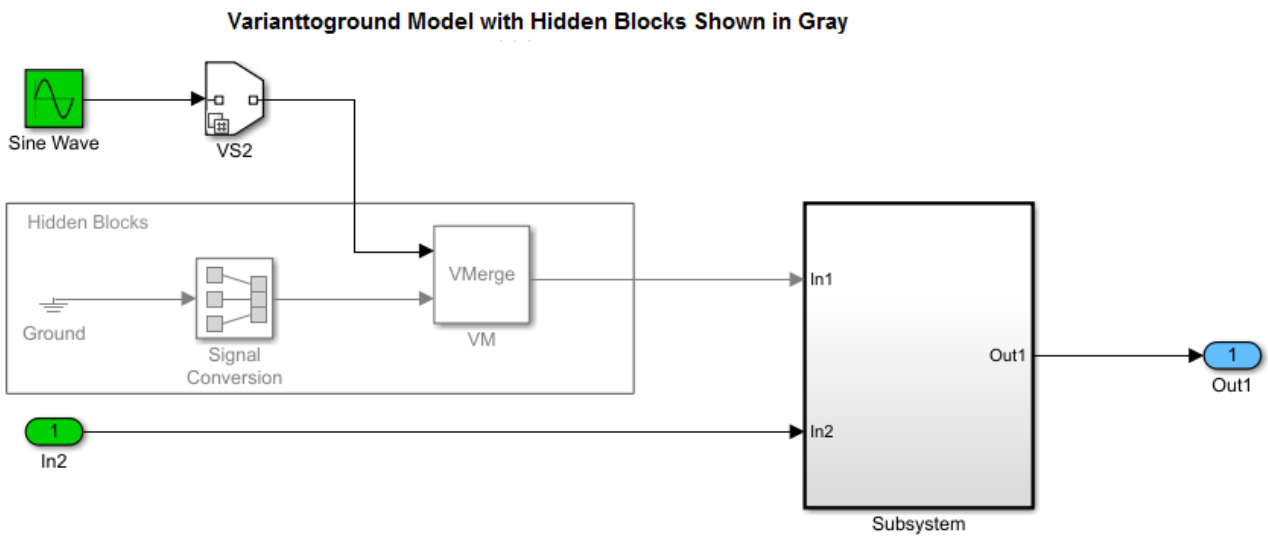
The following components are not conditionally compiled even if only code for variant subsystems or models that are conditionally compiled reference them.

- `rtModel` data structure fields
- `#include`'s of utility files
- Global constant parameter structure fields that are referenced by multiple subsystems activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are referenced by multiple subsystems that are activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are used by variant model blocks

Code Generation for Variant Blocks with One Variant Choice

For modeling patterns in which a Root Inport block connects to a Variant block with one variant choice, Simulink inserts a hidden block combination of a Ground block, Signal Conversion block, and a Variant Merge block. If the variant choice evaluates to false, this block combination produces an output of `0.0`.

For example, the model `Varianttoground` contains a Variant Source block with one variant choice. When the Variant Control `SYSCONST_A==6` evaluates to true, the input to `Subsystem` is a sine wave. When `SYSCONST_A==6` evaluates to false, the input to `Subsystem` is `0.0`.



The `varianttoground.c` file contains this code:

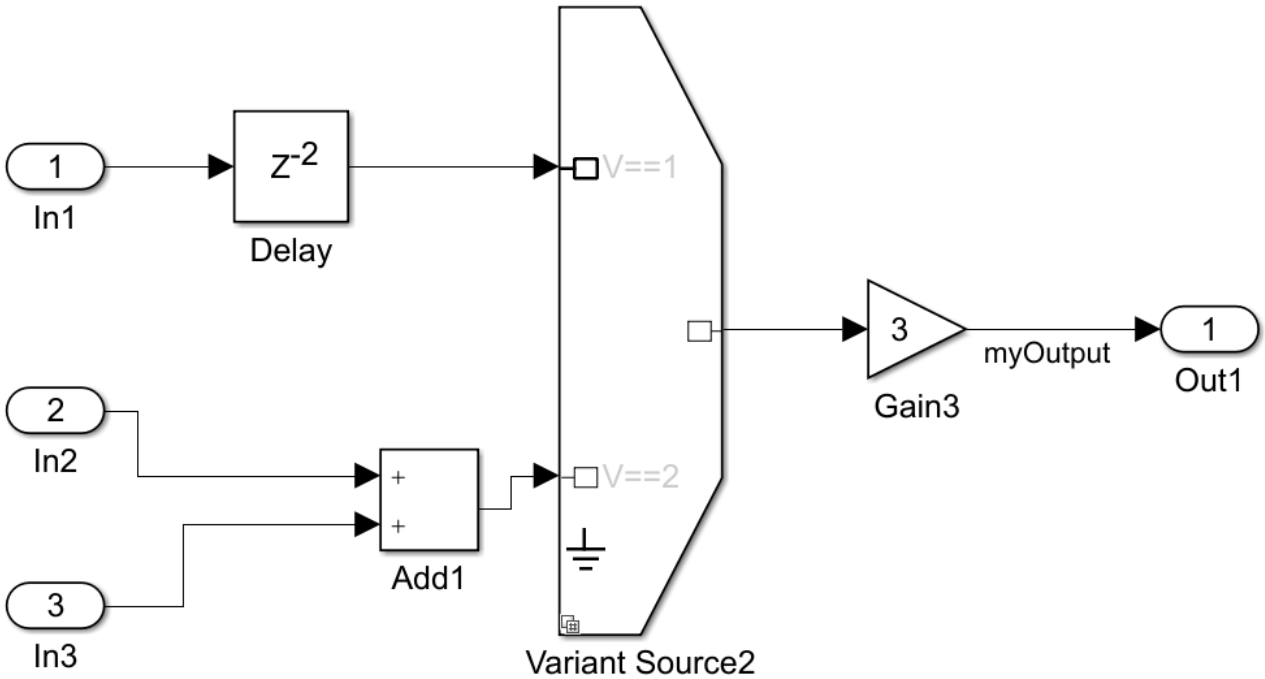
```
* Sin: '<Root>/Sine Wave' incorporates:
   *  SignalConversion generated from: '<Root>/Subsystem'
   */
#if SYSCONST_A == 6

  Varianttoground_B.VM_Conditional_Signal_Subsystem_0 = sin
    (Varianttoground_M->Timing.t[0]);

#else

  /* SignalConversion generated from: '<Root>/Subsystem' */
  Varianttoground_B.VM_Conditional_Signal_Subsystem_0 = 0.0;

#endif

  /* End of Sin: '<Root>/Sine Wave' */
```

The comments in the generated code indicate the presence of the hidden signal conversion block. In the hyperlinked comments, you can trace to the original block in the model that triggered the insertion of the hidden block. The code does not contain a comment for the Variant Merge block because this block does not have associated generated code. The Variant Merge block is used internally and is not in the Simulink library.

Global Data Guarding: Signals and States

When a model has variant blocks, the global signals and states are guarded in the generated code with their corresponding variant condition.

Consider a model containing a Variant Source block. The Variant Source block has conditions `V==1` and `V==2`.



In the generated code, global signals are guarded in declaration, model initialization, and definition of data as shown below.

Signals guarding:

Declaration:

```
/* Exported block signals */
#if V == 1 || V == 2

real_T myOutput;                        /* '<Root>/Gain3' */

#endif
```

Model initialization:

```
  /* exported global signals */
#if V == 1 || V == 2

  myOutput = 0.0;

#endif
```

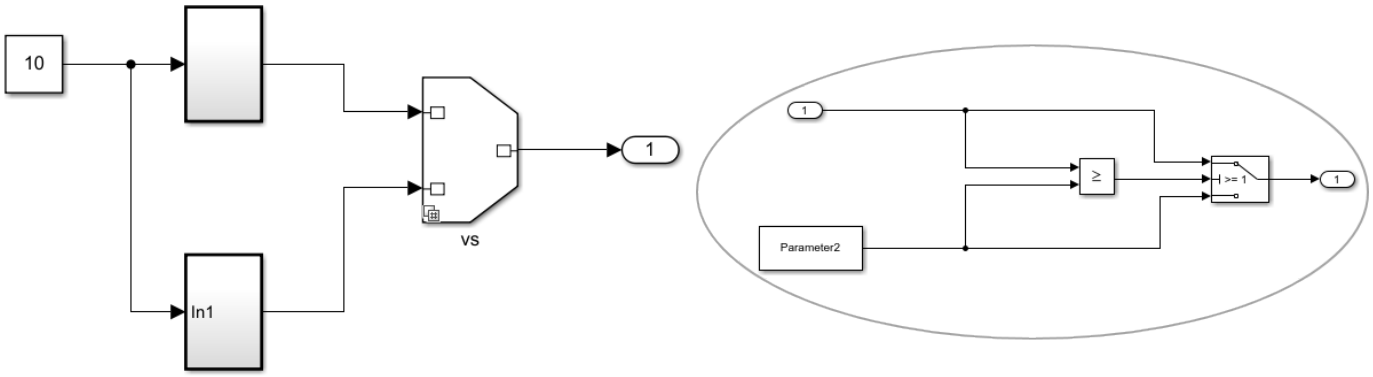
Definition of data:

```
 * Exported Global Signals
 *
 * Note: Exported global signals are block signals with an exported global
 * storage class designation.  Code generation will declare the memory for
 * these signals and export their symbols.
 *
 */
#if V == 1 || V == 2

extern real_T myOutput;                /* '<Root>/Gain3' */

#endif
```

Global Data Guarding: Parameters

When you generate code for a model containing variant blocks, the parameters are guarded in the generated code.

Consider a model with Constant block attached to a Variant Source block.



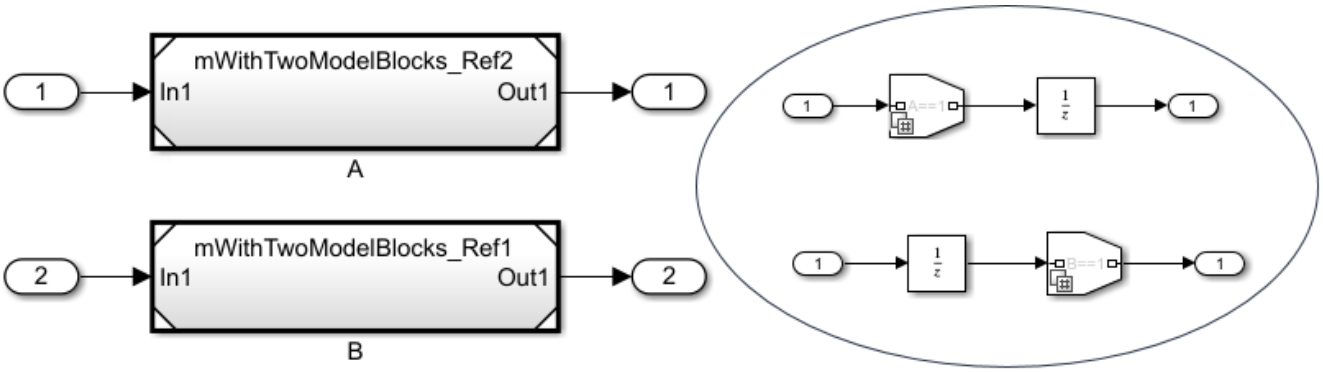
In the generated code, the parameter of the Constant block is guarded in header file as shown below.

```
/* Exported data definition */
/* Const memory section */
/* Definition for custom storage class: Const */
#if V == 2

const int32_T Parameter2 = 1;

#endif
```

Guarding Reference Model Headers

When a model has Model Reference blocks that are conditional due to inline variants, the header file and the instances referring to the Model Reference blocks are guarded.

Consider this model with two Model References blocks that are conditional due to inline variants.



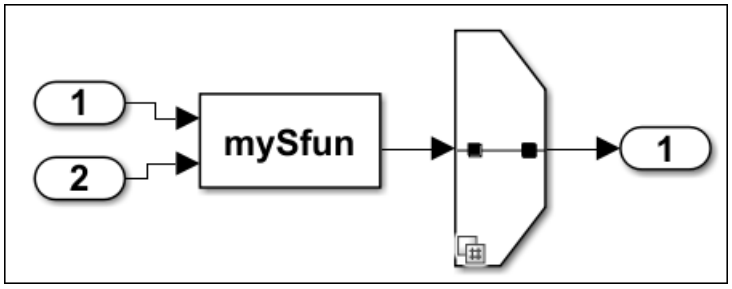
When you generate code for this model, the header file and instances referring to the Model Reference blocks are guarded as shown in the code below.

```
#include "mWithTwoModelBlocks_Top_types.h"
#include "multiword_types.h"

#if (A == 1) //Guarding
#define mWithTwoModelBlocks_Ref2_MDLREF_HIDE_CHILD_
#include "mWithTwoModelBlocks_Ref2.h"
#endif

#if (B == 1) //Guarding
#include "mWithTwoModelBlocks_Ref1.h"
#endif
```

Guarding S-Function Block Headers

When a model has an S-Function block that is conditional due to any connected variant blocks, Simulink guards the header files associated with the S-Function block with the corresponding variant condition in the generated code.

Consider a model with an S-Function block connected to a Variant Source block. In the generated code, the header file is guarded with the variant condition.




```
#ifndef mSFunctionInclude_COMMON_INCLUDES_
#define mSFunctionInclude_COMMON_INCLUDES_
#include "rtwtypes.h"
#include "rt_logging.h"
#if V == 1  //Guarding
#include "myHeader.h"
```

```
#endif
#endif                                  /* mSFunctionInclude_COMMON_INCLUDES_ */
```

Represent Subsystem and Variant Models in Generated Code

In this section...

"Step 1: Represent Variant Choices in Simulink" on page 27-22
"Step 2: Specify Conditions That Control Variant Choice Selection" on page 27-25
"Step 3: Configure Model for Generating Preprocessor Conditionals" on page 27-27
"Step 4: Review Generated Code" on page 27-28
"Limitations" on page 27-31

Required products: Simulink, Embedded Coder, Simulink Coder

Using Simulink, you can create models that are based on a modular design platform that comprises a fixed common structure with a finite set of variable components. The variability helps you develop a single design with variable components. For more information, see "What Are Variants and When to Use Them". When you implement variants in the generated code, you can:

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

Using Embedded Coder, you can generate code from Simulink models containing one or more variant choices. The generated code contains preprocessor conditionals that control the activation of each variant choice.

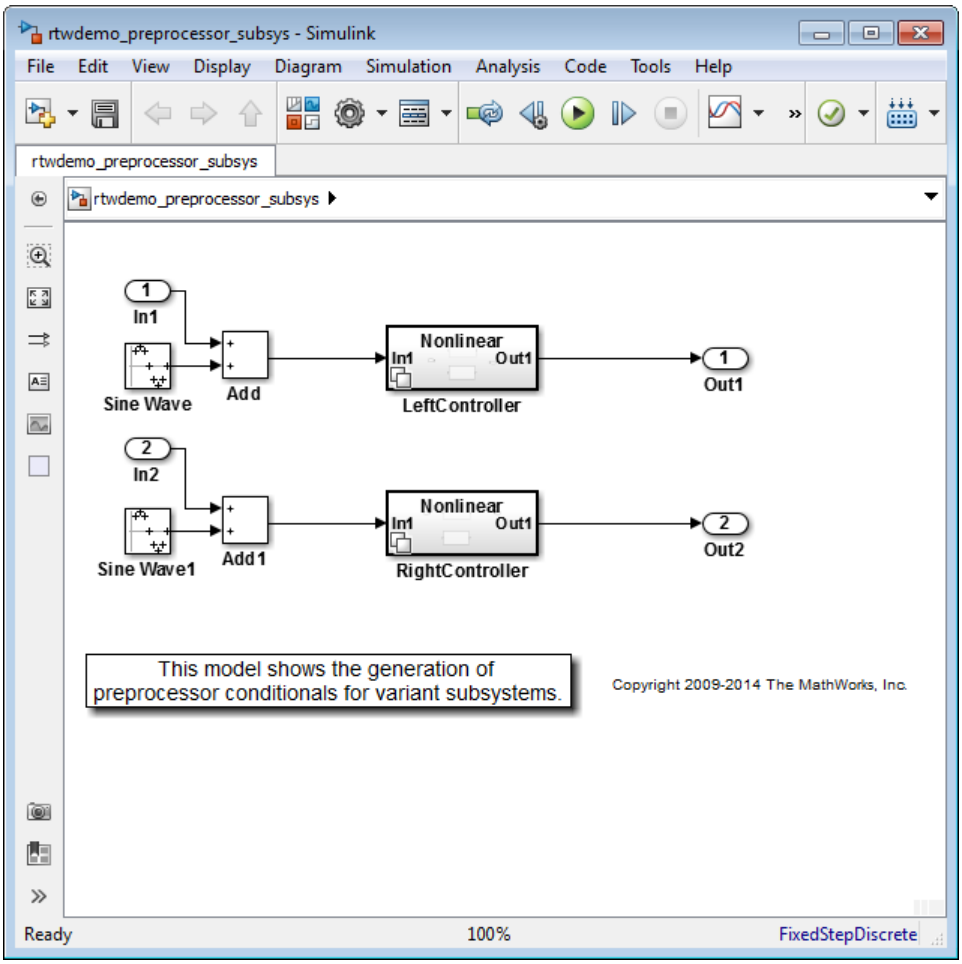
This example shows how to represent variant choices in a Simulink model and then prepare the model so that those variant choices are represented in generated code.

Step 1: Represent Variant Choices in Simulink

Variant choices are two or more configurations of a component in your model. This example uses the model `rtwdemo_preprocessor_subsys` to illustrate how to represent variant choices inside Variant Subsystem blocks. For other ways to represent variant choices, see "Compare Variant Blocks".

1. Open the model `rtwdemo_preprocessor_subsys`.

   ```
   open_system('rtwdemo_preprocessor_subsys')
   ```

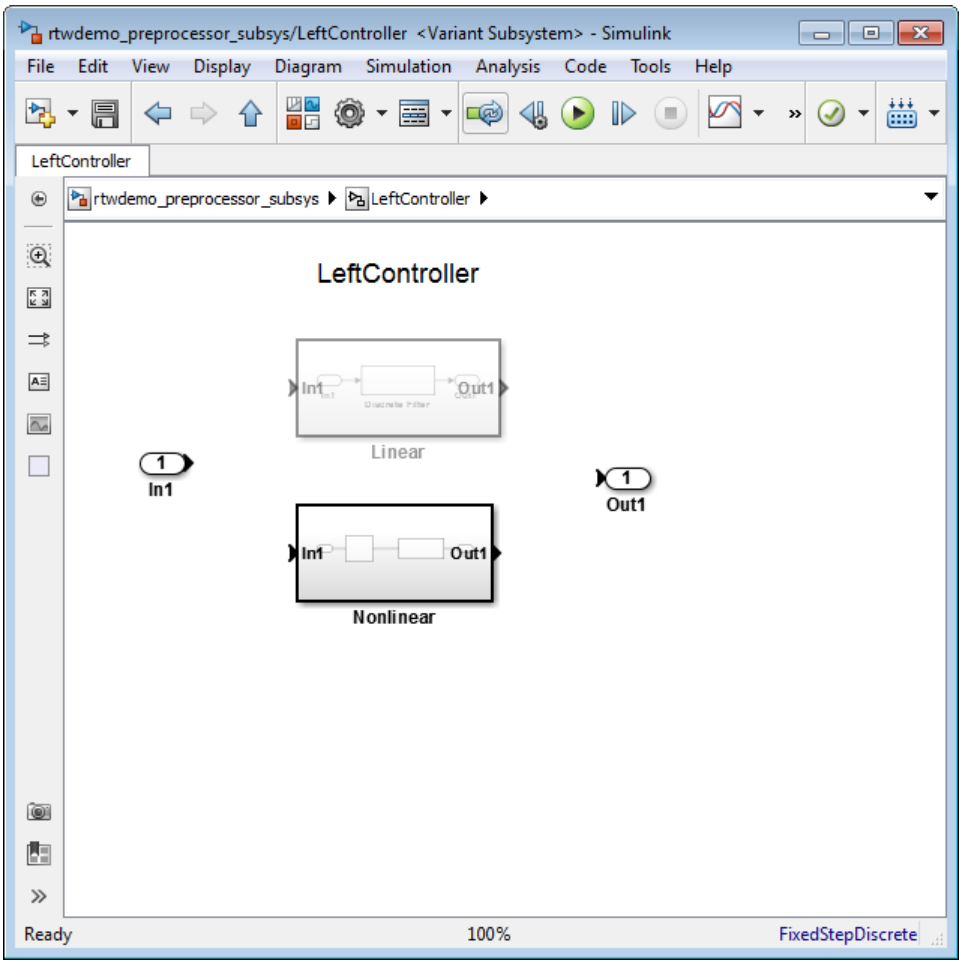


The model contains two Variant Subsystem blocks: **LeftController** and **RightController**.

Note You can only add Inport, Outport, Subsystem, and Model blocks inside a Variant Subsystem block.

---

2 Open the **LeftController** block.

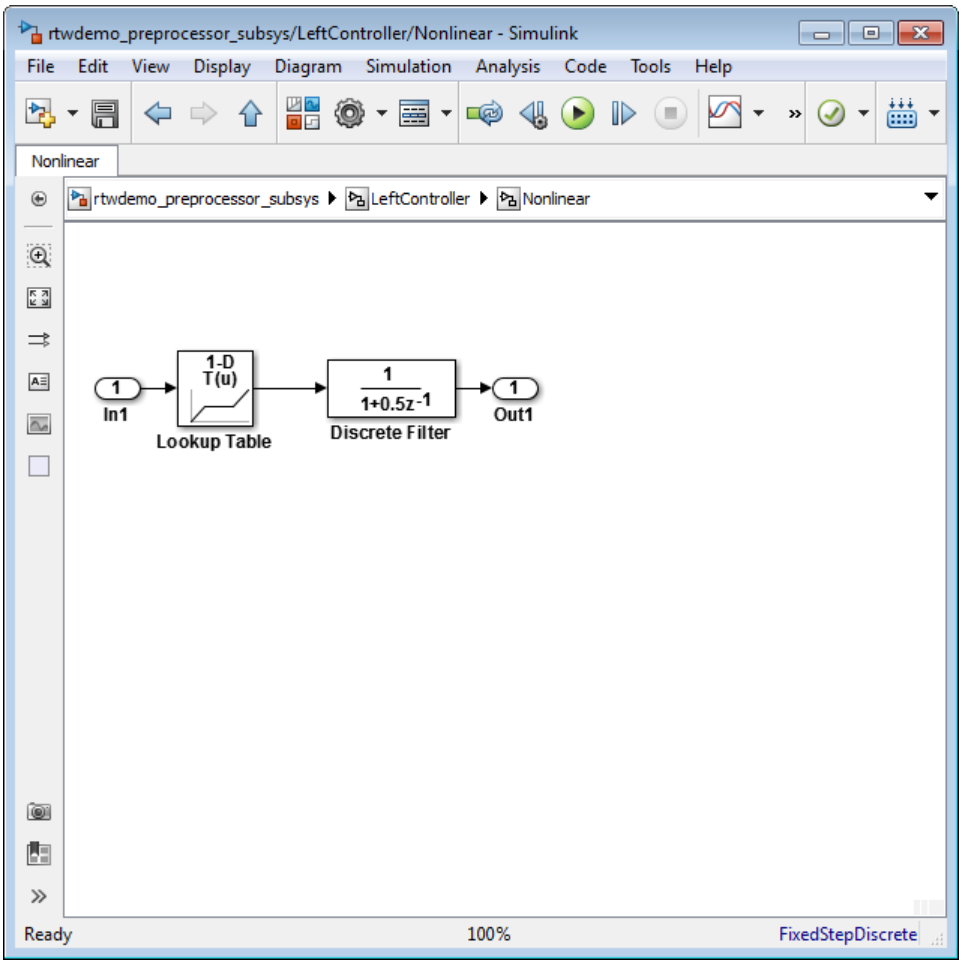


The **LeftController** block serves as the container for the variant choices. It contains two variant choices represented using Subsystem blocks **Nonlinear** and **Linear**. The nonlinear controller subsystems implement hysteresis, whereas the linear controller subsystems act as simple low-pass filters.

The Subsystem blocks have the same number of inports and outports as the containing Variant Subsystem block.

Variant choices can have different numbers of inports and outports. See “Map Inports and Outports of Variant Choices in Variant Subsystem”.

3 Open the **Nonlinear** block.



The **Nonlinear** block represents one variant choice that Simulink activates when a condition is satisfied. The **Linear** block represents another variant choice.

Tip When you are prototyping variant choices, you can create empty Subsystem blocks with no inputs or outputs inside a Variant Subsystem block. The empty subsystem recreates the situation in which that subsystem is inactive without the need for completely modeling the variant choice.

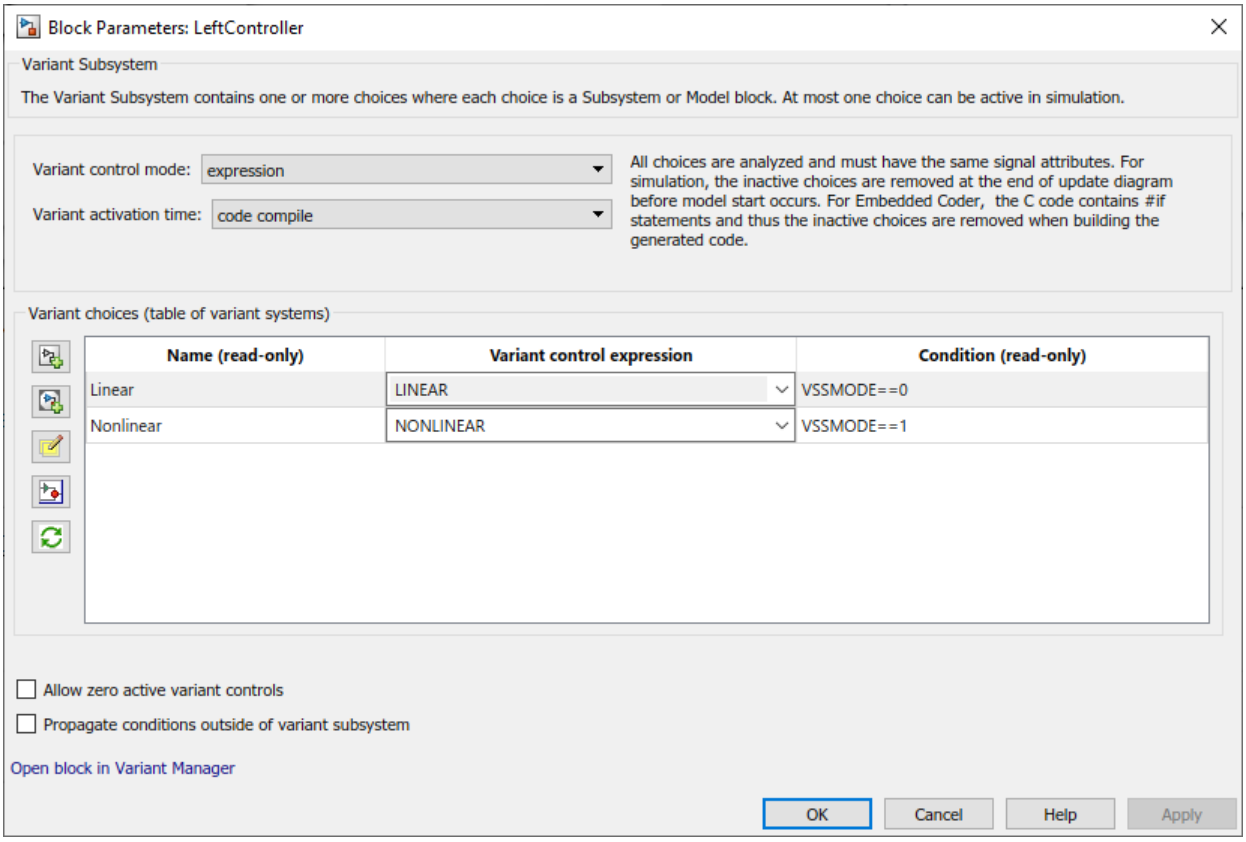
Step 2: Specify Conditions That Control Variant Choice Selection

You can switch between variant choices by constructing conditional expressions called variant controls for each variant choice represented in a Variant Subsystem block. Variant controls determine which variant choice is active, and changing the value of a variant control causes the active variant choice to switch.

A variant control is a Boolean expression that activates a specific variant choice when it evaluates to `true`.

For more information, see “Introduction to Variant Controls”.

1. Right-click the **LeftController** block and select **Block Parameters (Subsystem)**.



The **Condition** column displays the Boolean expression that when `true` activates each variant choice. In this example, these conditions are specified using `Simulink.Variant` objects `LINEAR` and `NONLINEAR`.

2 Use these commands to specify a variant control using a `Simulink.Variant` object.

```
LINEAR = Simulink.Variant;
LINEAR.Condition = 'VSSMODE==0';
NONLINEAR = Simulink.Variant;
NONLINEAR.Condition = 'VSSMODE==1';
```

Here, `VSSMODE` is called a variant control variable that can be specified in one of the ways listed in “Types of Variant Control Variables (Operands) in Variant Blocks”.

3 Define the variant control variable `VSSMODE`.

You can define `VSSMODE` as a scalar variable or as a `Simulink.Parameter` object. In addition to enabling the specification of parameter value, `Simulink.Parameter` objects allow you to specify other attributes such as data type that are required for generating code.

```
VSSMODE = Simulink.Parameter;
VSSMODE.Value = 1;
VSSMODE.DataType = 'int32';
VSSMODE.CoderInfo.StorageClass = 'Custom';
VSSMODE.CoderInfo.CustomStorageClass = 'ImportedDefine';
VSSMODE.CoderInfo.CustomAttributes.HeaderFile = 'rtwdemo_importedmacros.h';
```

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes.

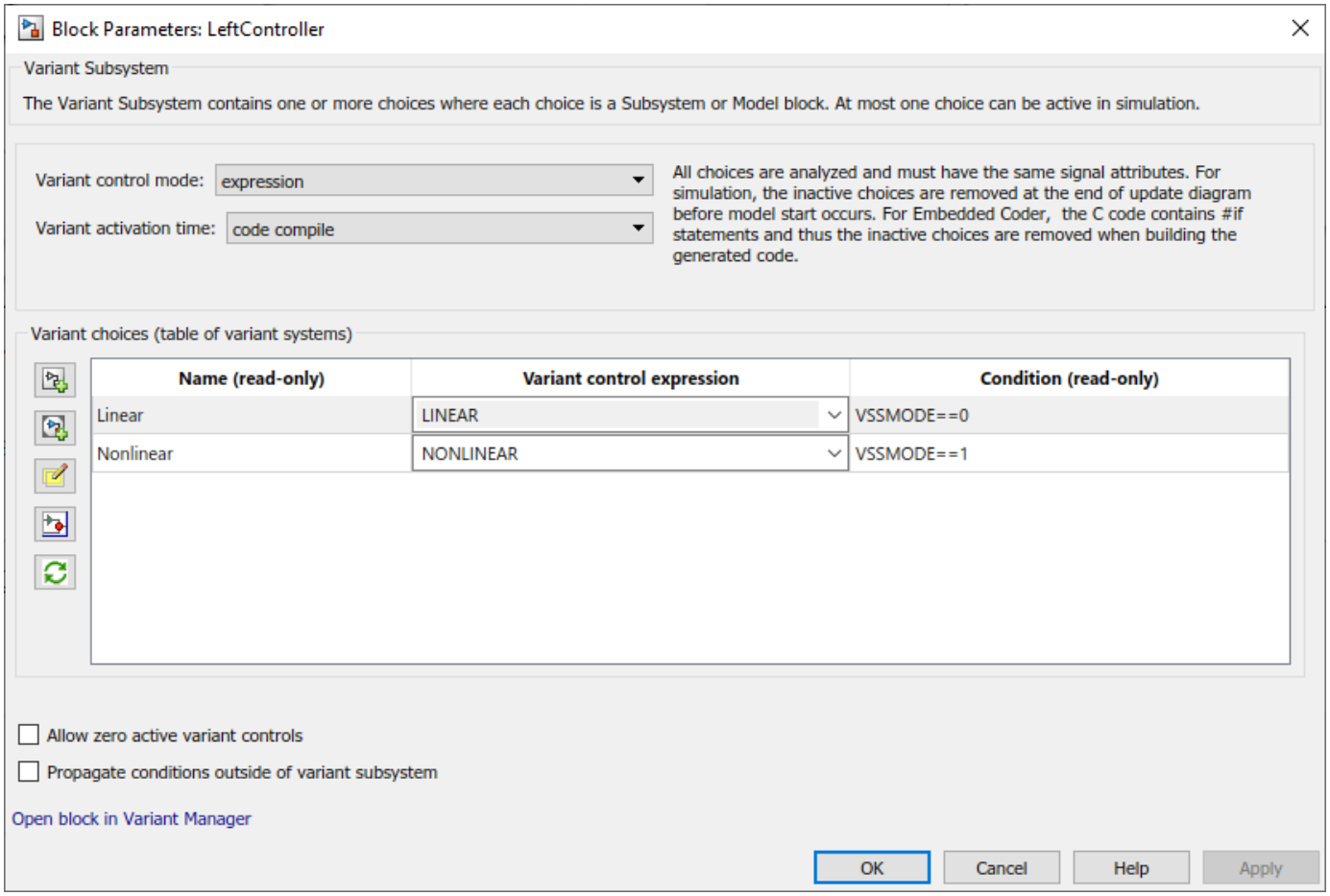
- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- Your own custom storage class that defines data as a macro

You can also convert a scalar variant control variable into a `Simulink.Parameter` object. See "Convert Variant Control Variables into Simulink.Parameter Objects".

Step 3: Configure Model for Generating Preprocessor Conditionals

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

1. In the **Modeling** tab of the Simulink toolstrip, click **Model Settings**.
2. Select the **Code Generation** pane, and set **System target file** to `ert.tlc`.
3. In the **Report** pane, select **Create code generation report**.
4. In the Configuration Parameters dialog box, clear **Ignore custom storage classes** and click **Apply**.
5. In your model, right-click the **LeftController** block and select **Block Parameters (Subsystem)**.
6. Set the **Variant activation time** parameter to `code compile`.



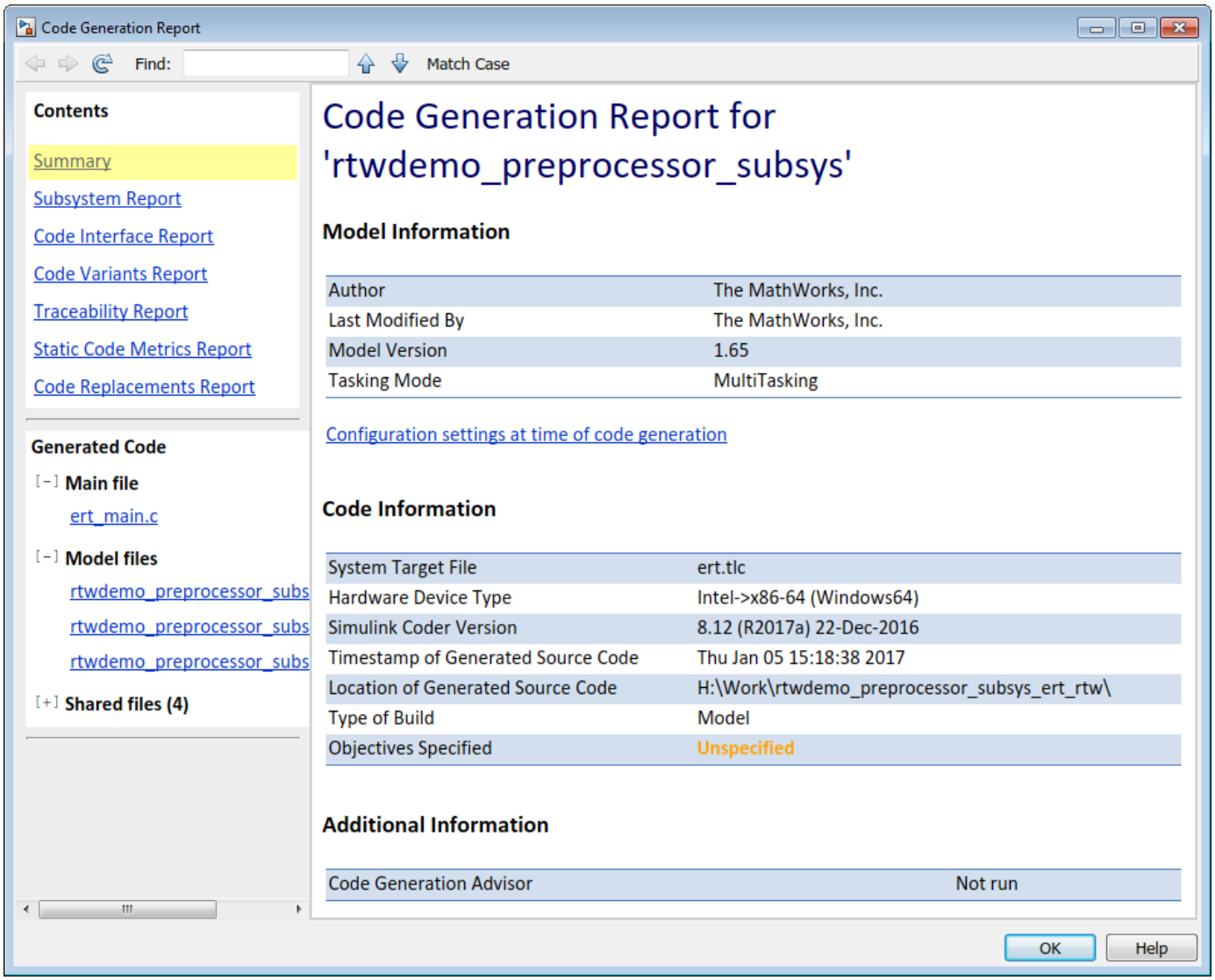
When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

7 Build the model.

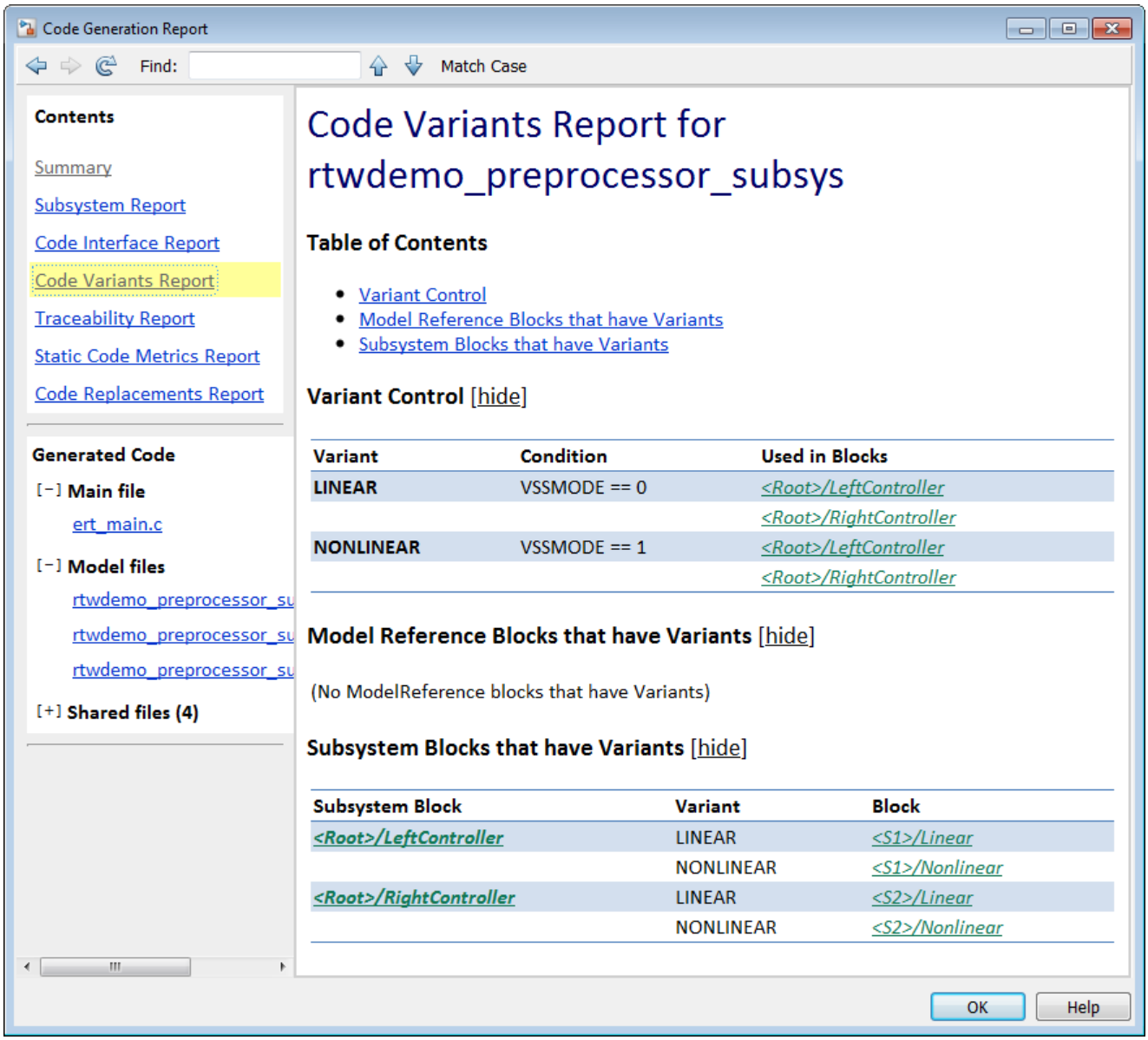
Step 4: Review Generated Code

The code generation report contains a section dedicated to the subsystems that have variants controlled by preprocessor conditionals.

1 In the **C Code** tab of the toolstrip, select **Open Report**.



2 Select the **Code Variant Report** from the left.



In this example, the generated code includes references to the `Simulink.Variant` objects `LINEAR` and `NONLINEAR`. The code also includes the definitions of macros corresponding to those variants. The definitions depend on the value of `VSSMODE`, which is supplied in an external header file `rtwdemo_importedmacros.h`. The active variant is determined by using preprocessor conditionals (`#if`) on the macros (`#define`) `LINEAR` and `NONLINEAR`.

3 Select the `rtwdemo_preprocessor_subsys_types.h` file from the left.

This file contains the definitions of macros `LINEAR` and `NONLINEAR`.

```
#ifndef LINEAR
    #define LINEAR      (VSSMODE == 0)
#endif

#ifndef NONLINEAR
```

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

4 Select the `rtwdemo_preprocessor_subsys.c` file from the left.

In this file, calls to the step and initialization functions of each variant are conditionally compiled.

```
    /* Outputs for Atomic SubSystem: '<Root>/LeftController' */
#if LINEAR
    /* Output and update for atomic system: '<S1>/Linear' */
    ...
#elif NONLINEAR
    /* Output and update for atomic system: '<S1>/Nonlinear' */
    ...
#endif
```

Limitations

- When you are generating code for Variant Subsystem blocks, the blocks cannot have:
  - Mass matrices
  - Function call ports
  - Outports with constant sample time
  - Simscape blocks
- The port numbers and names for each active child subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.

See Also

Related Examples

- “Implement Variations in Separate Hierarchy Using Variant Subsystems”

More About

- “What Are Variants and When to Use Them”
- “Introduction to Variant Controls”

Generate Preprocessor Conditionals for Variant Systems

In this section...

"Define Variant Controls" on page 27-32
"Configure Model for Generating Preprocessor Conditional Directives" on page 27-33
"Special Considerations for Generating Preprocessor Conditionals" on page 27-33
"Generate Variant Control Macros in Same Header File" on page 27-34

Define Variant Controls

For variant systems, variant controls determine which variant choice is active. You can specify a variant control as a condition expression, a `Simulink.Variant` object specifying a condition expression, a MATLAB variable, or a `Simulink.Parameter` object. This example shows how to define variant controls as `Simulink.Parameter` objects.

1. Open the Model Explorer. Select the **base workspace**. Alternatively, you can create a `Simulink.Parameter` in data dictionary.
2. In the Model Explorer, select **Add** > **Simulink Parameter**. Specify a name for the new parameter.
3. Use the function `Simulink.VariantManager.findVariantControlVars` to find and convert MATLAB variables used in variant control expressions into `Simulink.Parameter` objects. For an example, see "Convert Variant Control Variables into Simulink.Parameter Objects".
4. On the `Simulink.Parameter` property dialog box, specify the **Value** and **Data type**.
5. Select one of these **Storage class** values.
   - `Define`
   - `ImportedDefine(Custom)`
   - `CompilerFlag(Custom)`
   - A storage class created using the Custom Storage Class Designer. Your storage class must have the **Data initialization** parameter set to `Macro` and the **Data scope** parameter set to `Imported`. See "Control Data Representation by Configuring Storage Class Properties" on page 40-17 for more information.
6. Specify the value of the variant control. If the storage class is `ImportedDefine(Custom)`, do the following:
   a. Specify the **Header File** parameter as an external header file in the Custom Attributes section of the `Simulink.Parameter` property dialog box.
   b. Enter the values of the variant controls in the external header file.

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

   If the variant control is a `CompilerFlag` custom storage class, the value of the variant control is set at compile time. Use the **Configuration Parameters** > **Code Generation** >

Custom Code > **Code information** > **Defines** parameter to add a list of variant controls (macro definitions) to the compiler command line. For example, for variant control `VSSMODE`, in the text field for the **Defines** parameter, enter:

```
-DVSSMODE=1
```

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

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

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

Configure Model for Generating Preprocessor Conditional Directives

1. Open the Configuration Parameters dialog box.
2. Select the **Code Generation** pane, and set **System target file** as `ert.tlc`.
3. In the **Report** pane, select **Create code generation report**.
4. In the Configuration Parameters dialog box, clear the Ignore custom storage classes parameter. In order to generate preprocessor conditionals, you must use custom storage classes.
5. In the Variant Subsystem, Variant Source, or Variant Sink block parameter dialog boxes, set the **Variant activation time** parameter to `code compile`.
6. Generate code.

Special Considerations for Generating Preprocessor Conditionals

- The port numbers and names for each child variant subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.
- The code generation process checks that there is at least one active variant by using the variant control values stored in the base workspace. The variant control that evaluates to `true` becomes the active variant. If none of the variant controls evaluates to `true`, the default variant, if specified, becomes the active variant. The code generation process issues an error if an active variant does not exist.
- Implement the condition expressions of the variant controls such that only one variant control evaluates to `true`. The generated code includes a test of the variant controls to determine that there is only one active variant. If this test fails, your code will not compile.
- If you comment out child subsystems listed in the **Variant Choices** table in the Variant Subsystem block parameter dialog box, the code generator does not generate code for the commented out subsystems.
- For Variant Subsystems, the `model_private.h` file contains conditional parameter definitions. For example, if the value of a Constant block is a `Simulink.Parameter` with an `ImportedDefine` custom storage class, and the Constant block is in a Variant Subsystem, the conditional definition of the `Simulink.Parameter` is in the `model_private.h` file.

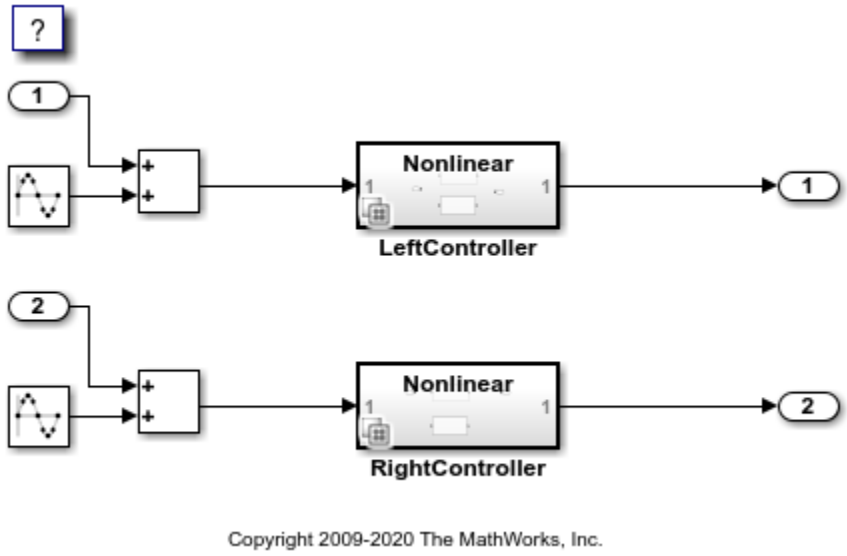
Generate Variant Control Macros in Same Header File

This example shows how to aggregate multiple variant control macros (`#define`) into the same generated header file. This aggregation makes it easier for you to manage the complexities inherent in a system with multiple interacting variant conditions.

Explore Example Model

Run the script `prepare_rtwdemo_preproc_subsys`, which opens the model `rtwdemo_preprocessor_subsys` and prepares it for this example.

```
prepare_rtwdemo_preproc_subsys;
```



The model contains two Variant Subsystem blocks.

Navigate inside the variant subsystems. The subsystems each have a linear and a nonlinear algorithm variant.

At the root level of the model, open the block dialog box of the variant subsystem labeled `LeftController`. The algorithm variants in the subsystem activate based on the states of two `Simulink.Variant` objects, `LINEAR` and `NONLINEAR`, in the base workspace.

The state of each object depends on the value of the variant control variable, `VSSMODE`, which is a `Simulink.Parameter` object in the base workspace. The parameter object uses the custom storage class `Define` and is configured to appear in the generated code as a C-code macro in `macros.h`.

Change Name of Generated Header File Through Model Explorer

In this example, change the name of the generated header file from `macros.h` to `variant_controls.h`.

On the **Modeling** tab, select **Model Explorer**.

In the **Model Hierarchy** pane, expand `Simulink Root` and select `Base Workspace`. The **Contents** pane displays all the objects in the base workspace.

Select `VSSMODE` and then in the **Simulink.Parameter:VSSMODE** pane, click the **Code Generation** tab.

In the **HeaderFile** parameter, specify the name of the file as `variant_controls.h`.

Alternatively, you can change the file name programmatically, by using this command. For more information on `Simulink.CoderInfo`, see `Simulink.CoderInfo`.

```
Simulink.CoderInfo.CustomAttributes.HeaderFile = 'variant_controls.h';
```

Reduce Maintenance Effort by Creating Custom Storage Class

To change the name of the header file, you must change the configuration of each parameter object. You can use the Model Data Editor to perform batch editing, but when you add a new variant control variable (parameter object), you must remember to specify the name of the header file for that object. Also, the Model Data Editor shows the parameter objects used by only one model at a time.

Instead, you can create a custom storage class and specify the name of the header file only once: In the definition of the custom storage class.

Set your current folder to a writable location.

At the command prompt, copy the built-in `SimulinkDemos` package into your current folder as `myPackage`.

```
copyfile(fullfile(matlabroot,...
    'toolbox','simulink','simdemos','dataclasses','+SimulinkDemos'),...
    '+myPackage','f')
```

Navigate inside the `+myPackage` folder to the file `Parameter.m` and open the file.

Uncomment the `methods` section that defines the method `setupCoderInfo`. In the call to the function `useLocalCustomStorageClasses`, replace `'packageName'` with `'myPackage'`. When you finish, the section appears as follows:

```
  methods
    function setupCoderInfo(h)
      % Use custom storage classes from this package
      useLocalCustomStorageClasses(h, 'myPackage');
    end
  end % methods
```

Save and close the file.

Set your current folder to the folder that contains the package `myPackage`.

Open the Custom Storage Class Designer.

```
cscdesigner('myPackage')
```

Select the custom storage class `Define`.

Click **Copy**. A new custom storage class, `Define_1`, appears. Select this new custom storage class.

Set **Name** to `VariantControlVar`.

Set **Header file** to `Specify`. In the text box, enter `variant_controls.h`.

Click **Apply**, **Save**, and **OK**.

At the command prompt, replace the `Simulink.Parameter` object `VSSMODE` with `myPackage.Parameter` object. Apply the new custom storage class `VariantControlVar`.

```
MODE_A = myPackage.Parameter;
MODE_A.Value = 1;
MODE_A.DataType = 'int32';
MODE_A.CoderInfo.StorageClass = 'Custom';
MODE_A.CoderInfo.CustomStorageClass = 'VariantControlVar';

MODE_B = myPackage.Parameter;
MODE_B.Value = 1;
MODE_B.DataType = 'int32';
MODE_B.CoderInfo.StorageClass = 'Custom';
MODE_B.CoderInfo.CustomStorageClass = 'VariantControlVar';
```

Now, to indicate that a parameter object represents a variant control variable, you can apply the custom storage class `VariantControlVar`. To change the name of the header file, use the Custom Storage Class Designer.

See Also

Related Examples

- “Create Variant Controls Programmatically”
- “Working with Variant Choices”

Represent Variant Source and Sink Blocks in Generated Code

In this section...

"Represent Variant Source and Variant Sink Blocks in Simulink" on page 27-37
"Specify Conditions That Control Variant Choice Selection" on page 27-40
"Review the Generated Code" on page 27-41
"Generate Code with Zero Active Variant Controls" on page 27-43

You can use Variant Source and Variant Sink blocks to perceive multiple implementations of a model in a single, unified block diagram. Each implementation depends on conditions that you set for Variant Source and Variant Sink blocks. Simulink propagates these conditions to upstream and downstream blocks including root input and root output ports.

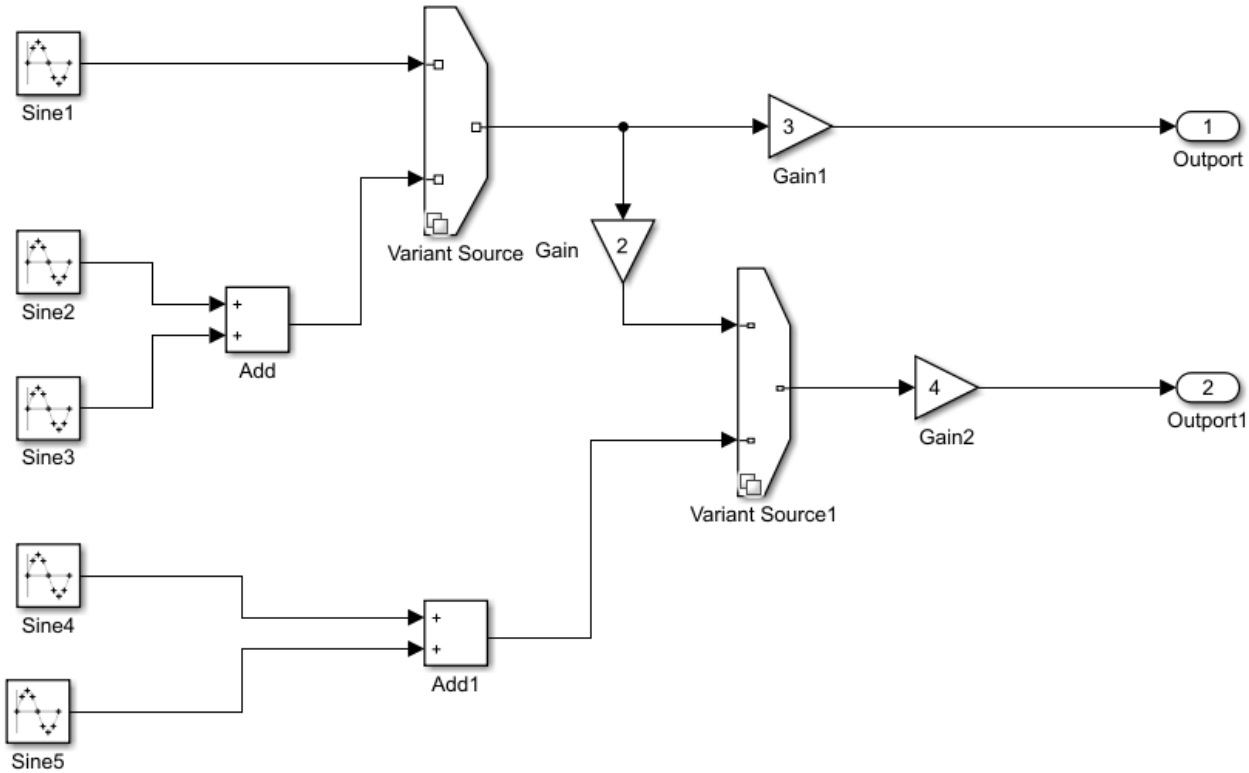
You can generate:

- Code from a Simulink model containing Variant Sink and Variant Source blocks.
- Code that contains preprocessor conditionals that control the activation of each variant choice.
- Preprocessor conditionals that allow for no active variant choice.

Represent Variant Source and Variant Sink Blocks in Simulink

This example shows how Variant Source blocks make model elements conditional.

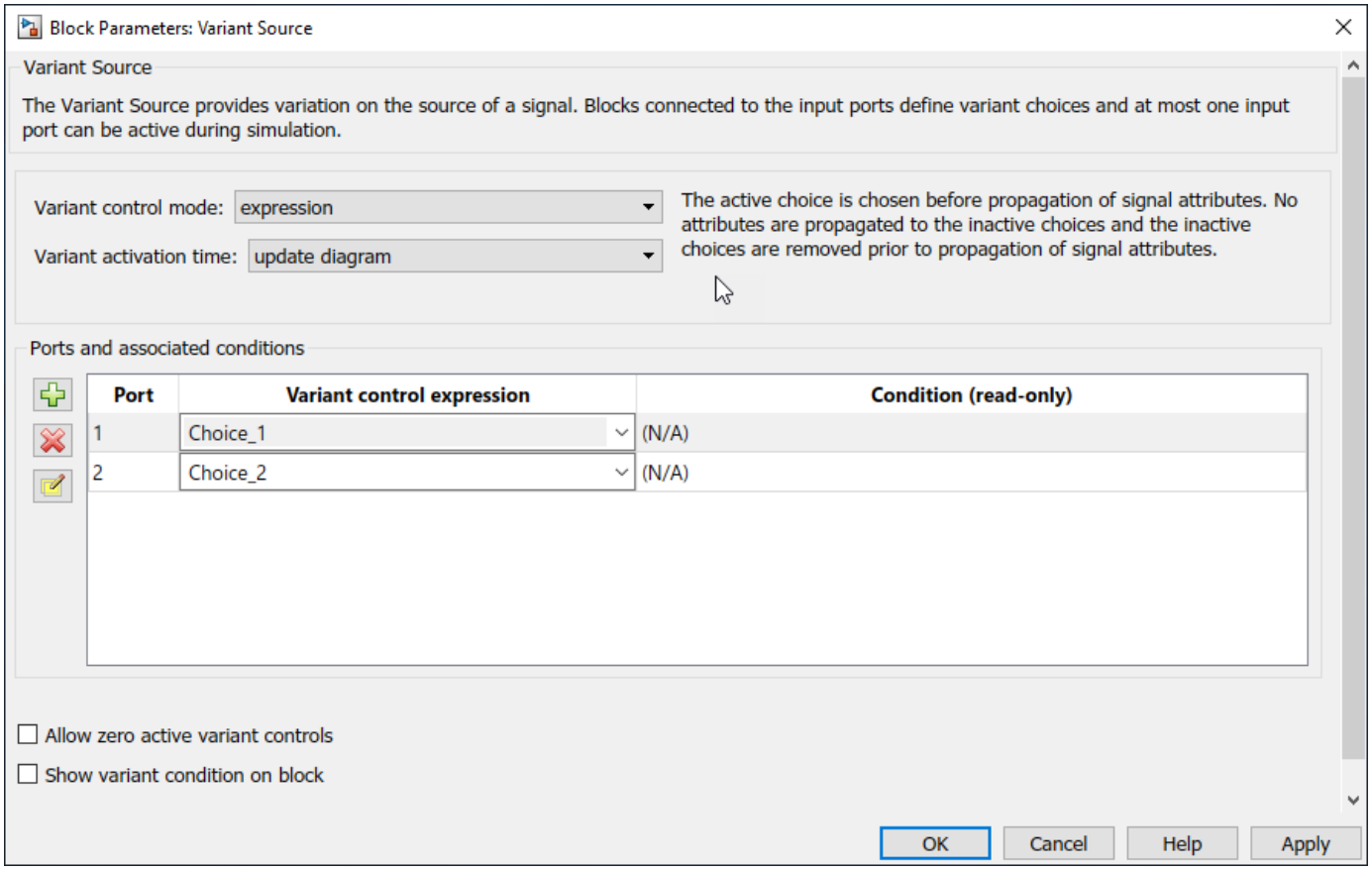
1. From the Simulink Block Library, add 1 Sine Wave Function block, two Add blocks, three Gain blocks, two Outports, and two Variant Source blocks into a new model.
2. Open the Sine Wave Function block. For the **Sine type** parameter, select `Sample based`. For the **Time (t)** parameter, select `Use simulation time`. For the **Sample time** parameter, insert a value of `0.2`.
3. Make four copies of the Sine Wave Function block.
4. Connect and name the blocks as shown.



5 Insert values of 2, 3, and 4 in the `Gain2`, `Gain3`, and `Gain4` blocks, respectively.

6 Give the model the name `inline_variants_example`.

7 Open the Block Parameters dialog box for `Variant Source`.



8. In the **Variant control** column, for Port 1, replace `Choice_1` with `V==1`. For Port 2, replace `Choice_2` with `V==2`.
9. Open the Block Parameters dialog box for `Variant Source1`.
10. In the **Variant control** column, replace `Choice_1` with `W==1`. For Port 2, replace `Choice_2` with `W==2`.
11. In the MATLAB Command Window, use these commands to define `V` and `W` as `Simulink.Parameter` objects.

```
V = Simulink.Parameter;
V.Value = 1;
V.DataType='int32';
V.CoderInfo.StorageClass = 'custom';
V.CoderInfo.CustomStorageClass = 'Define';
V.CoderInfo.CustomAttributes.HeaderFile='inline_importedmacro.h'

W = Simulink.Parameter;
W.Value = 2;
W.DataType='int32';
W.CoderInfo.StorageClass = 'custom';
W.CoderInfo.CustomStorageClass = 'Define';
W.CoderInfo.CustomAttributes.HeaderFile='inline_importedmacro.h'
```

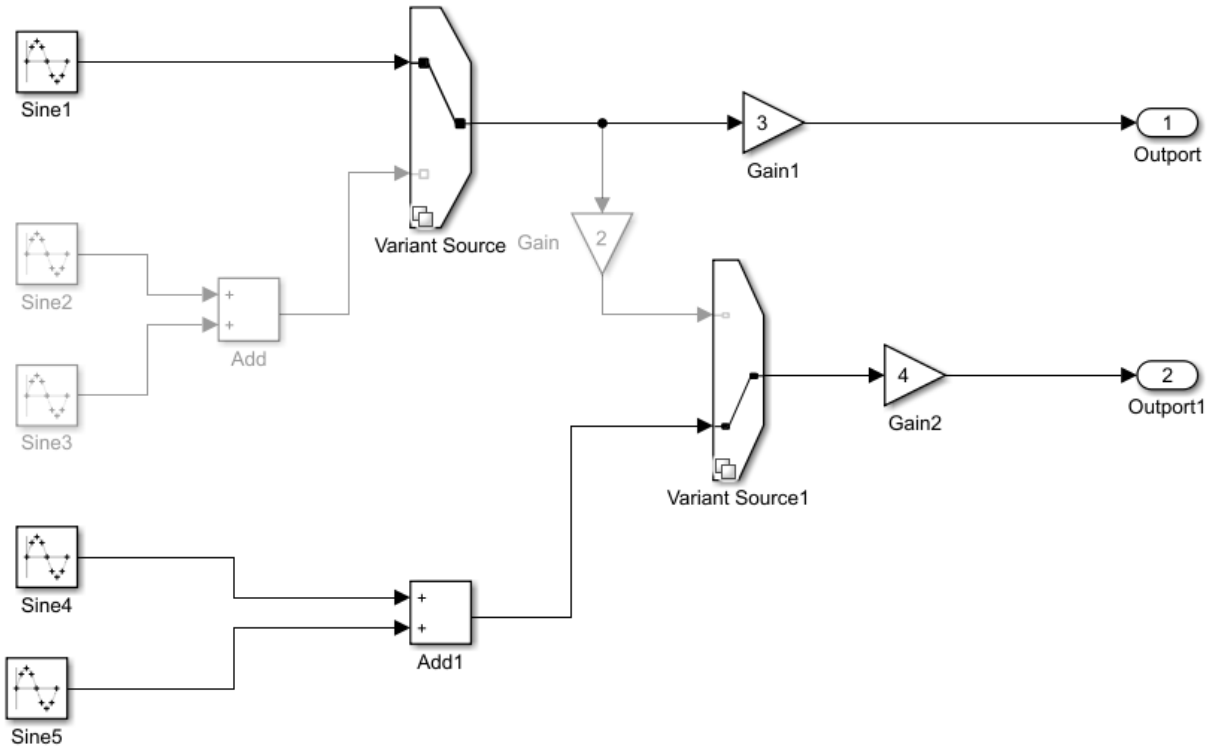
In this example, the variant control variables are `Simulink.Parameter` objects. For code generation, if you use `Simulink.Variant` objects to specify variant controls, use `Simulink.Parameter` objects or MATLAB variables to specify their conditions. .

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes:

- `Define` with header file specified
- `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- User-defined custom storage class that defines data as a macro in a specified header file

If you use scalar variant control variables to simulate the model, you can convert those variables into `Simulink.Parameter` objects. See “Convert Variant Control Variables into Simulink.Parameter Objects”.

12 Simulate the model.



Input port 1 is the active choice for `Variant Source` because the value of variant control variable `V` is `1`. Input port 2 is the active choice for `Variant Source1` because the value of variant control variable `W` is 2. The inactive choices are removed from execution, and their paths are grayed-out in the diagram.

Specify Conditions That Control Variant Choice Selection

You can generate code in which each variant choice is enclosed within C preprocessor conditionals `#if` and `#endif`. The compiler chooses the active variant at compile time and the preprocessor conditionals determine which sections of the code to execute.

1 In the **Modeling** tab of the Simulink toolstrip, click **Model Settings**.

2. On the **Code Generation** pane, and set **System target file** to `ert.tlc`.
3. On the **Solver** pane, set the **Type** parameter to `Fixed-step`.
4. In your model, open the block parameters dialog box for `Variant Source`.
5. Set the **Variant activation time** parameter to `code compile`. During an update diagram or simulation, when you set this parameter value, Simulink analyzes all variant choices. This analysis provides early validation of the code generation readiness of variant choices. During code generation, when you set this parameter value, the code generator generates preprocessor conditionals that control the activation of each variant choice.
6. Clear the **Allow zero active variant controls** parameter.
7. Open the Block Parameters dialog box for `Variant Source 1`. Repeat steps 5 through 7.
8. Build the model. When code generation is complete, the generated code is displayed in the Code view.

Review the Generated Code

1. In the Code view, select the `inline_variants_example.c` file.
2. In the `inline_variants_example.c` file, the call to the `inline_variants_example_step` function is conditionally compiled as shown:

```
/* Model step function */
void inline_variants_example_step(void)
{
  real_T rtb_VariantMerge_For_Variant_So;
  real_T rtb_VariantMerge_For_Variant__e;

  /* Sin: '<Root>/Sine2' incorporates:
   *  Sin: '<Root>/Sine3'
   *  Sum: '<Root>/Add'
   */
#if V == 2

  rtb_VariantMerge_For_Variant_So = sin((real_T)
    inline_variants_example_DW.counter * 2.0 * 3.1415926535897931 / 10.0) + sin
    ((real_T)inline_variants_example_DW.counter_g * 2.0 * 3.1415926535897931 /
     10.0);

#endif

  /* End of Sin: '<Root>/Sine2' */

  /* Sin: '<Root>/Sine4' incorporates:
   *  Sin: '<Root>/Sine5'
   *  Sum: '<Root>/Add1'
   */
#if W == 2

  rtb_VariantMerge_For_Variant__e = sin((real_T)
    inline_variants_example_DW.counter_c * 2.0 * 3.1415926535897931 / 10.0) +
    sin((real_T)inline_variants_example_DW.counter_i * 2.0 * 3.1415926535897931 /
        10.0);

#endif
```

```
  /* End of Sin: '<Root>/Sine4' */

  /* Sin: '<Root>/Sine1' */
#if V == 1

  rtb_VariantMerge_For_Variant_So = sin((real_T)
    inline_variants_example_DW.counter_m * 2.0 * 3.1415926535897931 / 10.0);

#endif

  /* End of Sin: '<Root>/Sine1' */

  /* Outport: '<Root>/Outport' incorporates:
   *  Gain: '<Root>/Gain1'
   */
  inline_variants_example_Y.Outport = 3.0 * rtb_VariantMerge_For_Variant_So;

  /* Gain: '<Root>/Gain' */
#if W == 1

  rtb_VariantMerge_For_Variant__e = 2.0 * rtb_VariantMerge_For_Variant_So;

#endif

  /* End of Gain: '<Root>/Gain' */

  /* Outport: '<Root>/Outport1' incorporates:
   *  Gain: '<Root>/Gain2'
   */
  inline_variants_example_Y.Outport1 = 4.0 * rtb_VariantMerge_For_Variant__e;

  /* Update for Sin: '<Root>/Sine2' incorporates:
   *  Sin: '<Root>/Sine3'
   */
#if V == 2

  inline_variants_example_DW.counter++;
  if (inline_variants_example_DW.counter == 10) {
    inline_variants_example_DW.counter = 0;
  }

  inline_variants_example_DW.counter_g++;
  if (inline_variants_example_DW.counter_g == 10) {
    inline_variants_example_DW.counter_g = 0;
  }

#endif

  /* End of Update for Sin: '<Root>/Sine2' */

  /* Update for Sin: '<Root>/Sine4' incorporates:
   *  Sin: '<Root>/Sine5'
   */
#if W == 2

  inline_variants_example_DW.counter_c++;
  if (inline_variants_example_DW.counter_c == 10) {
    inline_variants_example_DW.counter_c = 0;
```

```
    }

    inline_variants_example_DW.counter_i++;
    if (inline_variants_example_DW.counter_i == 10) {
      inline_variants_example_DW.counter_i = 0;
    }

  #endif

    /* End of Update for Sin: '<Root>/Sine4' */

    /* Update for Sin: '<Root>/Sine1' */
  #if V == 1

    inline_variants_example_DW.counter_m++;
    if (inline_variants_example_DW.counter_m == 10) {
      inline_variants_example_DW.counter_m = 0;
    }

  #endif

    /* End of Update for Sin: '<Root>/Sine1' */
  }
```

The variables `rtb_VariantMerge_For_Variant_So` and `rtb_VariantMerge_For_Variant_e` hold the input values to the Variant Source blocks. Notice that the code for these variables is conditional. The variables `inline_variants_example_Y.Outport` and `inline_variants_example_Y.Outport1` hold the output values of the Variant Source blocks. Notice that the code for these variables is not conditional.

Generate Code with Zero Active Variant Controls

You can generate code in which blocks connected to the input and the output of a Variant Source block are conditional.

1. For `Variant Source`, open the Block Parameters dialog box. Select the parameter **Allow zero active variant controls**.
2. For `Variant Source 1`, open the Block Parameters dialog box. Select the parameter **Allow zero active variant controls**.

When you select the **Allow zero active variant controls** parameter, you can generate code for a model containing Variant Source and Variant Sink blocks even when you specify a value for a variant control variable that does not allow for an active variant. Choosing a value for a variant control variable that does not allow for an active variant and not selecting the **Allow zero active variant controls** parameter, produces an error.

Generate code for `inline_variants_example`. Notice in the `inline_variants_example.c` file, that the code for the variables `inline_variants_example_Y.Outport1` and `inline_variants_example_Y.Outport2` is conditional.

```
/* Model step function */
void inline_variants_example_step(void)
{
 ...
#if V == 1 || V == 2
```

```
  inline_variants_example_Y.Outport = 3.0 * rtb_VariantMerge_For_Variant_So;

#endif


 ...
#if (V == 1 && W == 1) || (V == 2 && W == 1) || W == 2

  inline_variants_example_Y.Outport1 = 4.0 * rtb_VariantMerge_For_Variant__e;

#endif

 ...
```

See Also

Related Examples

- “Define and Configure Variant Sources and Sinks”
- “Variant Condition Propagation with Variant Sources and Sinks”
- “Introduction to Variant Controls”
- “Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model” (Simulink Check)

Configure Symbolic Dimensions for S-Function Blocks

To configure symbolic dimensions for S-function blocks, you can use the following C/C++ functions. You can configure S-functions to support forward propagation, backward propagation, or forward and backward propagation of symbolic dimensions during simulation.

Many of these functions return the variable `SymbDimsId`. A `SymbDimsId` is a unique integer value. This value corresponds to each symbolic dimension specification that you create or is the result of a mathematical operation that you perform with symbolic dimensions.

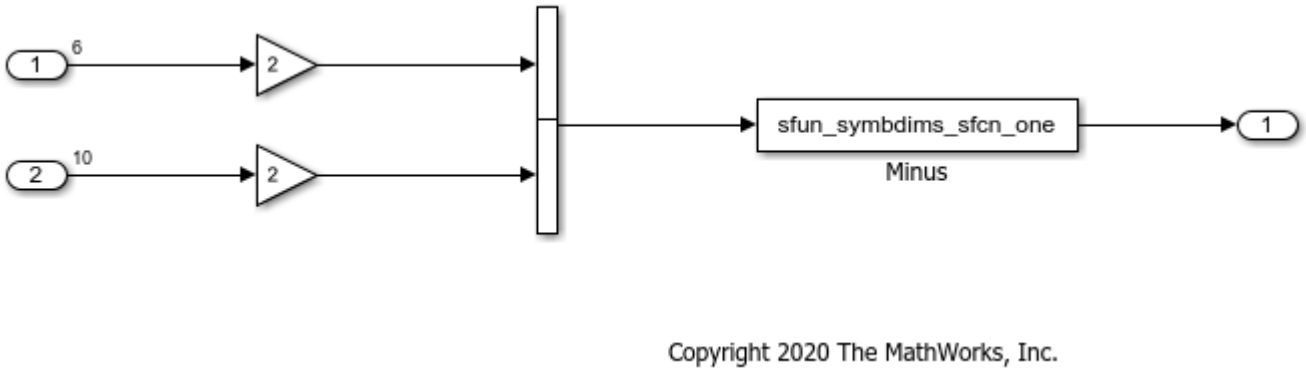
C/C++ S-Functions	Purpose
`ssSetSymbolicDimsSupport`	Specify whether or not an S-function supports symbolic dimensions.
`mdlSetInputPortSymbolicDimensions`	Specify the symbolic dimensions of an input port and how those dimensions propagate forward.
`mdlSetOutputPortSymbolicDimensions`	Specify the symbolic dimensions of an output port and how those dimensions propagate backward.
`ssRegisterSymbolicDimsExpr`	Create a `SymbDimsId` from an expression string (`aExpr`). The expression string must form a valid syntax in C.
`ssRegisterSymbolicDims`	Create a `SymbDimsId` from a vector of `SymbDimsId`s.
`ssRegisterSymbolicDimsString`	Create a `SymbDimsId` from an identifier string (`aString`).
`ssRegisterSymbolicDimsIntValue`	Create a `SymbDimsId` from an integer value (`aIntValue`)
`ssRegisterSymbolicDimsPlus`	Create a `SymbDimsId` by adding two symbolic dimensions.
`ssRegisterSymbolicDimsMinus`	Create a `SymbDimsId` by subtracting two symbolic dimensions.
`ssRegisterSymbolicDimsMultiply`	Create a `SymbDimsId` by multiplying two symbolic dimensions.
`ssRegisterSymbolicDimsDivide`	Create a `SymbDimsId` by dividing two symbolic dimensions.
`ssGetNumSymbolicDims`	Get the number of dimensions for a `SymbDimsId`.
`ssGetSymbolicDim`	Get a `SymbDimsId` from a vector of `SymbDimsId`s.
`ssSetInputPortSymbolicDimsId`	Set the precompiled `SymbDimsId` of an input port. You can call this function from inside the `mdlInitializeSizes` function.
`ssGetCompInputPortSymbolicDimsId`	Get the compiled `SymbDimsId` of an input port.

C/C++ S-Functions	Purpose
`ssSetCompInputPortSymbolicDimsId`	Set the compiled `SymbDimsId` of an input port.
`ssSetOutputPortSymbolicDimsId`	Set the precompiled `SymbDimsId` of an output port. You can call this function from inside the `mdlInitializeSizes` function.
`ssGetCompOutputPortSymbolicDimsId`	Get the compiled `SymbDimsId` of an output port.
`ssSetCompOutputPortSymbolicDimsId`	Set the compiled `SymbDimsId` of an output port.
`ssSetCompDWorkSymbolicDimsId`	Set the compiled `SymbDimsId` of an index of a block's data type work (`DWork`) vector.

Generate Code from an S-Function Block with Dimension Variants

The model `ex_msymbdims_sfcn_one` contains an S-function that subtracts the symbolic dimension `B` from a symbolic input dimension, `A + B`. The model supports forward propagation because the symbolic dimensions of the output ports are set. The model does not support backward propagation of symbolic dimensions because the compiled symbolic dimensions of the input ports are not set.

```
model='ex_msymbdims_sfcn_one';
open_system(model);
```



The `sfun_symbdims_sfcn_one.c` file contains this code.

```
type sfun_symbdims_sfcn_one.c
```

```
/* Copyright 2014 The MathWorks, Inc. */
#define S_FUNCTION_NAME  sfun_symbdims_sfcn_one
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#include <assert.h>

/* Function: mdlInitializeSizes ===============================================
```

```
 * Abstract:
 *   Setup sizes of the various vectors.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortDimensionInfo(S, 0, DYNAMIC_DIMENSION);
    ssSetInputPortRequiredContiguous(S, 0, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortDimensionInfo(S, 0, DYNAMIC_DIMENSION);
    ssSetNumSampleTimes(S, 1);

    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);

    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs( S, 0);

    ssAllowSignalsWithMoreThan2D(S);

    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);

    ssSetSymbolicDimsSupport(S, true);
}


/*If underspecified for propagation, say i/o are scalars.*/
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_DEFAULT_PORT_DIMENSION_INFO
static void mdlSetDefaultPortDimensionInfo(SimStruct *S)
{
    if( (ssGetOutputPortWidth(S, 0) == DYNAMICALLY_SIZED)
        && (ssGetInputPortWidth(S, 0) == DYNAMICALLY_SIZED) ) {
        ssSetOutputPortMatrixDimensions(S, 0, 1, 1 );
    }
}
#endif

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetInputPortSymbolicDimensions(SimStruct* S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
    // Set the compiled input symbolic dimension.
    ssSetCompInputPortSymbolicDimsId(S,  portIndex, symbDimsId);
    const SymbDimsId symbolId = ssRegisterSymbolicDimsString(S, "B");
    const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsMinus(S, symbDimsId, symbolId);
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, outputDimsId);
```

```
}
#endif

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetOutputPortSymbolicDimensions(SimStruct *S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
   // The input dimensions are not set, so this S-function only
    // supports forward propagation.
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, symbDimsId);
}
#endif

/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specify that we inherit our sample time from the driving block.
 *    However, we don't execute in minor steps.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}
/* Function: mdlOutputs =======================================================
 * Abstract:
 *    y=u';
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
}

/* Function: mdlTerminate =====================================================
 * Abstract:
 *    No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
    UNUSED_ARG(S); /* unused input argument */
}



#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

The Simulink.Parameter objects A and B contain these definitions.

```
type definitions.txt
```

```
A = Simulink.Parameter;
A.DataType = 'int32';
```

```
A.Value = 6;
A.Min=5;
A.Max=7;
A.CoderInfo.StorageClass = 'Custom';
A.CoderInfo.CustomStorageClass = 'ImportedDefine';
A.CoderInfo.CustomAttributes.HeaderFile = 'ImportedDefinitions.h';

B = Simulink.Parameter;
B.DataType = 'int32';
B.Value = 10;
B.Min=9;
B.Max=11;
B.CoderInfo.StorageClass = 'Custom';
B.CoderInfo.CustomStorageClass = 'ImportedDefine';
B.CoderInfo.CustomAttributes.HeaderFile = 'ImportedDefinitions.h';
```

The `ImportedDefinitions` header file contains this code.

```
type ImportedDefinitions.h
```

```
/* Copyright 2014 The MathWorks, Inc. */
#ifndef _IMPORTEDDEFINITIONS_H_
#define _IMPORTEDDEFINITIONS_H_
#define A 6
#define B 10
#endif
```

In the Configuration Paramters dialog box, the **Custom Code** pane contains the `#include "ImportedDefinitions.h"` statement.

The `sfun_symbdims_sfcn_one` S-function is inlined, so a TLC file with this code must be present on the path with the model and other supporting files.

```
type sfun_symbdims_sfcn_one.tlc
```

```
%implements "sfun_symbdims_sfcn_one" "C"


%function BlockInstanceSetup(block, system) void
%endfunction

%% Function: BlockOutputSignal ==================================================
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
%endfunction

%% Function: Outputs ============================================================
%%
%function Outputs(block, system) Output


  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %%
    %assign u  = LibBlockInputSignal( 0, "", lcv, idx)
    %assign y  = LibBlockOutputSignal(0, "", lcv, idx)
```

```
    %<y> = %<u>;

  %endroll

%endfunction %% Outputs
```

To generate a MEX function, at the command-line, enter this command:

```
mex sfun_symbdims_sfcn_one.c
```

```
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

Generate code for the model.

```
currentDir=pwd;
slbuild(model);
```

```
### Starting build procedure for: ex_msymbdims_sfcn_one
### Successful completion of build procedure for: ex_msymbdims_sfcn_one

Build Summary

Top model targets built:

Model                  Action                        Rebuild Reason
========================================================================================
ex_msymbdims_sfcn_one  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 8.5194s
```

View the generated code for the model. The `ex_msymbdims_sfcn_one_step` function contains this code. The S-function code is set inline and the symbolic dimensions A and B are in the generated code.

```
cfile=fullfile(currentDir, 'ex_msymbdims_sfcn_one_ert_rtw','ex_msymbdims_sfcn_one.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void ex_msymbdims_sfcn_one_step(void)
{
  int32_T i;

  /* Gain: '<Root>/Gain' incorporates:
   *  Concatenate: '<Root>/Matrix Concatenate'
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < A; i++) {
    ex_msymbdims_sfcn_one_B.MatrixConcatenate[i] = 2.0 *
      ex_msymbdims_sfcn_one_U.In1[i];
  }

  /* End of Gain: '<Root>/Gain' */
```

```
  /* Gain: '<Root>/Gain2' incorporates:
   *  Concatenate: '<Root>/Matrix Concatenate'
   *  Inport: '<Root>/In2'
   */
  for (i = 0; i < B; i++) {
    ex_msymbdims_sfcn_one_B.MatrixConcatenate[A + i] = 2.0 *
      ex_msymbdims_sfcn_one_U.In2[i];
  }

  /* End of Gain: '<Root>/Gain2' */

  /* S-Function (sfun_symbdims_sfcn_one): '<Root>/Minus' incorporates:
   *  Outport: '<Root>/Out1'
   */
  {
    int_T i1;
    const real_T *u0 = &ex_msymbdims_sfcn_one_B.MatrixConcatenate[0];
    real_T *y0 = &ex_msymbdims_sfcn_one_Y.Out1[0];
    for (i1=0; i1 < ((A + B)); i1++) {
      y0[i1] = u0[i1];
    }
  }
}
```

Close the model

```
bdclose(model);
```

S-Function That Supports Forward and Backward Propagation of Symbolic Dimensions

This S-function transposes two symbolic dimensions. It supports forward and backward propagation of symbolic dimensions because the compiled symbolic dimension of both the input and output ports are set.

```
static void mdlInitializeSizes(SimStruct *S)
{
     // Enable symbolic dimensions for the s-function.
     ssSetSymbolicDimsSupport(S, true);
}

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetInputPortSymbolicDimensions(SimStruct* S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);

    ssSetCompInputPortSymbolicDimsId(S,  portIndex, symbDimsId);

    assert(2U == ssGetNumSymbolicDims(S, symbDimsId));

    if (SL_INHERIT ==
        ssGetCompOutputPortSymbolicDimsId(S, portIndex)) {

        const SymbDimsId idVec[] = {
            ssGetSymbolicDim(S, symbDimsId, 1),
```

```
            ssGetSymbolicDim(S, symbDimsId, 0)};
        // Register the transposed dimensions.
       // Set the output symbolic dimension to the resulting id.
  const SymbDimsId outputDimsId =
            ssRegisterSymbolicDims(S, idVec, 2U);

        ssSetCompOutputPortSymbolicDimsId(S, portIndex,
           outputDimsId);
    }
}
#endif

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetOutputPortSymbolicDimensions(SimStruct *S,
    int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, symbDimsId);

    assert(2U == ssGetNumSymbolicDims(S, symbDimsId));

    if (SL_INHERIT ==
        ssGetCompInputPortSymbolicDimsId(S, portIndex)) {

        const SymbDimsId idVec[] = {
            ssGetSymbolicDim(S, symbDimsId, 1),
            ssGetSymbolicDim(S, symbDimsId, 0)};
        const SymbDimsId inputDimsId =
            ssRegisterSymbolicDims(S, idVec, 2U);
        // Register the transposed dimensions.
       // Set the input symbolic dimension to the resulting id.
        ssSetCompInputPortSymbolicDimsId(S, portIndex, inputDimsId);
    }
}
#endif
```

See Also

Related Examples

- “Implement Symbolic Dimensions for Array Sizes in Generated Code” on page 27-2

Options to Represent Variant Parameters in Generated Code

You can generate the code only for active values or for both active and inactive values of variant parameters in Simulink. You can also represent the values of variant parameters as inline variables or as tunable variables in the generated code.

Note For variant parameters associated with the `startup` activation time, you can generate code for both active and inactive choices by installing Simulink Coder or Embedded Coder.

For variant parameters associated with the `code compile` activation time:

- You must install Simulink Coder to generate code for only active values of variant parameters. Generating code using Simulink Coder is supported only if values of a variant parameter have same dimension.
- You must install Embedded Coder to generate code for both active and inactive values of variant parameters. Generating code using Embedded Coder is supported for variant parameters values having same and different dimensions.

Overview of Variant Parameters

Variant parameters can have multiple values. Each value of the variant parameter is associated with a variant condition expression. During simulation, the value of a variant parameter associated with the condition that evaluates to `true` is the active value of the variant parameter. The value associated with the condition that evaluates to `false` is the inactive value of the variant parameter. When you generate code, you can choose to include only the active value or both the active and inactive values in the generated code. You can also choose to represent the values as inline or as tunable variables in the generated code. For more information on variant parameters, see “Use Variant Parameters to Reuse Block Parameters with Different Values”.

Choice of Values in Generated Code

Simulink supports generating code only for active values or for both active and inactive values of variant parameters. Values included in the code are based on the variant activation time you specify for the variant parameters. For more information, see “Activate Variant During Different Stages of Simulation and Code Generation Workflow”.

When you specify the variant activation time as:

- `'update diagram'` –– Simulink does not analyze the values of variant parameters for incompatibilities in signal attributes. It generates code only for the active values of variant parameters.
- `'update diagram analyze all choices'` –– Simulink analyzes both active and inactive values of variant parameters for incompatibilities in signal attributes, however it generates code only for the active values of variant parameters.
- `'code compile'` –– Simulink analyzes both active and inactive values of variant parameters for incompatibilities in signal attributes and it also generates code for both active and inactive values of the variant parameters. The values are enclosed in C preprocessor conditional statements `#if` and `#endif` that are conditionally compiled when you compile the generated code.

- `'startup'` ––Simulink analyzes both active and inactive values of variant parameters for incompatibilities in signal attributes and it also generates code for both active and inactive values of the variant parameters. The values are enclosed in regular `if` conditions inside the `model_initialize` function of the generated code that are conditionally evaluated when you run the executable that is built from the code.

Types of Variables in Generated Code

Simulink supports representing the values of variant parameters as inline variables or as tunable variables in the generated code. The representation of values in the code are based on the storage class you specify for the variant parameters.

- Inline variable: If you specify the storage class of a variant parameter as `“Auto”`, which is a default storage class, and the activation time as `update diagram` or `update diagram analyze all choices`, then the variant parameter is inlined to the literal numeric value of the parameter in the generated code. If you specify the storage class of a variant parameter as `Auto` and the activation time as `code compile`, then the variant parameter is inlined as macros in the generated code. Inlined variables reduce global RAM usage and increase efficiency of the generated code. The code does not allocate memory to represent numeric block parameters, such as the **Gain** parameter of a Gain block. Instead, the code is inlined to the literal numeric value of the parameter.
- Tunable: If you specify the storage class of a variant parameter as anything other than `Auto`, then the variant parameter is represented using symbolic names in the generated code. For a list of built-in storage classes supported for different activation times, see “Storage Classes for Different Variant Activation Times” on page 27-54.

Storage Classes for Different Variant Activation Times

When you specify a condition expression to determine the active choice, each variant control variable of the expression has a storage class associated with it. Use storage classes to control the appearance and placement of the variant control variables in the generated code. Storage classes also prevent the optimizations such as elimination of the storage for variant control variables in the generated code. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

The variables used in the variant condition expressions must satisfy these criteria to generate code with Simulink Coder or Embedded Coder.

- The variant control variables in variant condition expressions must have these storage classes.
  - This table lists the built-in storage classes supported for different activation times to generate code using Simulink Coder and Embedded Coder.

Built-in Storage Class	Variant activation time	
	`code compile`	`startup`
“ExportedGlobal”	x	✓
“Model Default”	x	✓
“ImportedExtern, ImportedExternPointer”	x	✓

Built-in Storage Class	Variant activation time	
“Auto”	x	x

- This table lists the custom storage classes supported for different activation times to generate code using Embedded Coder.

Custom Storage Class	Variant activation time	
	`code compile`	`startup`
“GetSet” on page 36-81	x	✓
“Define, ImportedDefine” on page 36-80	✓	x
“CompilerFlag” on page 36-79	✓	x
Const and ConstVolatile. See, “Const, Volatile, and ConstVolatile” on page 36-79	x	x
“Bitfield” on page 36-79	x	✓
“Struct” on page 36-83	x	✓
“ExportToFile” on page 36-80	x	✓
“ImportFromFile” on page 36-82	x	✓
Volatile. See, “Const, Volatile, and ConstVolatile” on page 36-79	x	✓

- If the activation time of a variant control variable is set to `startup`, then specifying its storage class to any of these is not supported:
  - Storage classes such as `Define` that generates macros in the code.
  - Storage classes such as `Const` that generates constant qualified variables in the code.
- If a variant parameter is part of a variant parameter bank (a `Simulink.VariantBank` object) and if the activation time of the corresponding variant control variable is set to `startup`, specifying storage class using the `Specification` property of the `Simulink.VariantVariable` object is not supported. Use the `Simulink.VariantBankCoderInfo` class instead.
- If the activation time of a variant control variable is set to `code compile` and if the values have different dimensions, then specifying its storage class such as `Define` that generates macros in the code is not supported.

Symbolic Dimensions in Generated Code

If a variant parameter has values with multiple dimensions, Simulink generates symbols to represent dimensions in the generated code. Representing dimensions as symbols enable the code to flexibly switch between dimension values for the given active choice without regenerating code every time you change the value of the variant control variable. For more information on symbolic dimensions in

variant parameters, see “Compile Code Conditionally for All Values of Variant Parameters with Same and Different Dimensions”.

Group Variant Parameter Values in Generated Code

With Embedded Coder, you can use variant parameter banks to group variant parameters that have the same set of variant conditions into a structure in the generated code. The code generator supports variant parameter banks only for variant parameters with variant activation time set to `startup`.

The generated code contains an array of the variant parameter bank structure type. The code generator groups all choice values of the variant parameters based on variant conditions and each set of choice values becomes an element of the array. The code uses a pointer variable to access the active set of values from the array. Parameter bank switching using a pointer variable avoids copying the choice values into the main program memory and improves the efficiency and readability of the generated code. See `Simulink.VariantBank`.

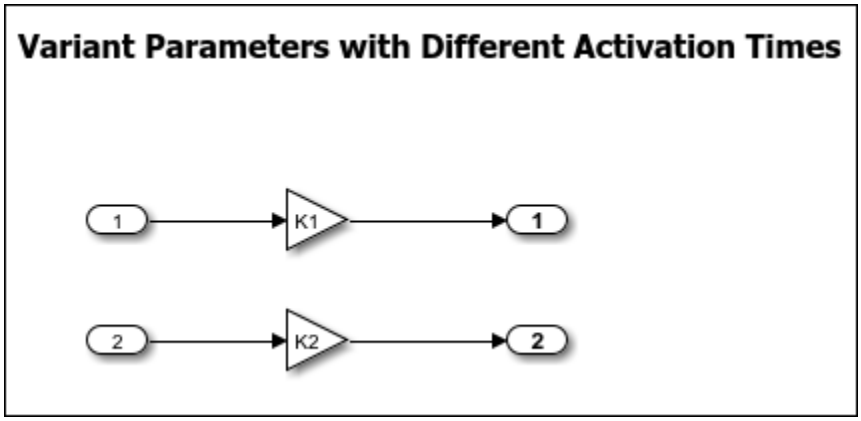
Generate C Code for Variant Parameters

This example shows how to generate C code for variant parameters.

Explore the Model

Consider the `slexVariantParameters` model. To open the model, use the following command.

```
openExample('simulink_variants/CreateASimpleVariantParameterModelExample');
```



The **Gain** parameters of the Gain1 and the Gain2 blocks are variant parameters with their values set to `K1` and `K2`, respectively. The variable `K1` has two values: `3.5` and `8.5`. The variable `K2` has two values: `4.5` and `9.5`.

Simulink chooses active values of `K1` and `K2` based on the value of the variant control variable `V`. If `V==1` evaluates to `true`, then the value of `K1` is set to `3.5`, and the value of `K2` is set to `8.5`. If `V==2` evaluates to `true`, then the value of `K1` is set to `4.5`, and the value of `K2` is set to `9.5`.

In this example, the variant activation time is set to `update diagram`, and the storage class is set as `Auto`. When you generate a code from this model, the code is generated only for the active values of `K1` and `K2`. In the generated code, the values are inlined to the literal value of the parameters.

Generate Inline and Tunable Code for Variant Parameters using Embedded Coder

Before you generate code from the model, you must have write permission in your current folder.

1. Prepare the model for code generation by specifying code generation settings in the Configuration Parameters dialog box. Choose the solver and code generation target, and check the model configuration for execution efficiency. For more details on each of these steps, see "Generate Code Using Embedded Coder" on page 53-2.
2. In the **Apps** gallery of the model toolstrip, click **Simulink Coder**.
3. On the **C Code** tab, click **Build**.

   The code generator creates the folder `slexVariantParameters_ert_rtw` in your current working folder and places source code files in that folder. The generated code is in two primary files: `slexVariantParameters.c` and `slexParameters.h`. The file `slexVariantParameters.h` contains the value of the variant control variable, and the file `slexVariantParameters.c` contains the values of the variant parameters.
4. In the **C Code** tab, select **Open Report**.
5. Select the `slexVariantParameters.c` file from the **Generated Code** pane of the report. The code is generated only for the active values of the `K1` and `K2`. In the generated code, the values are inlined to the literal value of the parameters.

   ```
   slexVariantParameters_Y.Out1 = 3.5 * slexVariantParameters_U.In1;
   slexVariantParameters_Y.Out2 = 4.5 * slexVariantParameters_U.In2;
   ```
6. In the model, change the activation time to `code compile` and the storage class to `ExportedGlobal`, then generate the code again. Observe the change in the style of the generated code.

   The code is generated only for all the active and inactive values of `K1` and `K2`. In the generated code, the values are represented using symbolic names.

   This table shows the difference in the style of the generated code for storage classes `Auto` and `ExportedGlobal` with different activation times.

Activation time	Storage class	
	`Auto` (default)	`ExportedGlobal`

`update diagram`	The code is generated only for the active values of the variant parameters. In the generated code, the values are inlined to the literal value of the parameters.
	
	Consider this snippet of the generated code. `4.5` and `3.5` are inline active values of the variant parameters **Gain1** and **Gain2**.
	
	`void slexVariantParameters_step(void)`
	`{`
	`slexVariantParameters_Y.Out1 = 3.5 * slexVariantParameters_U.In1;`
	`slexVariantParameters_Y.Out2 = 4.5 * slexVariantParameters_U.In2;`
	`}`
		The code is generated only for the active values of variant parameters. In the generated code, the values are represented using symbolic names.
		
		Consider this snippet of the generated code. Here, `K1` and `K2` are symbolic names for active values of variant parameters **Gain1** and **Gain2**.
		
		`real_T K1 = 3.5;`
		`real_T K2 = 4.5;`
		`…`
		`…`
		
		`void slexVariantParameters_step(void)`
		`{`
		`slexVariantParameters_Y.Out1 = K1 * slexVarian`
		`slexVariantParameters_Y.Out2 = K2 * slexVarian`
		`}`

`update diagram analyze all choices`	The generated code is same as that of `update diagram`. The only difference is that the active and the inactive values of variant parameters are analyzed for consistency of signal attributes such as dimension and data type across the model.	The generated code is same as that of `update diagram`. The only difference is that the active and the inactive values of variant parameters are analyzed for consistency of signal attributes such as dimension and data type across the model.

`code compile`	The code is generated for all the active and inactive values of the variant parameters. All of the values are analyzed for consistency of signal attributes such as dimension and data type across the model. In the generated code, the values are inlined as macros and are enclosed in preprocessor conditionals `#if` and `#elif`.
	
	Consider this snippet of the generated code. The active and inactive values are enclosed in C preprocessor conditional statements `#if` and `#elif`. `rtCP_Gain1_K1` and `rtCP_Gain_K2` are the macros for variant parameters **Gain1** and **Gain2**.
	
	`#if V == 1`
	`#define rtCP_Gain1_K1 (3.5)`
	`#elif V == 2`
	`#define rtCP_Gain1_K1 (8.5)`
	`#endif`
	
	`#if V == 1`
	`#define rtCP_Gain_K2 (4.5)`
	`#elif V == 2`
	`#define rtCP_Gain_K2 (9.5)`
	`#endif`
	
	`void slexVariantParameters_step(void)`
	`{`
	`slexVariantParameters_Y.Out1 = rtCP_Gain_K1 * slexVariantParameters_U.In1;`
	`slexVariantParameters_Y.Out2 = rtCP_Gain1_K2 * slexVariantParameters_U.In2;`
	`}`
		The code is generated for all the active and inactive values of variant parameters. All of the values are analyzed for consistency of signal attributes such as dimension and data type across the model. In the generated code, the values are represented using symbolic names and are enclosed in preprocessor conditionals `#if` and `#elif`.
		
		Consider this snippet of the generated code. The active and inactive values are enclosed in C preprocessor conditional statements `#if` and `#elif`. Here, `K1` and `K2` are symbolic names for variant parameters **Gain1** and **Gain2**.
		
		`#if V == 1`
		`real_T K1 = 3.5;`
		`real_T K2 = 4.5;`
		
		`#elif V == 2`
		`real_T K1 = 8.5;`
		`real_T K2 = 9.5;`
		`#endif`
		
		`void slexVariantParameters_step(void)`
		`{`
		`slexVariantParameters_Y.Out1 = K1 * slexVarian`
		`slexVariantParameters_Y.Out2 = K2 * slexVarian`
		`)`

`startup`	The code is generated for all the active and inactive values of the variant parameters. All of the values are analyzed for consistency of signal attributes such as dimension and data type across the model. In the generated code, the values are inlined as macros and are enclosed in regular `if` conditions.
	
	Consider this snippet of the generated code. The active and inactive values are enclosed in regular `if` condition statements inside the `model_initialize` function. The `model_initialize` function also contains an assertion statement to check that at least one variant value is active while running the executable. `slexVariantParameters_P.K1` and `slexVariantParameters_P.K2` are the members of a structure for variant parameters **Gain1** and **Gain2**.
	
	`static void slexVaria_startupVariantChecker(void)`
	`{`
	`  /* startup variant condition checks */`
	`  utAssert((V == 1) + (V == 2) == 1);`
	`}`
	
	`void slexVariantParameters_step(void)`
	`{`
	`slexVariantParameters_Y.Out1 = slexVariantParameters_P.K1 * ...`
	`slexVariantParameters_U.In1;`
	`slexVariantParameters_Y.Out2 = slexVariantParameters_P.K2 * ...`
	`slexVariantParameters_U.In2;`
	`}`
	
	`void slexVariantParameters_initialize(void)`
	`{`
	`if (V == 1) {`
	`    slexVariantParameters_P.K1 = 3.5;`
	`    slexVariantParameters_P.K2 = 4.5;`
	`  } else if (V == 2) {`
	`    slexVariantParameters_P.K1 = 8.5;`
	`    slexVariantParameters_P.K2 = 9.5;`
	`  }`
	`  slexVaria_startupVariantChecker();`
	`}`
		The code is generated for all the active and inactive values of variant parameters. All of the values are analyzed for consistency of signal attributes such as dimension and data type across the model. In the generated code, the values are represented using symbolic names and are enclosed in regular `if` conditions.
		
		Consider this snippet of the generated code. The active and inactive values are enclosed in regular `if` condition statements inside the `model_initialize` function. The `model_initialize` function contains an assertion statement to check that at least one variant value is active. Here, `K1` and `K2` are symbolic names for variant parameters **Gain1** and **Gain2**.
		
		`static void slexVaria_startupVariantChecker(vo`
		`{`
		`  /* startup variant condition checks */`
		`  utAssert((V == 1) + (V == 2) == 1);`
		`}`
		
		`void slexVariantParameters_step(void)`
		`{`
		`slexVariantParameters_Y.Out1 = K1 * slexVarian`
		`slexVariantParameters_Y.Out2 = K2 * slexVarian`
		`}`
		
		`void slexVariantParameters_initialize(void)`
		`{`
		`if (V == 1) {`
		`    K1 = 3.5;`
		`    K2 = 4.5;`
		`  } else if (V == 2) {`
		`    K1 = 8.5;`
		`    K2 = 9.5;`
		`  }`
		`  slexVaria_startupVariantChecker();`
		`}`

See Also

“Use Variant Parameters to Reuse Block Parameters with Different Values” | “Choose Storage Class for Controlling Data Representation in Generated Code”

Related Examples

- “Change Active Values and Activation Time of Variant Parameters”
- “Compile Code Conditionally for All Values of Variant Parameters with Same and Different Dimensions”
- “Run Executable for Different Variant Parameter Values Without Recompiling Code”

Generate Code for Variant Subsystem with Child Subsystems of Different Output Signal Dimensions

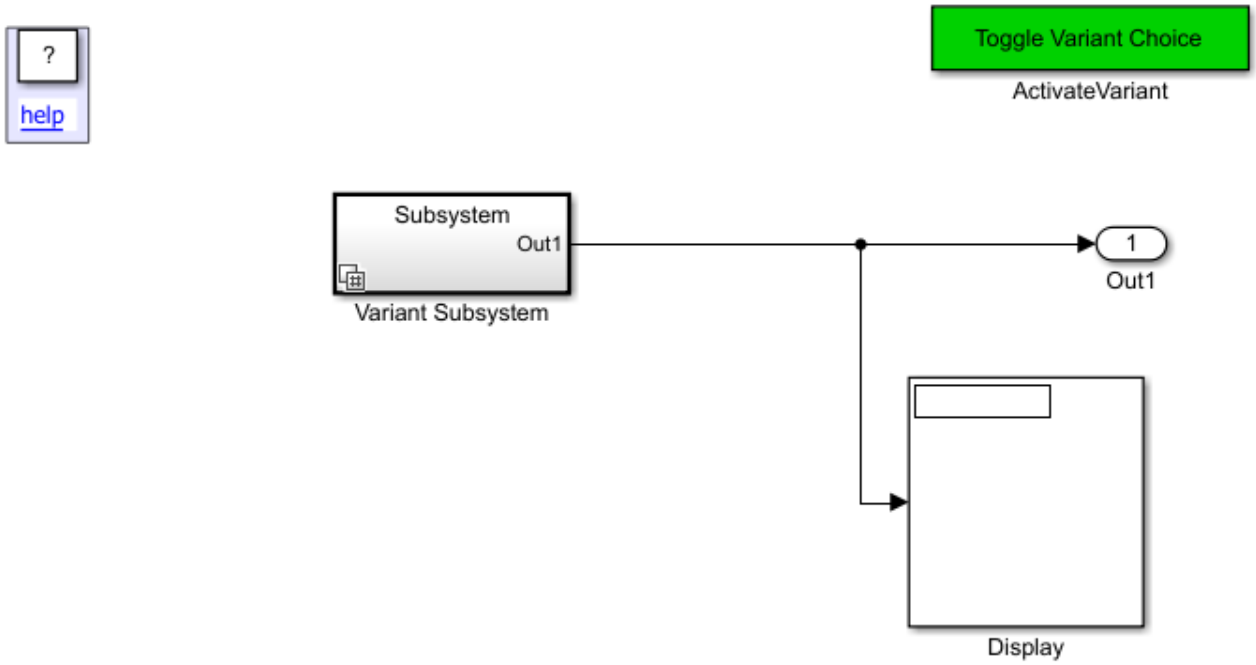
In this section...

"Example Model" on page 27-62
"Simulate Model" on page 27-62
"Generate Code" on page 27-63

This example shows how to use symbolic dimensions to generate code with preprocessor conditionals for a variant subsystem consisting of child subsystems of different output signal dimensions. The value of the variant control variable determines the active variant choice and the output signal dimensions. By changing the value of the variant control variable, you change the active variant and the output signal dimensions in the generated code.

Example Model

The model `slexVariantSymbolicDims` contains a Variant Subsystem consisting of the child subsystems `Subsystem` and `Subsystem1`. When the variant control variable `Var` has a value of `1`, `Subsystem` is the active variant. When `Var` has a value of `2`, `Subsystem1` is the active variant.



Simulate Model

To generate code with preprocessor conditionals, the output signal dimensions of the child subsystems must be the same during simulation. In this example, double-clicking the subsystem `Activate Variant Choice` changes the active variant and the output signal dimension. When `Var`

equals 1, the output signal dimension of each child subsystem is 5. When Var equals 2, the output signal dimension of each child subsystem is 6.

1. Open the example model slexVariantSymbolicDims.
2. On the **Debug** tab, select **Information Overlays** > **Signal Dimensions**.
3. Open the Variant Subsystem Block Parameters dialog box. The **Variant activation time** parameter is set to code compile.
4. Open Subsystem. In the Constant Block Parameters dialog box, the **Constant value** parameter is P1.
5. Open Subsystem1. In the Constant Block Parameters dialog box, the **Constant value** parameter is P2.
6. Open the base workspace. The Simulink.Parameters P1 and P2 are arrays with dimensions '[1,A]'. The Simulink.Parameter A has a value of 5. Var has a value of 1.
7. Simulate the model. Subsystem is the active variant with an output signal dimension of 5.
8. Double-click the masked subsystem ActivateVariant.
9. In the base workspace, Var has a value of 2. P1 and P2 have a dimension of 6. A has a value of 6.
10. Simulate the model. Subsystem1 is the active variant with an output signal dimension of 6.

In the base workspace, A has a **Storage class** of ImportedDefine(Custom). To use a Simulink.Parameter object for dimension specification, it must have one of these storage classes:

- Define or ImportedDefine with header file specified
- CompilerFlag
- User-defined custom storage class that defines data as a macro in a specified header file

In the base workspace, P1 and P2 have a storage class of ImportedExtern. A Simulink.Parameter object that uses a Simulink.Parameter for symbolic dimension specification must have a storage class of either ImportedExtern or ImportedExternPointer, or a storage class with the **Data initialization** property set to None.

Generate Code

1. Open the header file slexVariantSymbolicDims_variant_defines.h. The definition of A is conditional upon the value of Var.

```
/* Copyright 2016 The MathWorks, Inc. */
// To select variant choice during compile, define Var at compile time,

#ifndef Var
#define Var 1
#endif

#if Var == 1
#define A 5
#elif Var == 2
#define A 6
#else
#error "Variant control variable, Var, must be defined as 1 or 2"
#endif
```

2. Generate code.

3 Open the `slexVariantSymbolicDims.h` file. The output dimension size is `A`.

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  int32_T Out1[A];                      /* '<Root>/Out1' */
} ExternalOutputs_slexVariantSymb;
```

4 Open the `slexVariantSymbolicDims.c` file. If `Var` equals `1`, `P1` has five values. If `Var` equals `2`, `P2` has six values. In the Configuration Parameters dialog box, on the **Code Generation** > **Custom Code** pane, the **Additional code** parameter contains this code.

```
/* user code (top of source file) */
#if Var == 1

int32_T P1[] = { 5, 5, 5, 5, 5 };

#elif Var == 2

int32_T P2[] = { 6, 6, 6, 6, 6, 6 };

#endif
```

Preprocessor conditionals control the size of `A` and which array, `P1` or `P2`, is active in the generated code. By changing the value of `Var`, you can change the size of `A` and the active array.

See Also

Related Examples

- “Implement Symbolic Dimensions for Array Sizes in Generated Code” on page 27-2
- “Represent Subsystem and Variant Models in Generated Code” on page 27-22

Use Variant Subsystem to Generate Code That Uses C Preprocessor Conditionals

This example shows how to use Simulink® variant subsystems to generate C preprocessor conditionals that control which child subsystem of the variant subsystem is active in the generated code produced by the Simulink Coder™.

Overview of Variant Subsystems

A Variant Subsystem block contains two or more child subsystems where one child is active during model execution. The active child subsystem is referred to as the *active variant*. You can programmatically switch the active variant of the Variant Subsystem block by changing values of variables in the base workspace, or by manually overriding variant selection using the Variant Subsystem block dialog. The *active variant* is programmatically wired to the Inport and Outport blocks of the Variant Subsystem by Simulink during model compilation.

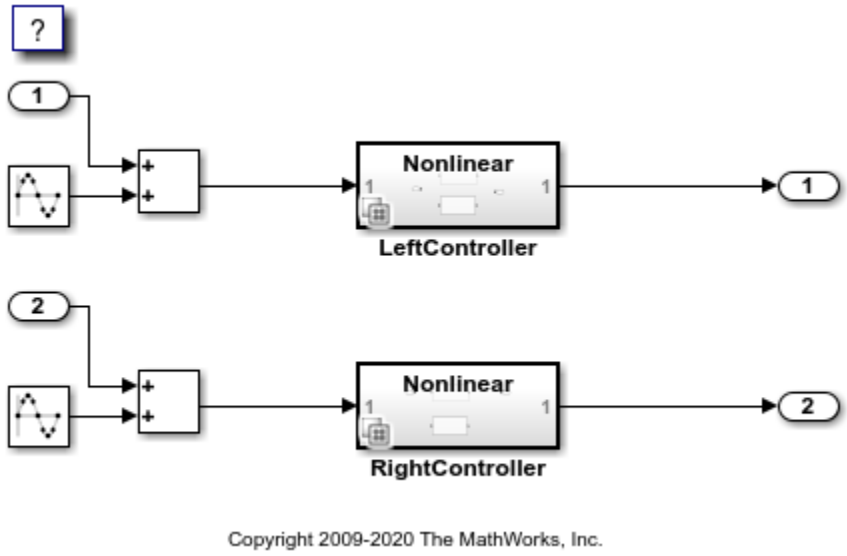
To programmatically control variant selection, a `Simulink.Variant` object is associated with each child subsystem in the Variant Subsystem block dialog. `Simulink.Variant` objects are created in the MATLAB® base workspace. These objects have a property named `Condition`, an expression, which evaluates to a Boolean value and is used to determine the active variant child subsystem.

By default the generated code contains only the active variant. Alternatively, you can parameterize the selection of the active variant, and make it dependent on the values of variables and objects in the base MATLAB workspace. When you generate code, you can generate code for all variants and defer the choice of active variant until it is time to compile that code.

Specifying Variants for a Subsystem Block

Opening the example model `rtwdemo_preprocessor_subsys` will run the **PostLoadFcn** defined in the "File: ModelProperties: Callbacks" dialog. This will populate the base workspace with the variables for the Variant Subsystem blocks.

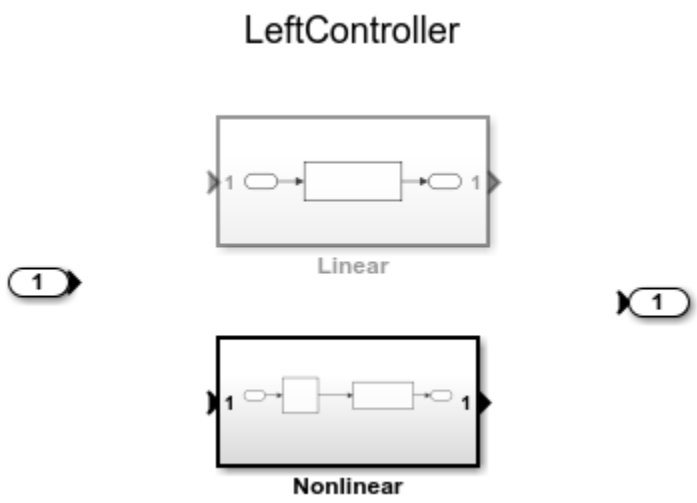
```
open_system('rtwdemo_preprocessor_subsys')
```



The LeftController variant subsystem contains two child subsystems: Linear and Nonlinear. The LeftController/Linear child subsystem executes when the Simulink.Variant object LINEAR evaluates to `true`, and the LeftController/Nonlinear child subsystem executes when the Simulink.Variant object NONLINEAR evaluates to `true`.

Simulink.Variant objects are specified for the LeftController subsystem by right-clicking the LeftController subsystem and selecting `Subsystem Parameters`, which will open the LeftController subsystem block dialog.

```
open_system('rtwdemo_preprocessor_subsys/LeftController');
```



The LeftController subsystem block dialog creates an association between the Linear and Nonlinear subsystems with the two Simulink.Variant objects, `LINEAR` and `NONLINEAR`, that exist in the base workspace. These objects have a property named `Condition`, an expression, which evaluates to a Boolean value and determines the active variant child subsystem (Linear or Nonlinear). The condition is also shown in the subsystem block dialog. In this example, the conditions of `LINEAR` and `NONLINEAR` are 'VSSMODE == 0' and 'VSSMODE == 1', respectively.

In this example, the Simulink.Variant objects are created in the base workspace.

```
LINEAR = Simulink.Variant;
LINEAR.Condition = 'VSSMODE==0';
NONLINEAR = Simulink.Variant;
NONLINEAR.Condition = 'VSSMODE==1';
```

Specifying a Variant Control Variable

Variant objects allow you to reuse arbitrarily complex conditions throughout a model. Multiple Variant Subsystem blocks can use the same Simulink.Variant objects, allowing you to toggle the activation of choices as a set. You can toggle the set prior to simulation by changing the value of `VSSMODE` in the MATLAB environment or when compiling the generated code, as explained in the next section. In this example, LeftController and RightController reference the same variant objects, so that you can toggle them simultaneously.

The nonlinear controller subsystems implement hysteresis, while the linear controller subsystems act as simple low-pass filters. Open the subsystem for the left channel. The subsystems for the right channel are similar.

The generated code accesses the variant control variable `VSSMODE` as a user-defined macro. In this example, `rtwdemo_importedmacros.h` supplies `VSSMODE`. Within the MATLAB environment, you specify `VSSMODE` using a `Simulink.Parameter` object. Its value will be ignored when generating code including preprocessor conditionals. However, the value is used for simulation. The legacy header file specifies the value of the macro to be used when compiling the generated code, which ultimately activates one of the two specified variants in the embedded executable.

Variant control variables can be defined as `Simulink.Parameter` objects with one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- User-defined custom storage class that defines data as a macro in a specified header file

```
VSSMODE = Simulink.Parameter;
VSSMODE.Value = 1;
VSSMODE.DataType = 'int32';
VSSMODE.CoderInfo.StorageClass = 'Custom';
VSSMODE.CoderInfo.CustomStorageClass = 'ImportedDefine';
VSSMODE.CoderInfo.CustomAttributes.HeaderFile = 'rtwdemo_importedmacros.h';
```

Simulating the Model with Different Variants

Because you set the value of `VSSMODE` to 1, the model uses the nonlinear controllers during simulation.

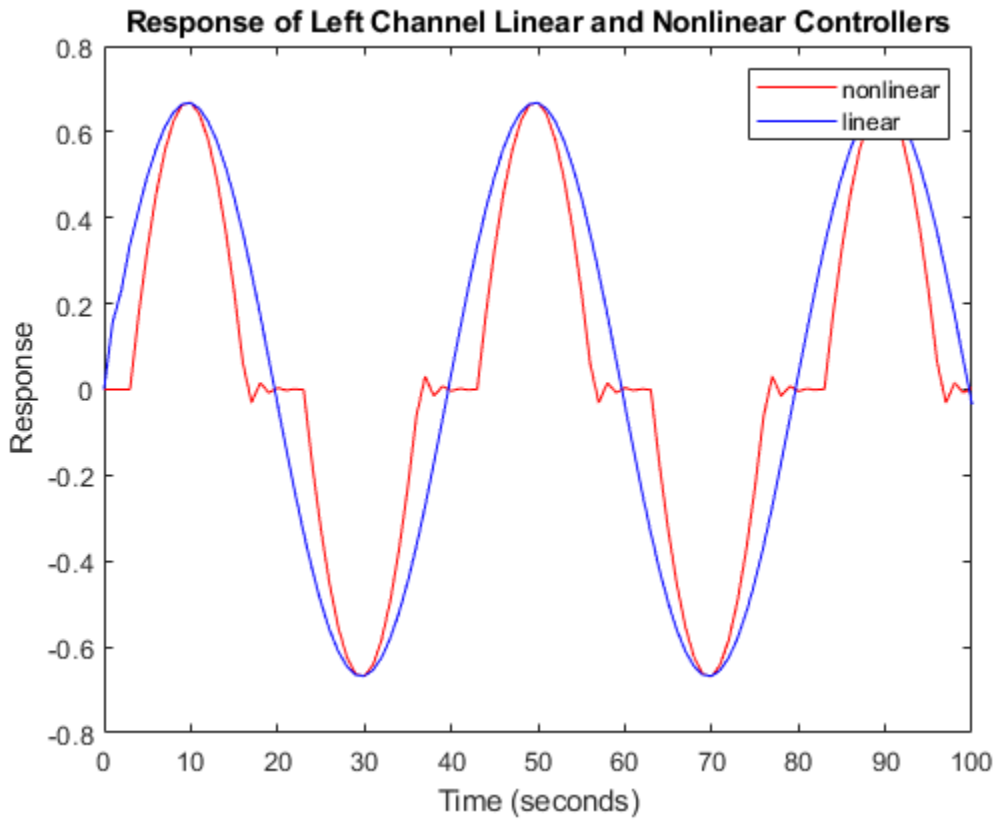
```
sim('rtwdemo_preprocessor_subsys')
youtnl = yout;
```

If you change the value of `VSSMODE` to 0, the model uses the linear controllers during simulation.

```
VSSMODE.Value = int32(0);
sim('rtwdemo_preprocessor_subsys')
youtl = yout;
```

You can plot and compare the response of the linear and nonlinear controllers:

```
figure('Tag','CloseMe');
plot(tout, youtnl.signals(1).values, 'r-', tout, youtl.signals(1).values, 'b-')
title('Response of Left Channel Linear and Nonlinear Controllers');
ylabel('Response');
xlabel('Time (seconds)');
legend('nonlinear','linear')
axis([0 100 -0.8 0.8]);
```



Using C Preprocessor Conditionals

This example model has been configured to generate C preprocessor conditionals. To generate code for the model, in **C Code** tab of the toolstrip select **Build**.

To activate code generation of preprocessor conditionals, check whether the following conditions are true:

- Select an Embedded Coder® target in **Code Generation > System target file** in the Configuration Parameters dialog box
- In the Variant Subsystem block parameter dialog box, set the **Variant activation time** parameter to `code compile`.

In this example, the generated code includes references to the Simulink.Variant objects `LINEAR` and `NONLINEAR`, as well as the definitions of macros corresponding to those variants. Those definitions depend on the value of `VSSMODE`, which is supplied in an external header file `rtwdemo_importedmacros.h`. The active variant is determined by using preprocessor conditionals (#if) on the macros (#define) `LINEAR` and `NONLINEAR`.

The macros `LINEAR` and `NONLINEAR` are defined in the generated `rtwdemo_preprocessor_subsys_types.h`, header file:

```
#ifndef LINEAR
#define LINEAR      (VSSMODE == 0)
#endif
```

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

In the generated code, the code related to the variants is guarded by C preprocessor conditionals. For example, in `rtwdemo_preprocessor_subsys.c`, the calls to the step and initialization functions of each variant are conditionally compiled:

```
/* Outputs for atomic SubSystem: '<Root>/LeftController' */
#if LINEAR
    /* Output and update for atomic system: '<S1>/Linear' */
#elif NONLINEAR
    /* Output and update for atomic system: '<S1>/Nonlinear' */
#endif
```

Close the model, figure, and workspace variables from the example.

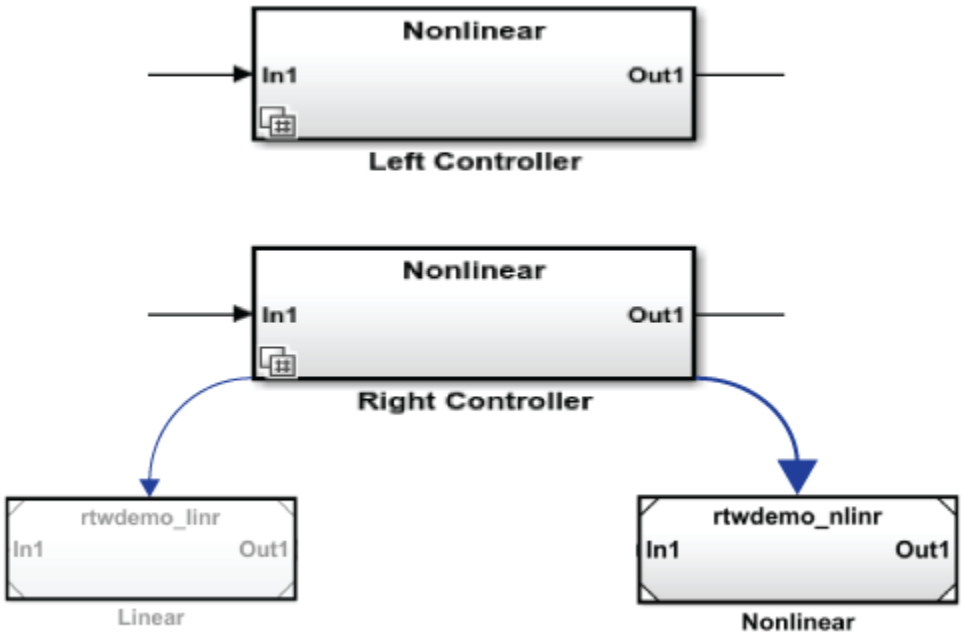
```
bdclose('rtwdemo_preprocessor_subsys')
close(findobj(0,'Tag','CloseMe'));
clear LINEAR NONLINEAR VSSMODE
clear tout yout youtl youtnl
```

Use Variant Models to Generate Code That Uses C Preprocessor Conditionals

This example shows you how to use variant models to generate code that uses preprocessor conditionals to control which code is linked into the embedded executable.

Overview of Variant Models

You can use a Model block to reference one Simulink® model (the *child model*) from another Simulink® model (the *parent model*). A Variant Subsystem block can have different *variants*. The variants can include a set of Model blocks, from which the Variant Subsystem block selects one. The figure below is a conceptual depiction of variant models. In this example the `Right Controller` model block can potentially select from two Model blocks. The models referenced by the Model blocks provide variations upon a nominal, prescribed functionality.

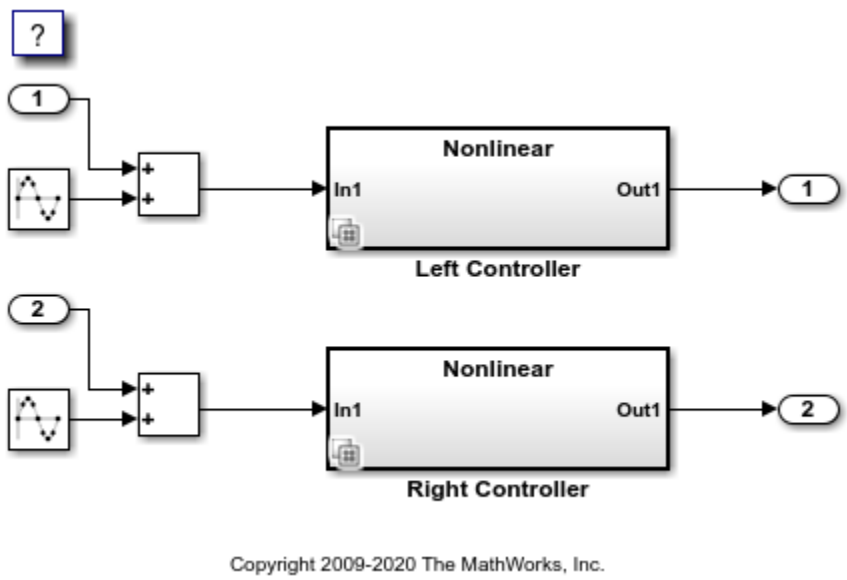


A Variant Subsystem block has only one variant active at a time. You can use the Variant Subsystem block dialog to select the active variant. Alternatively, you can parameterize the selection of the active variant, and make it dependent on the values of variables and objects in the base MATLAB® workspace. When you generate code, you can generate code for all variants and defer the choice of active variant until it is time to compile that code.

Specifying Variant Models

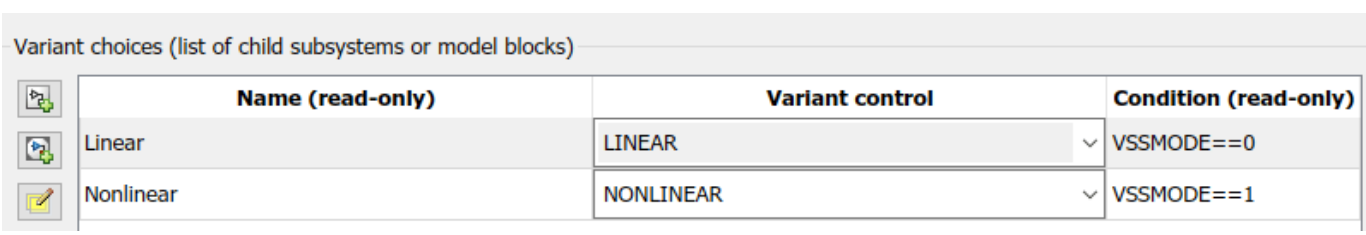
Opening the example model `rtwdemo_preprocessor` runs the **PostLoadFcn** defined in the "File: ModelProperties: Callbacks" dialog. This populates the base workspace with the control variables for the Variant Subsystem blocks.

```
open_system('rtwdemo_preprocessor')
```

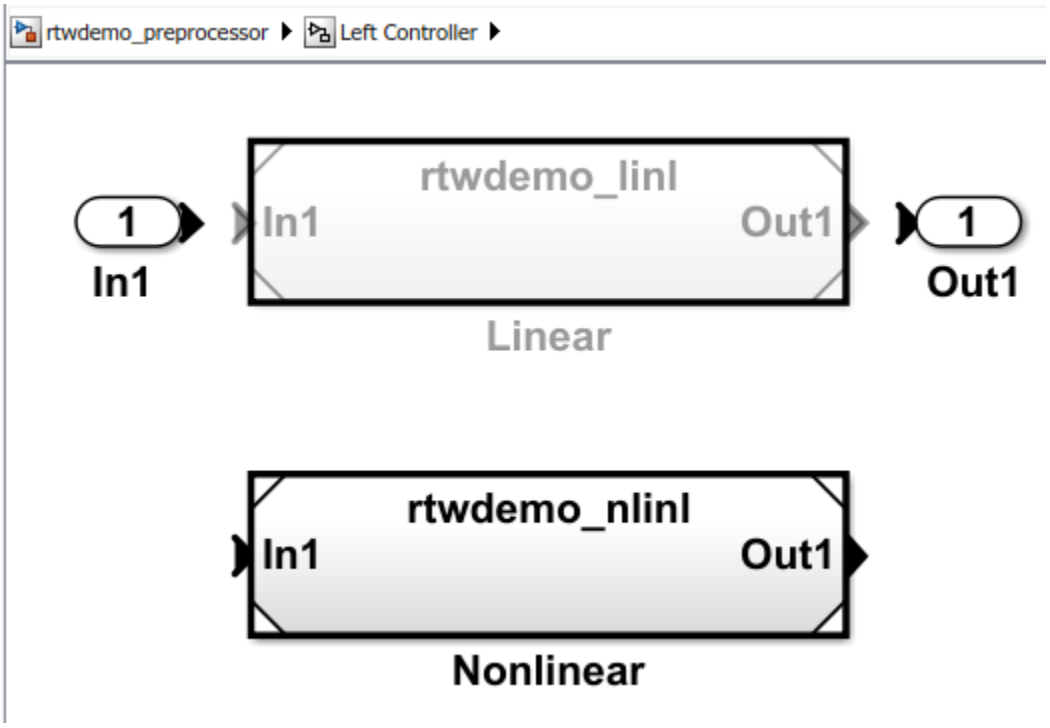


The `Left Controller` variant subsystem contains two child Model blocks: `Linear` and `Nonlinear`. The `Left Controller/Linear` child model executes when the `Simulink.Variant` object `LINEAR` evaluates to `true`, and the `Left Controller/Nonlinear` child model executes when the `Simulink.Variant` object `NONLINEAR` evaluates to `true`.

`Simulink.Variant` objects are specified for the `Left Controller` subsystem by right-clicking the `Left Controller` subsystem and selecting `Subsystem Parameters`, which opens the `Left Controller` Variant Subsystem block dialog box.



The `Left Controller` subsystem block dialog creates an association between the `Linear` and `Nonlinear` Model blocks with the two `Simulink.Variant` objects from the base workspace, `LINEAR` and `NONLINEAR`. These objects have a property named `Condition`, which is an expression that evaluates to a Boolean value and determines the active variant model (`Linear` or `Nonlinear`). The condition is also shown in the subsystem block dialog. In this example, the conditions of `LINEAR` and `NONLINEAR` are `'VSSMODE == 0'` and `'VSSMODE == 1'`, respectively.



In this example, `Simulink.Variant` objects are created in the base workspace.

```
LINEAR = Simulink.Variant;
LINEAR.Condition = 'VSSMODE==0';
NONLINEAR = Simulink.Variant;
NONLINEAR.Condition = 'VSSMODE==1';
```

Specifying a Variant Control Variable

Variant objects allow you to reuse arbitrarily complex conditions throughout a model. Multiple Variant Subsystem blocks can use the same `Simulink.Variant` objects, allowing you to toggle the activation of variant models as a set. You can toggle the set prior to simulation by changing the value of `VSSMODE` in the MATLAB environment or when compiling the generated code, as explained in the next section. In this example, `Left Controller` and `Right Controller` reference the same variant objects, so that you can toggle them simultaneously.

The nonlinear controller models implement hysteresis, while the linear controller models act as simple low-pass filters. Open the subsystem for the left channel. The models for the right channel are similar.

The generated code accesses the variant control variable `VSSMODE` as a user-defined macro. In this example, `rtwdemo_importedmacros.h` supplies `VSSMODE`. Within the MATLAB environment, you specify `VSSMODE` using a `Simulink.Parameter` object. Its value will be ignored when generating code including preprocessor conditionals. However, the value is used for simulation. The legacy header file specifies the value of the macro to be used when compiling the generated code, which ultimately activates one of the two specified variants in the embedded executable.

Variant control variables can be defined as `Simulink.Parameter` objects with one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant` (AUTOSAR)
- User-defined custom storage class that defines data as a macro in a specified header file

```
VSSMODE = Simulink.Parameter;
VSSMODE.Value = 1;
VSSMODE.DataType = 'int32';
VSSMODE.CoderInfo.StorageClass = 'Custom';
VSSMODE.CoderInfo.CustomStorageClass = 'ImportedDefine';
VSSMODE.CoderInfo.CustomAttributes.HeaderFile = 'rtwdemo_importedmacros.h';
```

Simulating the Model with Different Variants

Because you set the value of `VSSMODE` to `1`, the model uses the nonlinear controllers during simulation.

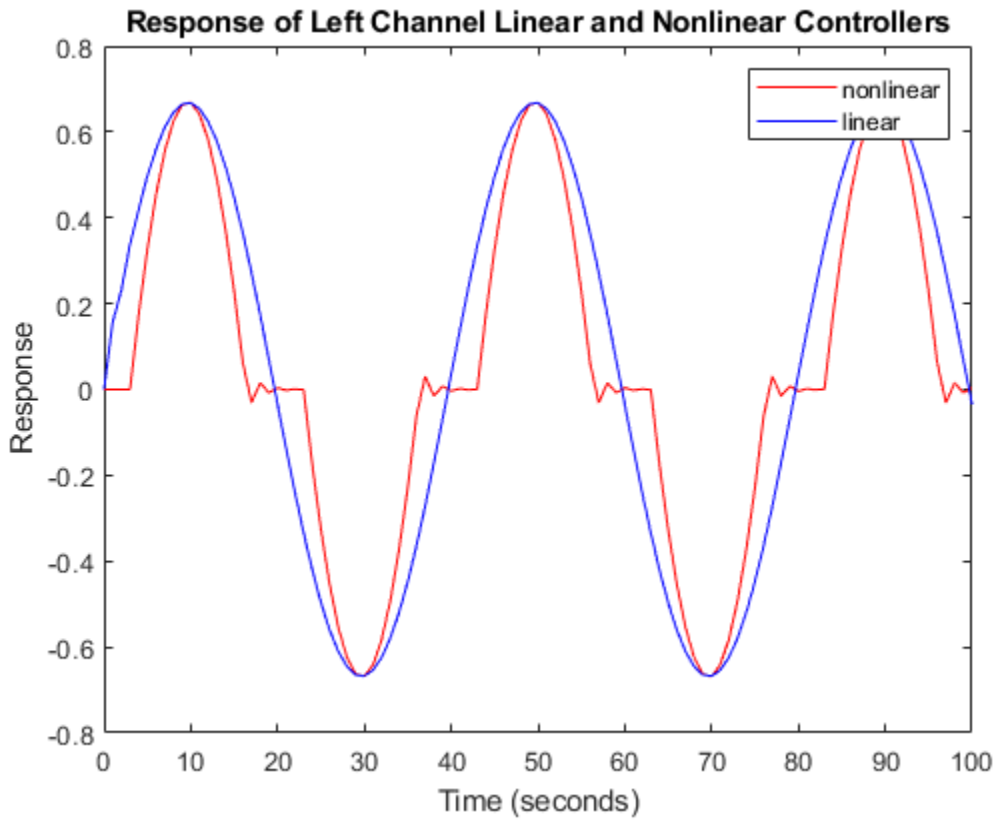
```
sim('rtwdemo_preprocessor')
youtnl = yout;
```

If you change the value of `VSSMODE` to `0`, the model uses the linear controllers during simulation.

```
VSSMODE.Value = int32(0);
sim('rtwdemo_preprocessor')
youtl = yout;
```

You can plot and compare the response of the linear and nonlinear controllers:

```
figure('Tag','CloseMe');
plot(tout, youtnl.signals(1).values, 'r-', tout, youtl.signals(1).values, 'b-')
title('Response of Left Channel Linear and Nonlinear Controllers');
ylabel('Response');
xlabel('Time (seconds)');
legend('nonlinear','linear')
axis([0 100 -0.8 0.8]);
```



Using C Preprocessor Conditionals

This example model has been configured to generate C preprocessor conditionals. To generate code for the model, in **C Code** tab of the toolstrip select **Build**.

To activate code generation of preprocessor conditionals, check whether the following conditions are true:

- Select an Embedded Coder® target in **Code Generation > System target file** in the Configuration Parameters dialog box
- In the Variant Subsystem block parameter dialog box, set the **Variant activation time** parameter to `code compile`.

In this example, the generated code includes references to the `Simulink.Variant` objects `LINEAR` and `NONLINEAR`, as well as the definitions of macros corresponding to those variants. Those definitions depend on the value of `VSSMODE`, which is supplied in an external header file `rtwdemo_importedmacros.h`. The active variant is determined by using preprocessor conditionals (`#if`) on the macros (`#define`) `LINEAR` and `NONLINEAR`.

The macros `LINEAR` and `NONLINEAR` are defined in the generated `rtwdemo_preprocessor_types.h` header file:

```
|#ifndef LINEAR|
|#define LINEAR      (VSSMODE == 0)|
|#endif|
```

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

In the generated code, the code related to the variants is guarded by C preprocessor conditionals. For example, in `rtwdemo_preprocessor.c`, the calls to the step and initialization functions of each variant are conditionally compiled:

```
  /* Outputs for Atomic SubSystem: '<Root>/Left Controller' */

  /* ModelReference: '<S1>/Linear' incorporates:
   *  ModelReference: '<S1>/Nonlinear'
   */
#if LINEAR

  rtwdemo_linl(&rtb_Add, &rtb_VariantMergeForOutportOut_h,
               &(rtwdemo_preprocessor_DWork.Linear_InstanceData_c.rtdw));

#elif NONLINEAR

  /* ModelReference: '<S1>/Nonlinear' */
  rtwdemo_nlinl(&rtb_Add, &rtb_VariantMergeForOutportOut_h,
                &(rtwdemo_preprocessor_DWork.Nonlinear_InstanceData_e.rtdw));

#endif                                  /* LINEAR */
```

Close the model, figure, and workspace variables from the example.

```
bdclose('rtwdemo_preprocessor')
close(findobj(0,'Tag','CloseMe'));
clear LINEAR NONLINEAR VSSMODE
clear tout yout youtl youtnl
```

Compile Code Conditionally for All Values of Variant Parameters with Same and Different Dimensions

This example shows how to generate a C code that contains all the active and inactive values of variant parameters. The values are enclosed in preprocessor conditionals `#if` and `#elif` that enables you to compile the code conditionally based on the condition that evaluates to `true`.

In this example, you will learn how to:

- Use storage classes to control the appearance, placement, definition, and declaration of variant parameter variables in the generated code.
- Use symbolic dimensions to propagate varying dimensions of variant parameters as symbols in the generated code. Representing dimensions as symbols enables the code to flexibly switch between dimension values for the given active choice without regenerating code every time you change the value of the variant control variable.

Overview of Variant Parameters

Variant parameters can have multiple values. Each value of the variant parameter is associated with a variant condition expression. During simulation, the value of the variant parameter associated with the condition that evaluates to `true` is the active value of that parameter. The value associated with the condition that evaluates to `false` is the inactive value of that variant parameter. For more information, see “Use Variant Parameters to Reuse Block Parameters with Different Values”.

In the generated code, the active and inactive values of the variant parameters are enclosed in C preprocessor conditionals `#if` and `#elif`. The preprocessor conditionals enable you to conditionally compile the code for a given active value. You generate the code only once. You can then choose the active choice of variant parameters by changing the value of the variant control variable before running the code. You do not need to regenerate the code for different variant parameter choices.

- **Representation of variant parameter variables**: The values are represented as inline or tunable variables in the generated code. The representation is determined by the storage class that you specify for the variant parameters in the model. The variant parameters with default storage class `Auto` are represented as inline variables. The variant parameters with storage class other than `Auto` are represented as tunable variables in the generated code. For more information on how different storage classes are represented in the generated code, see “Choose Storage Class for Controlling Data Representation in Generated Code”.
- **Representation of variant parameter dimensions**: If the values of a variant parameter have different dimensions, then the dimensions are represented as symbols. These symbols propagate throughout the model during simulation and then go into the generated code. To represent dimensions as symbols, you must set **Modeling** > **Model Settings** > **Diagnostics** > **Advanced Parameters** > **Allow symbolic dimension specification** to `on`. For more information on symbolic dimensions, see “Allow symbolic dimension specification”.

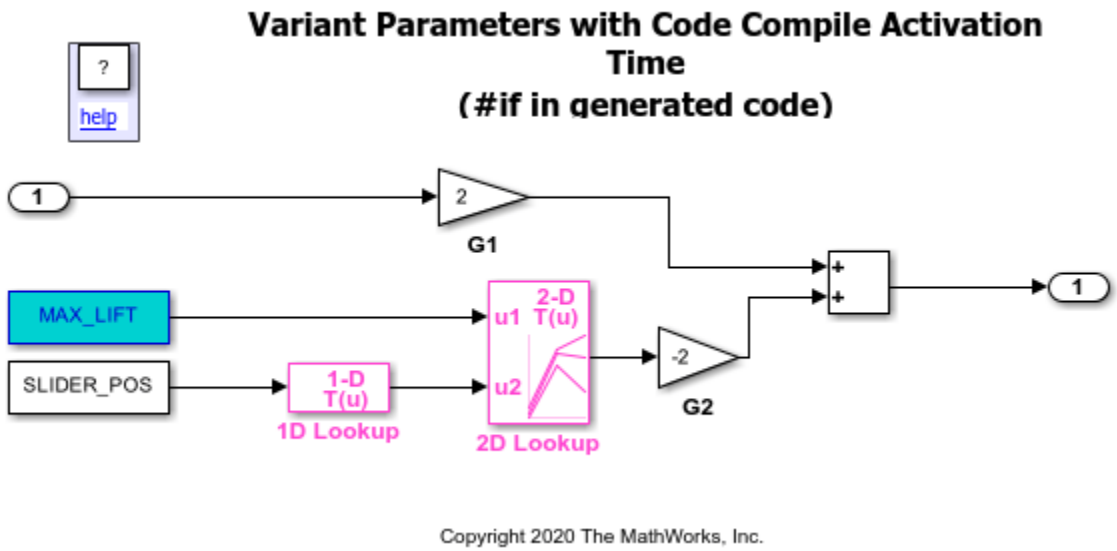
Prerequisite

Before you start this example, we recommend you complete “Options to Represent Variant Parameters in Generated Code” on page 27-53.

Explore the model

1. Open the model.

```
open_system('slexVariantParametersCC')
```



The model contains blocks that have variant parameters that are specified as `Simulink.VariantVariable` objects. The objects are defined in the `slexVariantParameterCCData.m` file.

- **Constant**: The **Constant** parameters of the Constant blocks are variant parameters. The parameters are specified as variant variable objects `MAX_LIFT` and `SLIDER_POS`.
- **Table data** and **Breakpoint sets**: The **Table data** and **Breakpoint sets** of the n-D Lookup Table blocks are variant parameters. The parameters are specified as variant variable objects `T1Break1`, `T1Data`, `T2Break`, and `T2Data`.

2. Open the `slexVariantParameterData.m`. Observe these settings in the file:

- In this file, `VCtrl` is the `Simulink.VariantControl` object that determines the active value of the variant parameter objects `MAX_LIFT`, `SLIDER_POS`, `T1Break1`, `T1Data`, `T2Break`, and `T2Data`. The value of `VCtrl` is set to `1`, and its activation time is specified as `code compile`. During simulation, the condition `VCtrl==1` evaluates to `true`. All the values associated with `VCtrl==1` become active, and the values associated with `VCtrl==2` become inactive. When you generate a code from this model, all the active and inactive values are enclosed in preprocessor conditionals `#if` and `#elif`.
- The variant parameter objects `MAX_LIFT` and `SLIDER_POS` have scalar numeric values with a default storage class `Auto`. When you generate the code, the values of the **Constant** parameters are inlined as macros. The variant parameter objects `T1Break1`, `T1Data`, `T2Break`, and `T2Data` are of `Simulink.Parameter` type with storage class specified as `ExportedGlobal`. When you generate the code, the values are represented by symbolic names as tunable variables in the generated code.

- The variant parameter objects `T1Break1` and `T1Data` have values with different dimensions. When you generate the code, the dimensions of these parameters are represented as symbols.

Set Active Choice of Variant Parameters

1. On the Simulink® toolstrip, click **Run**. During simulation, `VCtrl==1` evaluates to `true`. All the values associated with `VCtrl==1` are assigned to the corresponding variant variable objects and, subsequently, to the variant parameters that use these objects. For example, when `VCtrl==1` evaluates to `true`, the value of `MAX_LIFT` is set to `10`. As the **Constant** parameter of the `Constant1` block is set to `MAX_LIFT`, the value of the **Constant** parameter is also set to `10`.

2. To change the active values, change the value of `VCtrl` to `2`, then simulate the model again.

```
VCtrl.Value = 2;
```

During simulation, all the values associated with `VCtrl==2` are assigned to the variant variable objects, and those values are then assigned to the variant parameters using those objects.

Generate Code Using Embedded Coder

To generate code for variant parameters that have values with different dimensions, use Embedded Coder.

Before you generate code from the model, make sure that you have write permission in your current folder. To generate code, in the **Apps** gallery of the model toolstrip, click **Embedded Coder**. On the **C Code** tab, click **Build**. For more information, see “Generate Code Using Embedded Coder” on page 53-2.

Review Inline and Tunable Parameters in Generated Code

1. In the **C Code** tab, select **Open Report**.

2. Select the `slexVariantParametersCC_types.h` file from the **Generated Code** pane of the report. This file defines the value of the variant control variable `VCtrl` as `1`. The variant control variable determines the active value of variant parameters.

```
#ifndef VCtrl
#define VCtrl                          1
#endif
```

3. Select the `slexVariantParametersCC_private.h` file. This file includes macros (`#define`) corresponding to all the values of the variant parameters with default storage class `Auto`. The active values of variant parameters `MAX_LIFT` and `SLIDER_POS` are enclosed in the C preprocessor conditional statement (`#if`) on the macros `rtCP_Constant_MAX_LIFT` and `rtCP_Constant1_SLIDER_POS`.

```
#if VCtrl == 1 || VCtrl == 2
/* Variable: MAX_LIFT
 * Referenced by: '<Root>/Constant'
 */
#if VCtrl == 1
#define rtCP_Constant_MAX_LIFT         (10.0)
#elif VCtrl == 2
#define rtCP_Constant_MAX_LIFT         (20.0)
#endif
#endif
```

```
#if VCtrl == 1 || VCtrl == 2
/* Variable: SLIDER_POS
 * Referenced by: '<Root>/Constant1'
 */
#if VCtrl == 1
#define rtCP_Constant1_SLIDER_POS      (0.0)
#elif VCtrl == 2
#define rtCP_Constant1_SLIDER_POS      (0.5)
#endif
#endif
```

4. Select the `slexVariantParametersCC.h` file. This file includes symbolic names for all the values of the variant parameters with storage class set to `ExportGlobal`. The variables are defined as `extern` variables.

```
/* Exported Global Parameters
#if VCtrl == 1 || VCtrl == 2
extern real_T T1Break[T1Break_dim0]; /* Variable: T1Break
                                      * Referenced by:'<Root>/1D Lookup'
                                      */
#endif

#if VCtrl == 1 || VCtrl == 2
extern real_T T1Data[T1Data_dim0];   /* Variable: T1Data
                                      * Referenced by:'<Root>/1D Lookup'
                                      */
#endif

#if VCtrl == 1 || VCtrl == 2
extern real_T T2Break[3];            /* Variable: T2Break
                                      * Referenced by:'<Root>/2D Lookup'
                                      */
#endif

#if VCtrl == 1 || VCtrl == 2
extern real_T T2Data[9];             /* Variable: T2Data
                                      * Referenced by:'<Root>/2D Lookup'
                                      */
#endif
```

5. Select the `slexVariantParametersCC.c` file. All the values of variant parameters are enclosed in C preprocessor conditional statements `#if` and `#elif`. When you compile this code, Simulink evaluates the preprocessor conditionals and compiles the code only for the active values of variant parameters. You can then specify a different value for `VCtrl` and recompile the same code for any other active values of variant parameters.

```
/* Exported block parameters */
#if VCtrl == 1

real_T T1Break[T1Break_dim0] = { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0,
2.0, 3.0, 4.0, 5.0 } ;              /* Variable: T1Break
                                     * Referenced by: '<Root>/1D Lookup'
                                     */

real_T T1Data[T1Data_dim0] = { -0.99990920426259511, -0.999329299739067,
-0.99505475368673046, -0.9640275800758169, -0.76159415595576485, 0.0,
0.76159415595576485, 0.9640275800758169, 0.99505475368673046,
0.999329299739067, 0.99990920426259511 } ;
```

```
                                    /* Variable: T1Data
                                     * Referenced by: '<Root>/1D Lookup'
                                     */

real_T T2Break[3] = { -10.0, 0.0, 10.0 } ;
                                   /* Variable: T2Break
                                    * Referenced by: '<Root>/2D Lookup'
                                    */

real_T T2Data[9] = { 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0,
20.0, 23.0 } ;                     /* Variable: T2Data
                                    * Referenced by: '<Root>/2D Lookup'
                                    */

#elif VCtrl == 2

real_T T1Break[T1Break_dim0] = { -10.0, -9.0, -8.0, -7.0, -6.0, -5.0,
-4.0, -3.0,-2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
9.0, 10.0 } ;                     /* Variable: T1Break
                                   * Referenced by: '<Root>/1D Lookup'
                                   */

real_T T1Data[T1Data_dim0] = { -0.99999999587769273, -0.999999969540041,
-0.99999977492967584, -0.99999833694394469, -0.99998771165079559,
-0.99990920426259511, -0.999329299739067, -0.99505475368673046,
-0.9640275800758169, -0.76159415595576485, 0.0, 0.76159415595576485,
0.9640275800758169, 0.99505475368673046, 0.999329299739067,
0.99990920426259511, 0.99998771165079559, 0.99999833694394469,
0.99999977492967584, 0.999999969540041, 0.99999999587769273 } ;
                                  /* Variable: T1Data
                                   * Referenced by: '<Root>/1D Lookup'
                                   */

real_T T2Break[3] = { -20.0, 0.0, 20.0 };
                                  /* Variable: T2Break
                                   * Referenced by: '<Root>/2D Lookup'
                                   */

real_T T2Data[9] = { 8.0, 32.0, 20.0, 10.0, 38.0, 36.0, 12.0,
40.0, 46.0 } ;                    /* Variable: T2Data
                                   * Referenced by: '<Root>/2D Lookup'
                                   */

#endif
```

In this file, calls to the step function of each variant are conditionally compiled. In the step function, the macros and symbol names of variant parameters are used to form the equation.

```
/* Model step function */

void slexVariantParameters_step(void)
{
  slexVariantParametersCC_Y.Out1 = look2_binlx(rtCP_Constant_MAX_LIFT,
  look1_binlx(rtCP_Constant1_SLIDER_POS, T1Break, T1Data, T1Data_dim0
  - 1U), T2Break, T2Break, T2Data,
  slexVariantParametersCC_ConstP.uDLookup_maxIndex, 3U) * -2.0 + 2.0 *
  slexVariantParametersCC_U.In1;
}
```

Review Dimension Symbol in Generated Code

Select the `slexVariantParametersCC.c` file from the **Generated Code** pane of the report. In this file, the dimensions of variant parameters `T1Break` and `T1Data` are represented as symbols `T1Break_dim0` and `T1Data_dim0` and are enclosed in C preprocessor conditional statements `#if` and `#elif`. When you compile this code, Simulink evaluates the preprocessor conditionals and preserves the code only for the active values of variant parameters. If you specify the value of `VCtrl` as `1`, the condition `VCtrl == 1` evaluates to `true`, and the values enclosed in `V == 1` becomes active during code compilation. You can then specify a different value for `VCtrl` and recompile the same code for any other active values of `T1Break` and `T1Data`.

See Also

- “Change Active Values and Activation Time of Variant Parameters”
- “Run Executable for Different Variant Parameter Values Without Recompiling Code”

Timers in Simulink Coder

- “Absolute and Elapsed Time Computation” on page 28-2
- “Access Timers Programmatically” on page 28-4
- “Specify Clock Resolution Used by Target Environment Clock” on page 28-7
- “Generate Code for an Elapsed Time Counter” on page 28-11
- “Absolute Time Limitations” on page 28-14

Absolute and Elapsed Time Computation

In this section...

"About Timers" on page 28-2
"Timers for Periodic and Asynchronous Tasks" on page 28-2
"Allocation of Timers" on page 28-2
"Integer Timers in Generated Code" on page 28-3
"Elapsed Time Counters in Triggered Subsystems" on page 28-3

About Timers

Certain blocks require the value of either *absolute* time (that is, the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). Targets that support the real-time model (`rtModel`) data structure provide efficient time computation services to blocks that request absolute or elapsed time. Absolute and elapsed timer features include

- Timers are implemented as unsigned integers in generated code.
- In multirate models, at most one timer is allocated per rate. If no blocks executing at a given rate require a timer, a timer is not allocated to that rate. This minimizes memory allocated for timers and significantly reduces overhead involved in maintaining timers.
- Allocation of elapsed time counters for use of blocks within triggered subsystems is minimized, further reducing memory usage and overhead.
- S-function and TLC APIs let your S-functions access timers, in simulation and code generation.
- The word size of the timers is determined by a user-specified maximum counter value that you set with model configuration parameter **Application lifespan (days)**. If you specify this value, timers do not overflow. For more information, see "Control Memory Allocation for Time Counters".

See "Absolute Time Limitations" for more information about absolute time and the restrictions that it imposes.

Timers for Periodic and Asynchronous Tasks

Timing services provided for blocks execute within *periodic* tasks (that is, tasks running at the model base rate or subrates).

The code generator also provides timer support for blocks whose execution is *asynchronous* with respect to the periodic timing source of the model. See the following topics:

- "Timers in Asynchronous Tasks" on page 30-38
- "Create a Customized Asynchronous Library" on page 30-40

Allocation of Timers

If you create or maintain an S-Function block that requires absolute or elapsed time data, it must register the requirement (see "Access Timers Programmatically" on page 28-4). In multirate models, timers are allocated on a per-rate basis. For example, consider a model structured as follows:

- There are three rates, A, B, and C, in the model.
- No blocks running at rate B require absolute or elapsed time.
- Two blocks running at rate C register a requirement for absolute time.
- One block running at rate A registers a requirement for absolute time.

In this case, two timers are generated, running at rates A and C respectively. The timing engine updates the timers as the tasks associated with rates A and C execute. Blocks executing at rates A and C obtain time data from the timers associated with rates A and C.

Integer Timers in Generated Code

In the generated code, timers for absolute and elapsed time are implemented as unsigned integers. The default size is 64 bits. This is the amount of memory allocated for a timer if you set model configuration parameter **Application lifespan (days)** to `inf`. For an application with a sample rate of 1000 MHz, a 64-bit counter will not overflow for more than 500 years. See “Timers in Asynchronous Tasks” on page 30-38 and “Control Memory Allocation for Time Counters” on page 76-10 for more information.

Elapsed Time Counters in Triggered Subsystems

Some blocks, such as the Discrete-Time Integrator block, perform computations requiring the elapsed time (delta T) since the previous block execution. Blocks requiring elapsed time data must register the requirement (see “Access Timers Programmatically” on page 28-4). A triggered subsystem then allocates and maintains a single elapsed time counter if required. This timer functions at the subsystem level, not at the individual block level. The timer is generated if the triggered subsystem (or a unconditionally executed subsystem within the triggered subsystem) contains one or more blocks requiring elapsed time data.

Note If you are using simplified initialization mode, elapsed time is reset on first execution after becoming enabled, whether or not the subsystem is configured to reset on enable. For more information, see “Underspecified initialization detection”.

See Also

More About

- “Access Timers Programmatically”
- “Generate Code for an Elapsed Time Counter”
- “Optimize Memory Usage for Time Counters”
- “Absolute Time Limitations”

Access Timers Programmatically

In this section...

"About Timer APIs" on page 28-4
"C API for S-Functions" on page 28-4
"TLC API for Code Generation" on page 28-6

About Timer APIs

This topic describes APIs that let your S-functions take advantage of the efficiencies offered by absolute and elapsed timers. SimStruct macros are provided for use in simulation, and TLC functions are provided for inlined code generation. Note that

- To generate and use the new timers as described above, your S-functions must register the need to use an absolute or elapsed timer by calling `ssSetNeedAbsoluteTime` or `ssSetNeedElapseTime` in `mdlInitializeSampleTime`.
- Existing S-functions that read absolute time but do not register by using these macros continue to operate as expected, but generate less efficient code.

C API for S-Functions

The SimStruct macros described in this topic provide access to absolute and elapsed timers for S-functions during simulation.

In the functions below, the `SimStruct *S` argument is a pointer to the `simstruct` of the calling S-function.

- `void ssSetNeedAbsoluteTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires absolute time data, and allocates an absolute time counter for the rate at which the S-function executes (if such a counter has not already been allocated).
- `int ssGetNeedAbsoluteTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires absolute time.
- `double ssGetTaskTime(SimStruct *S, tid)`: read absolute time for a given task with task identifier `tid`. `ssGetTaskTime` operates transparently, regardless of whether or not you use the new timer features. `ssGetTaskTime` is documented in the SimStruct Functions chapter of the Simulink documentation.
- `void ssSetNeedElapseTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires elapsed time data, and allocates an elapsed time counter for the triggered subsystem in which the S-function executes (if such a counter has not already been allocated). See also "Elapsed Time Counters in Triggered Subsystems" on page 28-3.
- `int ssGetNeedElapseTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires elapsed time.
- `void ssGetElapseTime(SimStruct *S, (double *)elapseTime)`: returns, to the location pointed to by `elapseTime`, the value (as a `double`) of the elapsed time counter associated with the S-function.
- `void ssGetElapseTimeCounterDtype(SimStruct *S, (int *)dtype)`: returns the data type of the elapsed time counter associated with the S-function to the location pointed to by

dtype. This function is intended for use with the ssGetElapseTimeCounter function (see below).

- void ssGetElapseResolution(SimStruct *S, (double *)resolution): returns the resolution (that is, the sample time) of the elapsed time counter associated with the S-function to the location pointed to by resolution. This function is intended for use with the ssGetElapseTimeCounter function (see below).
- void ssGetElapseTimeCounter(SimStruct *S, (void *)elapseTime): This function is provided for the use of blocks that require the elapsed time values for fixed-point computations. ssGetElapseTimeCounter returns, to the location pointed to by elapseTime, the integer value of the elapsed time counter associated with the S-function. If the counter size is 64 bits, the value is returned as an array of two 32-bit words, with the low-order word stored at the lower address.

  To determine how to access the returned counter value, obtain the data type of the counter by calling ssGetElapseTimeCounterDtype, as in the following code:

```
int    *y_dtype;
ssGetElapseTimeCounterDtype(S, y_dtype);

 switch(*y_dtype) {
  case SS_DOUBLE_UINT32:
      {
          uint32_T dataPtr[2];
          ssGetElapseTimeCounter(S, dataPtr);
      }
      break;
  case SS_UINT32:
      {
         uint32_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
     }
      break;
  case SS_UINT16:
     {
          uint16_T dataPtr[1];
          ssGetElapseTimeCounter(S, dataPtr);
     }
      break;
  case SS_UINT8:
     {
         uint8_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
    }
     break;
  case SS_DOUBLE:
      {
         real_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
     }
      break;
  default:
    ssSetErrorStatus(S, "Invalid data type for elaspe time
        counter");
    break;
}
```

If you want to use the actual elapsed time, issue a call to the `ssGetElapseTime` function to access the elapsed time directly. You do not need to get the counter value and then calculate the elapsed time.

```
double *y_elapseTime;
.
.
.
ssGetElapseTime(S, elapseTime)
```

TLC API for Code Generation

The following TLC functions support elapsed time counters in generated code when you inline S-functions by writing TLC scripts for them.

- `LibGetTaskTimeFromTID(block)`: Generates code to read the absolute time for the task in which `block` executes.

  `LibGetTaskTimeFromTID` is documented with other sample time functions in the TLC Function Library Reference pages of the Target Language Compiler documentation.

  Note Do not use `LibGetT` for this purpose. `LibGetT` always reads the base rate (`tid 0`) timer. If `LibGetT` is called for a block executing at a subrate, the wrong timer is read, causing serious errors.

- `LibGetElapseTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See "Generate Code for an Elapsed Time Counter" on page 28-11 for an example of code generated by this function.
- `LibGetElapseTimeCounter(system)`: Generates code to read the integer value of the elapsed time counter for `system`. (`system` is the parent system of the calling block.) This function should be used in conjunction with `LibGetElapseTimeCounterDtypeId` and `LibGetElapseTimeResolution`. (See the discussion of `ssGetElapseTimeCounter` above.)
- `LibGetElapseTimeCounterDtypeId(system)`: Generates code that returns the data type of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)
- `LibGetElapseTimeResolution(system)`: Generates code that returns the resolution of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)

See Also

More About

- "Absolute and Elapsed Time Computation"
- "Generate Code for an Elapsed Time Counter"
- "Absolute Time Limitations"

Specify Clock Resolution Used by Target Environment Clock

For model functions that include blocks that request absolute or elapsed time values, improve the entry-point code generated for those functions by configuring the model to use a specific clock resolution. Clock resolution is the smallest increment of a clock value. For example, if a clock increments its value once per second, the clock resolution is 1 second.

Benefits of specifying a clock resolution include:

- Clock resolution that aligns with target environment clock requirements. For models configured to use a service code interface, specifying a clock resolution results in code that reads time values that produce more accurate results.
- Decoupling of the clock resolution and the solver properties that Simulink uses during simulation, such as the fixed-step size and sample times. The decoupling enables you to generate code that aligns with the target environment clock resolution and produces more accurate results.
- Influencing the data type that Simulink and the code generator use to represent time values. For example, in normal, accelerator, and rapid accelerator modes, Simulink uses the specified clock resolution to deduce fixed-point data types, which produces fixed-point simulation and generated code execution output that match.
- Portability of models between code interface configurations. You can attach a model that has a specified clock resolution to a shared Embedded Coder Dictionary that defines a data or service code interface configuration.
- Generated code that is easier to read.

Specify Clock Resolution

To specify a clock resolution for the code generator to apply for a model, configure the model with these model configuration parameter settings:

- Set **System target file** (`SystemTargetFile`) to `ert.tlc`.
- Set solver parameter **Type** (`SolverType`) to `Fixed-step`.
- Set **Clock resolution** (`ClockResolution`) to a scalar value of type `double`, which represents the clock resolution in seconds.

If the model includes periodic functions, the clock resolution that you specify must divide the sample times used by the periodic functions with no remainder. For example, if a model includes periodic functions that use sample times `1`, `0.2`, and `0.1`, the `0.05` is a valid clock resolution setting but `0.03` is not valid. If the value that you specify for the clock resolution is not a divisor, you must change the clock resolution setting or the offending sample time.

You can specify a clock resolution for components and subcomponents in a model hierarchy. The clock resolution settings for models across the hierarchy must match. For models configured to use a service interface, the code generator checks for clock resolution consistency across entry-point functions generated from the same model function used across the model hierarchy. If the clock resolution settings are not consistent, the code generator returns an error.

How Code Generator Uses Specified Clock Resolution

How the code generator uses a specified clock resolution depends on the model code interface configuration. For models configured to use a data code interface, a specified clock resolution influences how the code generator represents the clock tick in the generated code. The clock tick is

the increment of a processor clock. For models that you configure with a service code interface, a specified clock resolution influences timer service interface calls in the code.

For a model configured to use a service code interface, the code generator checks the value of the related **Application lifespan (days)** parameter. Timer service interfaces support 32-but clock tick values. If the **Application lifespan (days)** parameter setting is large enough to result in a clock tick that cannot be stored as a 32-bit value, then an overflow condition results and the code generator returns and error.

For aperiodic functions that are driven by an S-function that specifies the `SS_OPTION_ASYNCHRONOUS` option and a clock resolution, the clock resolution that the S-function specifies overrides the setting of the **Clock resolution** parameter.

For a model that uses a service code interface, when you specify a clock resolution, the code generator facilitates implementation of the timer service functions that run in the target environment and are called by the generated entry-point functions by including the clock resolution in two places:

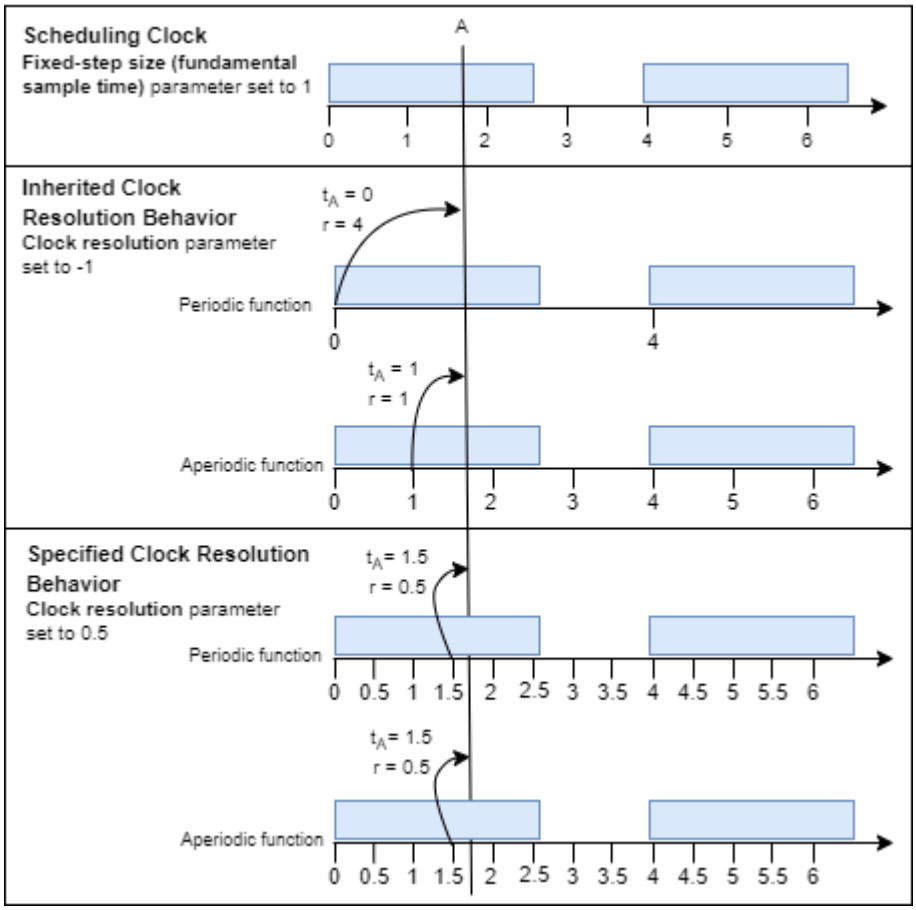
- In the Timer Service Interface section of the Code Interface Report. See “Analyze Generated Service Code Interface Report” on page 54-21.
- As metadata in the generated code descriptor file `codedescriptor.dmr`. You can access the metadata by using the code descriptor programming interface. See “Get Metadata About Service Interface” on page 54-54.

If you choose not to specify a clock resolution, the code generator uses these default values for the clock resolution:

- Model fixed-step size (fundamental sample time), for aperiodic functions
- Function sample time, for periodic functions

Suppose a model includes an aperiodic function and a periodic function, both of which use a block that requests absolute time from a target environment clock. The fixed-step size (fundamental sample time) for the model is `1` and the sample time configured for the periodic function is `4`. The model is configured to use a service code interface that applies the during-execution data communication method.

This figure shows the functions requesting the absolute time value at time A. When using the inherited clock resolution behavior (with **Clock resolution** set to `-1`), you can get time values that are more accurate by using aperiodic functions rather than periodic functions. You can get time values that are even more accurate by specifying a clock resolution as a scalar value.



Improve Accuracy of Time Values Read by Generated Entry-Point Functions

This example shows that for a model that is configured to use a service code interface, you can improve the accuracy of the time values read by generated entry-point functions by configuring the model with a specific clock resolution. The aperiodic function `Integrator` in example model `ComponentDeploymentFcn` includes the Discrete Time Integrator block, which requests the clock tick. Model configuration parameter **Fixed-step size (fundamental sample time)** is set to `auto`.

Generate and inspect the code for the model with the **Clock resolution** parameter set to the default value `-1`. In this case, the code generator initializes the clock resolution based on scheduling properties for the model style and the type of entry-point function.

Open example model `ComponentDeploymentFcn`.

```
open_system("ComponentDeploymentFcn.slx");
```

Use the default clock resolution to generate code, then inspect the code.

1. Open the Embedded Coder app.

2. Configure the `Integrator` function to use a timer service interface that is configured for during-execution communication. In the Code Mappings Editor – C, on the **Functions** tab, select the row for function `Exported Funciton:Aperiodic`. Click the pencil button . In the dialog box that appears, set the **Timer Service** property to `get_tick_during`. Save the model.

3. In the toolstrip, in the **Modeling** tab, click **Modeling Settings**.

4. In the Configuration Parameters dialog box, search for and select the **Clock resolution** parameter. The parameter is set to the default value `-1`. Close the dialog box.

5. Generate code.

6. In the code generation report, select the Code Interface Report and navigate to the Timer Service Interfaces section. The report shows that the clock resolution is set to 1 second.

7. In the generated file `ComponentDeployentFcn.c`, inspect the code for entry-point function `cd_integrator`. The code generator uses the clock resolution to calculate the integer value in the assignment statement for variable `tmp`. The value 1.25 is the product of the clock resolution (1) and the gain value specified for the Discrete Time Integrator block (1.25) and is the frequency at which the function requests the elapsed time.

```
tmp = 1.25 * (real_T)Integrator_ELAPS_T;
```

Configure the model with a clock resolution of `0.5`. Then, generate and reinspect the code.

1. In the toolstrip, click **Settings**.

2. In the Configuration Parameters dialog box, search for and select the **Clock resolution** parameter. Set the parameter to `0.5`. Apply the configuration change and close the dialog box.

3. Generate code.

4. In the code generation report, open the Code Interface Report and navigate to the Timer Service Interfaces section. The report shows that the clock resolution is set to `0.5` seconds.

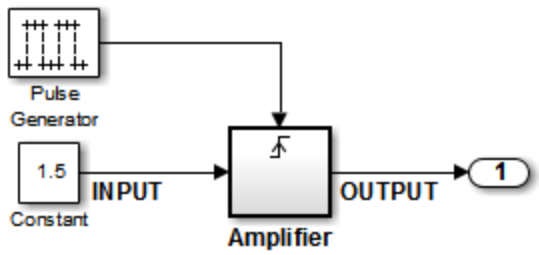
5. In the generated file `ComponentDeployentFcn.c`, reinspect the code for entry-point function `cd_integrator`. In this case, the code generator uses the specified clock resolution `0.5` to calculate the integer value in the assignment statement for variable `tmp`. The value `0.625` is the product of the clock resolution (`0.5`) and the gain value specified for the Discrete Time Integrator block (`1.25`). The generated code calls the timer service more frequently, which results in the function reading time values that are more accurate.

```
tmp = 0.625 * (real_T)Integrator_ELAPS_T;
```

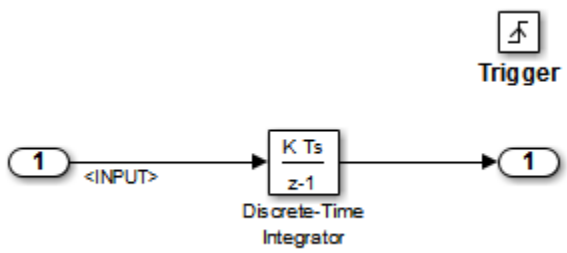
For more information about configuring timer service interfaces for models configured to use a service interface configuration, see “Timer Service Interfaces” on page 3-25. For information about configuring a model that uses a data code interface, see “Data Interfaces” on page 3-19.

Generate Code for an Elapsed Time Counter

This example shows a model that includes a triggered subsystem, `Amplifier`, consisting of a Discrete-Time Integrator block that uses an elapsed time counter. The model `ex_elapseTime` is in the folder `matlab/help/toolbox/rtw/examples`.



ex_elapseTime Model



Amplifier Subsystem

Code in the generated header file `ex_elapseTime.h` for the model uses 64 bits to implement the timer for the base rate (`clockTick0` and `clockTickH0`).

```
/*
 * Timing:
 * The following substructure contains information regarding
 * the timing information for the model.
 */
struct {
  time_T taskTime0;
  uint32_T clockTick0;
  uint32_T clockTickH0;
  time_T stepSize0;
  time_T tFinal;
  boolean_T stopRequestedFlag;
} Timing;
```

The code generator allocates storage for the previous-time value and elapsed-time value of the `Amplifier` subsystem (`Amplifier_PREV_T`) in the `D_Work(states)` structure in `ex_elapsedTime.h`.

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T DiscreteTimeIntegrator_DSTATE;
```

```
    /* '<S1>/Discrete-Time Integrator' */
  int32_T clockTickCounter;
    /* '<Root>/Pulse Generator' */
  uint32_T Amplifier_ELAPS_T[2];
    /* '<Root>/Amplifier' */
  uint32_T Amplifier_PREV_T[2];
    /* '<Root>/Amplifier' */
    } DW_ex_elapseTime_T;
```

The elapsed time computation is performed as follows within the `ex_elapseTime_step` function:

```
/* ---
   Outputs for Triggered SubSystem:
      '<Root>/Amplifier' incorporates:
         TriggerPort: '<S1>/Trigger'
 --- */
 zcEvent = rt_ZCFcn(RISING_ZERO_CROSSING,
        &ex_elapseTime_PrevZCX.Amplifier_Trig_ZCE,
        ((real_T)rtb_PulseGenerator));
    if (zcEvent != NO_ZCEVENT) {
   elapseT_H = ex_elapseTime_M->Timing.clockTickH0 -
        ex_elapseTime_DW.Amplifier_PREV_T[1];
      if (ex_elapseTime_DW.Amplifier_PREV_T[0] >
       ex_elapseTime_M->Timing.clockTick0) {
     elapseT_H--;
      }

   ex_elapseTime_DW.Amplifier_ELAPS_T[0] =
      ex_elapseTime_M->Timing.clockTick0
      - ex_elapseTime_DW.Amplifier_PREV_T[0];
   ex_elapseTime_DW.Amplifier_PREV_T[0] =
      ex_elapseTime_M->Timing.clockTick0;
   ex_elapseTime_DW.Amplifier_ELAPS_T[1] =
      elapseT_H;
   ex_elapseTime_DW.Amplifier_PREV_T[1] =
      ex_elapseTime_M->Timing.clockTickH0;
```

As shown above, the elapsed time is maintained as a state of the triggered subsystem. The Discrete-Time Integrator block finally performs its output and update computations using the elapsed time.

```
/* ---
DiscreteIntegrator: '<S1>/Discrete-Time Integrator'
--- */
OUTPUT = ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE;

/* ---
Update for DiscreteIntegrator:
   '<S1>/Discrete-Time Integrator' incorporates:
      Constant: '<Root>/Constant'
--- */
   ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE += 0.3 * (real_T)
     ex_elapseTime_DW.Amplifier_ELAPS_T[0] * 1.5;
```

See Also

More About

- “Absolute and Elapsed Time Computation”
- “Absolute Time Limitations”

Absolute Time Limitations

Absolute time is the time that has elapsed from the beginning of program execution to the present time, as distinct from *elapsed time*, the interval between two events. See “Absolute and Elapsed Time Computation” on page 28-2 for more information.

When you design an application that is intended to run indefinitely, you must take care when logging time values, or using charts or blocks that depend on absolute time. If the value of time reaches the largest value that can be represented by the data type used by the timer to store time, the timer overflows and the logged time or block output is incorrect.

If your target uses `rtModel`, you can avoid timer overflow by specifying a value for model configuration parameter **Application lifespan**. See “Integer Timers in Generated Code” on page 28-3 for more information.

The following limitations apply to absolute time:

- If you log time values by selecting model configuration parameter **Time**, your model uses absolute time.
- Every Stateflow chart that uses time is dependent on absolute time. The only way to eliminate the dependency is to change the Stateflow chart to not use time.
- The following Simulink blocks depend on absolute time:
  - Backlash
  - Chirp Signal
  - Clock
  - Derivative
  - Digital Clock
  - Discrete-Time Integrator (only when used in triggered subsystems)
  - From File
  - From Workspace
  - Pulse Generator
  - Ramp
  - Rate Limiter
  - Repeating Sequence
  - Signal Generator
  - Sine Wave (only when the **Sine type** parameter is set to `Time-based`)
  - Step
  - To File
  - To Workspace (only when logging to `StructureWithTime` format)
  - Transport Delay
  - Variable Transport Delay

  In addition to the Simulink blocks above, blocks in other blocksets may depend on absolute time. See the documentation for the blocksets that you use.

See Also

More About

- “Absolute and Elapsed Time Computation”

Time-Based Scheduling in Simulink Coder

Time-Based Scheduling and Code Generation

In this section...

"Sample Time Considerations" on page 29-2
"Tasking Modes" on page 29-2
"Model Execution and Rate Transitions" on page 29-3
"Execution During Simulink Model Simulation" on page 29-4
"Model Execution in Real Time" on page 29-4
"Single-Tasking Versus Multitasking Operation" on page 29-4

Sample Time Considerations

Simulink models run at one or more sample times. The Simulink product provides considerable flexibility in building multirate systems, that is, systems with more than one sample time. However, this same flexibility also allows you to construct models for which the code generator cannot generate real-time code for execution in a multitasking environment. To make multirate models operate as expected in real time (that is, to give the right answers), you sometimes must modify your model or instruct the Simulink engine to modify the model for you. In general, the modifications involve placing Rate Transition blocks between blocks that have unequal sample times. The following sections discuss issues you must address to use a multirate model in a multitasking environment. For a comprehensive discussion of sample times, including rate transitions, see "What Is Sample Time?", "Sample Times in Subsystems", "Sample Times in Systems", "Resolve Rate Transitions", and associated topics.

Tasking Modes

There are two execution modes for a fixed-step model: single-tasking and multitasking. These modes are available only for fixed-step solvers. To select an execution mode, select model configuration parameter **Treat each discrete rate as a separate task**. When you select this parameter, multitasking execution is applied for a multirate model. When you clear this parameter, single-tasking execution is applied.

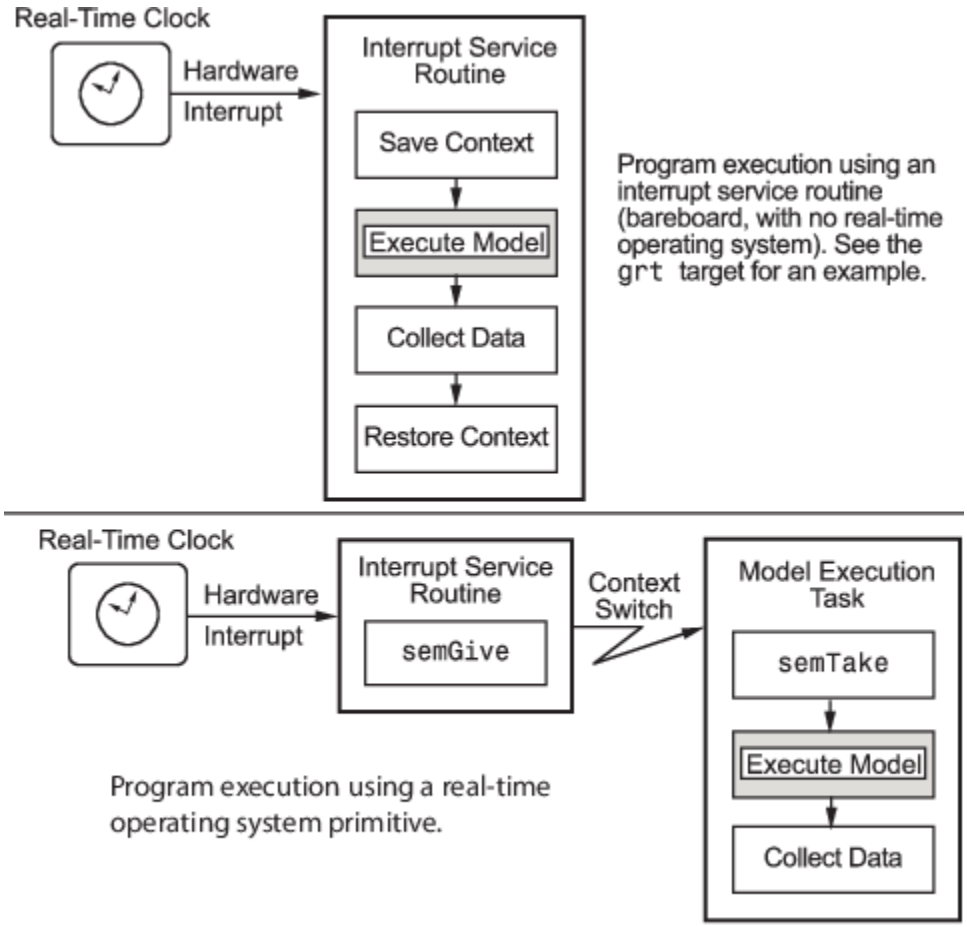
Note A model that is multirate and uses multitasking cannot reference a multirate model that uses single-tasking.

Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on *bare-metal* target hardware, where the model runs in the context of an interrupt service routine (ISR).

The fact that a system (such as The Open Group UNIX or Microsoft Windows systems) is multitasking does not imply that your program can execute in real time. This is because the program might not preempt other processes when required.

In operating systems (such as PC-DOS) where only one process can exist at a given time, an interrupt service routine (ISR) must perform the steps of saving the processor context, executing the model code, collecting data, and restoring the processor context.

Other operating systems, such as POSIX-compliant ones, provide automatic context switching and task scheduling. This simplifies the operations performed by the ISR. In this case, the ISR simply enables the model execution task, which is normally blocked. The next figure illustrates this difference.



Model Execution and Rate Transitions

To generate code that executes as expected in real time, you (or the Simulink engine) might need to identify and handle sample rate transitions within the model. In multitasking mode, by default the Simulink engine flags errors during simulation if the model contains invalid rate transitions. You can use the model configuration parameter **Multitask data transfer** to alter this behavior. Parameter **Single task data transfer** is available for the same purpose for single-tasking mode.

To avoid raising rate transition errors, insert Rate Transition blocks between tasks. You can request that the Simulink engine handle rate transitions automatically by inserting hidden Rate Transition blocks. See “Automatic Rate Transition” on page 29-19 for an explanation of this option.

To understand such problems, first consider how Simulink simulations differ from real-time programs.

Execution During Simulink Model Simulation

Before the Simulink engine simulates a model, it orders the blocks based upon their topological dependencies. This includes expanding virtual subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

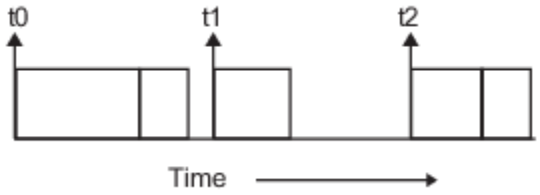
The key to this process is the ordering of blocks. A block whose output is directly dependent on its input (that is, a block with direct feedthrough) cannot execute until the block driving its input executes.

Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The output of such a block is determined by a value stored in memory, which can be updated independently of its input. During simulation, computations are performed prior to advancing the variable corresponding to time. This results in computations occurring instantaneously (that is, no computational delay).

Model Execution in Real Time

A real-time program differs from a Simulink simulation in that the program must execute the model code synchronously with real time. Every calculation results in some computational delay. This means the sample intervals cannot be shortened or lengthened (as they can be in a Simulink simulation), which leads to less efficient execution.

Consider the following timing figure.



Note the processing inefficiency in the sample interval `t1`. That interval cannot be compressed to increase execution speed because, by definition, sample times are clocked in real time.

You can circumvent this potential inefficiency by using the multitasking mode. The multitasking mode defines tasks with different priorities to execute parts of the model code that have different sample rates.

See “Multitasking and Pseudomultitasking Modes” on page 29-9 for a description of how this works. It is important to understand that section before proceeding here.

Single-Tasking Versus Multitasking Operation

Single-tasking programs require longer sample intervals, because all computations must be executed within each clock period. This can result in inefficient use of available CPU time, as shown in the previous figure.

Multitasking mode can improve the efficiency of your program if the model is large and has many blocks executing at each rate.

However, if your model is dominated by a single rate, and only a few blocks execute at a slower rate, multitasking can actually degrade performance. In such a model, the overhead incurred in task switching can be greater than the time required to execute the slower blocks. In this case, it is more efficient to execute all blocks at the dominant rate.

If you have a model that can benefit from multitasking execution, you might need to modify your model by adding Rate Transition blocks (or instruct the Simulink engine to do so) to generate expected results.

For more information about the two modes of execution and examples, see “Modeling for Single-Tasking Execution” on page 29-6 and “Modeling for Multitasking Execution” on page 29-9.

See Also

More About

- “What Is Sample Time?”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Configure Time-Based Scheduling”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Resolve Rate Transitions”
- “Handle Rate Transitions”
- “Time-Based Scheduling and Code Generation”
- “Modeling for Single-Tasking Execution”
- “Modeling for Multitasking Execution”
- “Time-Based Scheduling Example Models”

Modeling for Single-Tasking Execution

Single-Tasking Mode

You can execute model code in a strictly single-tasking manner. While this mode is less efficient with regard to execution speed, in certain situations, it can simplify your model.

In single-tasking mode, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The next figure illustrates the inefficiency inherent in single-tasking execution.

Single-tasking system execution requires a base sample rate that is long enough to execute one step through the entire model.

Build a Program for Single-Tasking Execution

To use single-tasking execution, clear model configuration parameter **Treat each discrete rate as a separate task**. If you select the parameter, single-tasking mode is used in the following cases:

- If your model contains one sample time
- If your model contains a continuous and a discrete sample time and the fixed step size is equal to the discrete sample time

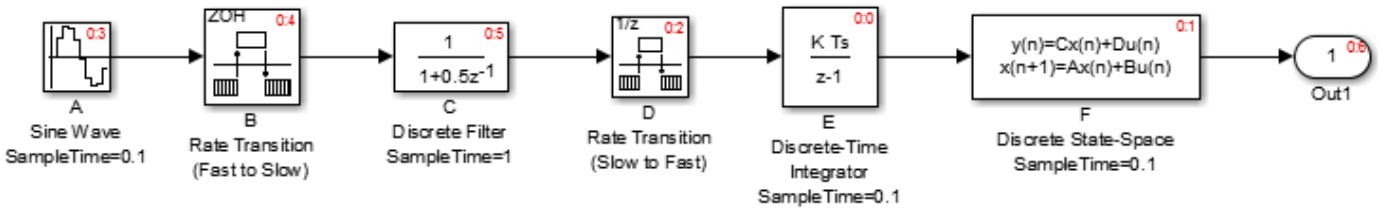
Single-Tasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting model configuration parameter **Treat each discrete rate as a separate task**.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. For more information, see “Simulation Phases in Dynamic Systems”.

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected) mode, with block parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** selected

This example considers the execution of the above model when model configuration parameter **Treat each discrete rate as a separate task** is cleared, which indicates the single-tasking mode.

In a single-tasking system, if you select model configuration parameter **Block reduction**, fast-to-slow Rate Transition blocks are optimized out of the model. The default case is shown (parameter **Block reduction** selected), so block B does not appear in the timing diagrams in this section. For more information, see “Block reduction”.

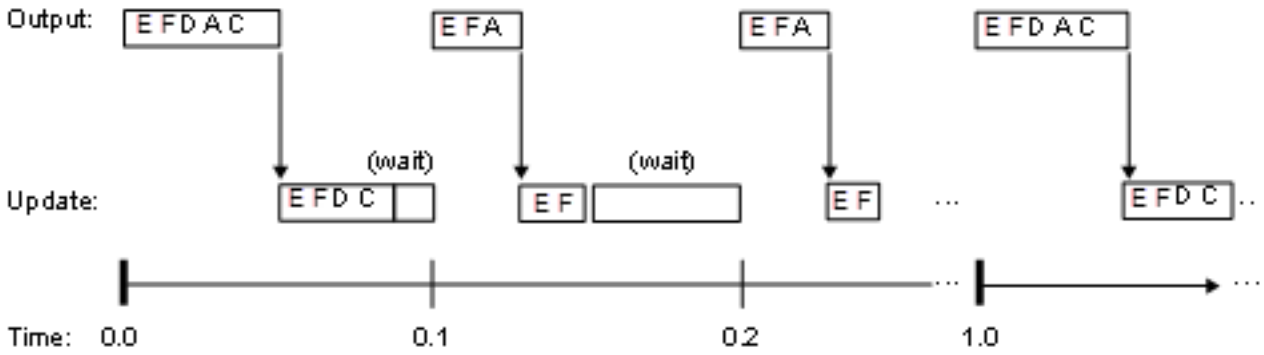
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

Execution Order and Sample Times (Single-Tasking)

Blocks (in Execution Order)	Sample Time (in Seconds)	Output	Update
E	0.1	Y	Y
F	0.1	Y	Y
D	1	Y	Y
A	0.1	Y	N
C	1	Y	Y

Real-Time Single-Tasking Execution

The next figure shows the scheduling of computations when the generated code is deployed in a real-time system. The generated program is shown running in real time, under control of interrupts from a 10 Hz timer.



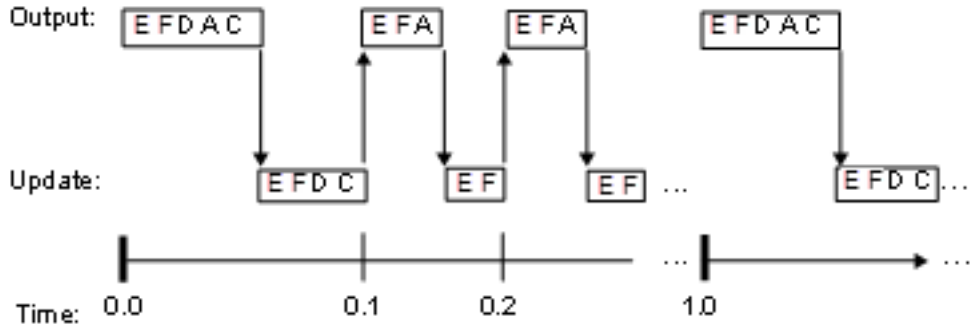
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

The fast blocks execute on every tick, at intervals of 0.1 second. Output computations are followed by update computations.

The system spends some portion of each time interval (labeled “wait”) idling. During the intervals when only the fast blocks execute, a larger portion of the interval is spent idling. This illustrates an inherent inefficiency of single-tasking mode.

Simulated Single-Tasking Execution

The next figure shows the execution of the model during the Simulink simulation loop.



Because time is simulated, the placement of ticks represents the iterations of the simulation loop. Blocks execute in exactly the same order as in the previous figure, but without the constraint of a real-time clock. Therefore there is no idle time between simulated sample periods.

See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Configure Time-Based Scheduling”
- “Time-Based Scheduling Example Models”

Modeling for Multitasking Execution

Multitasking and Pseudomultitasking Modes

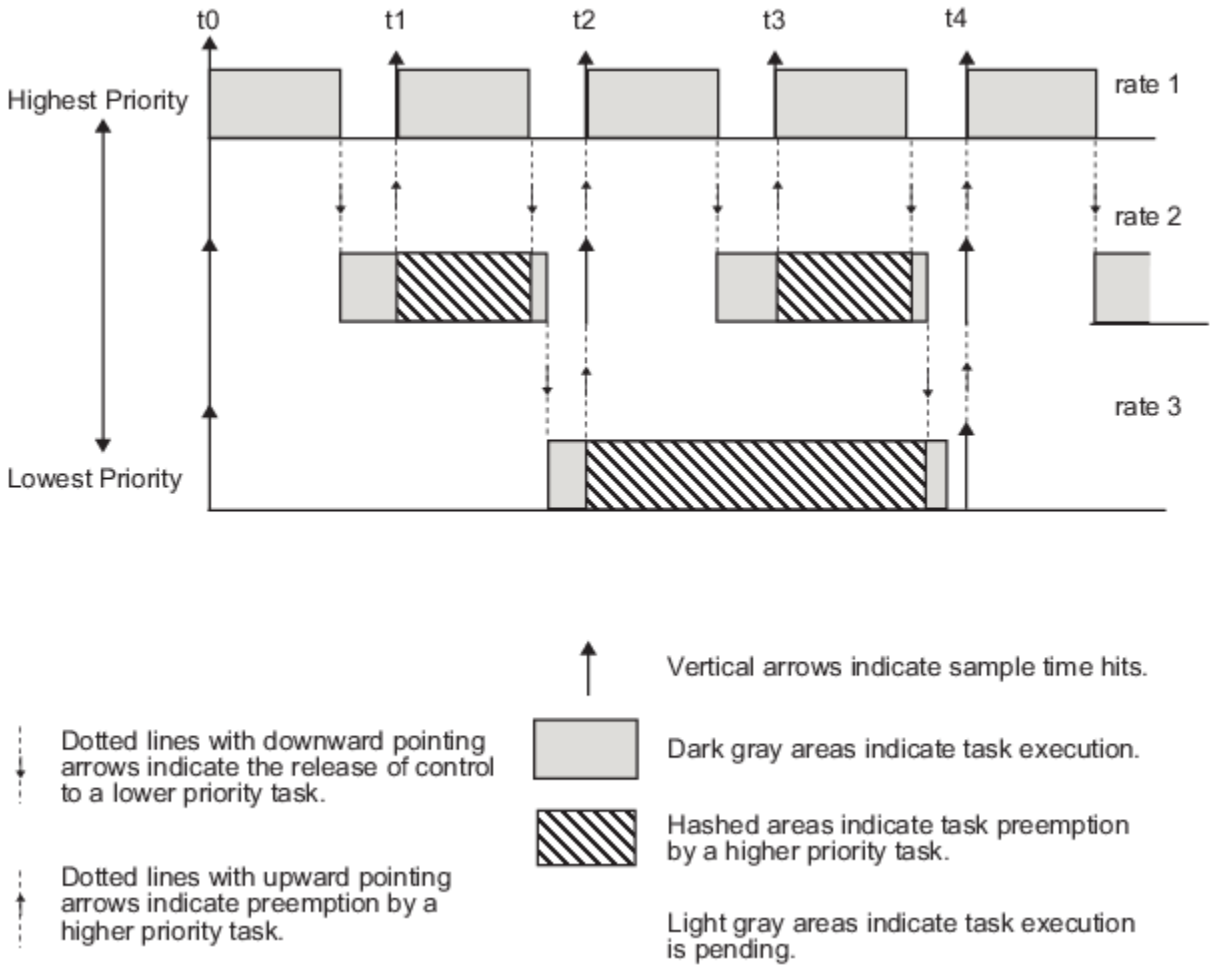
When periodic tasks execute in a multitasking mode, by default the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority tasks is used for processing lower priority tasks. This results in efficient program execution.

Where tasks are asynchronous rather than periodic, there may not necessarily be a relationship between sample rates and task priorities; the task with the highest priority need not have the fastest sample rate. You specify asynchronous task priorities using Async Interrupt and Task Sync blocks. You can switch the sense of what priority numbers mean by selecting or clearing model configuration parameter **Higher priority value indicates higher task priority**.

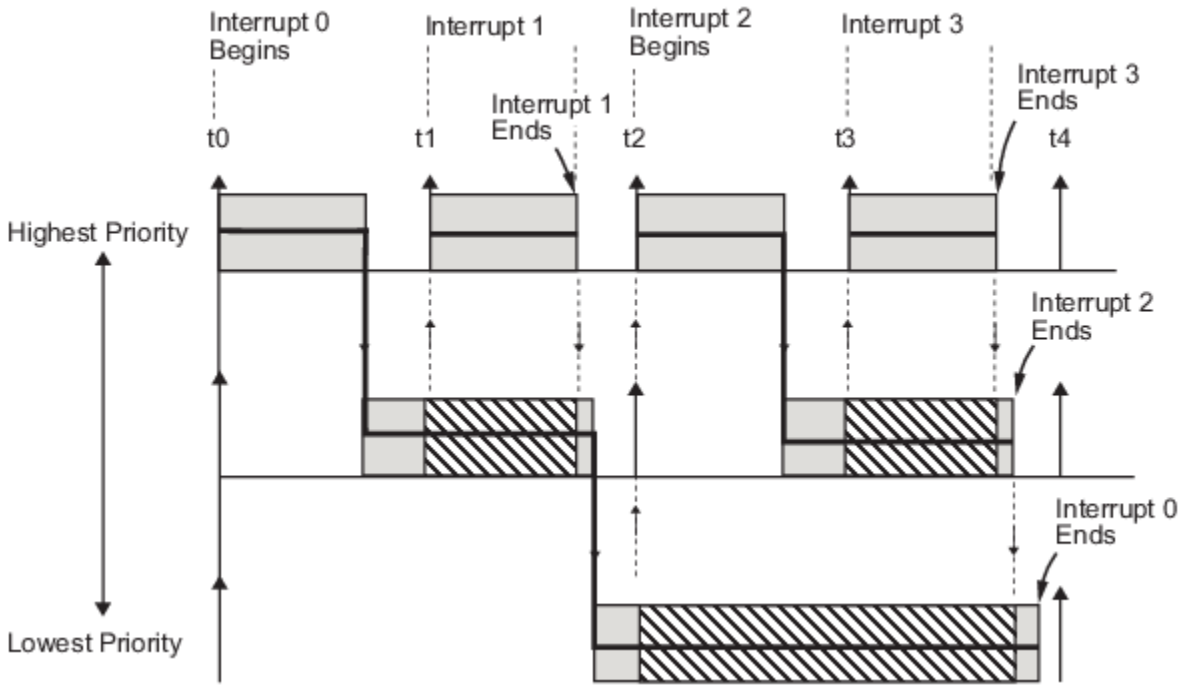
In multitasking environments (that is, under a real-time operating system), you can define separate tasks and assign them priorities. For bare-metal target hardware (that is, no real-time operating system present), you cannot create separate tasks. However, generated application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by programmatic context switching.

This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (that is, faster sample rate) code. Once complete, control is returned to the preempted ISR.

The next figures illustrate how timing of tasks in multirate systems are handled by the code generator in multitasking, pseudomultitasking, and single-tasking environments.



The next figure shows how overlapped interrupts are used to implement pseudomultitasking. In this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.



Build a Program for Multitasking Execution

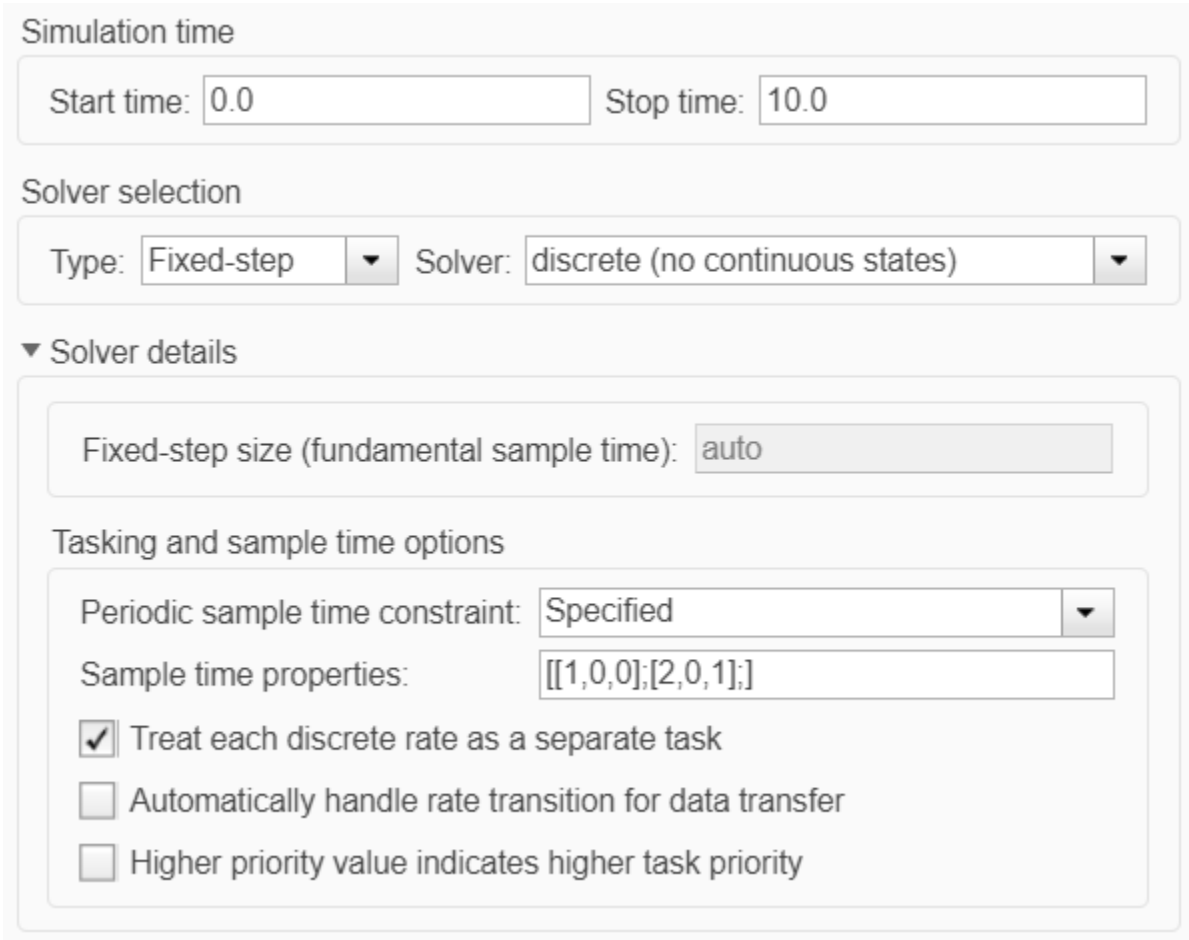
To use multitasking execution, select model configuration parameter **Treat each discrete rate as a separate task**. This parameter is active only if you set **Solver type** to `Fixed-step`. `Auto` mode results in a multitasking environment if your model has two or more different sample times. A model with a continuous and a discrete sample time runs in single-tasking mode if the fixed-step size is equal to the discrete sample time.

Execute Multitasking Models

In cases where the continuous part of a model executes at a rate that is different from the discrete part, or a model has blocks with different sample rates, the Simulink engine assigns each block a *task identifier* (`tid`) to associate the block with the task that executes at the block's sample rate.

You set sample rates and their constraints on the **Solver** pane of the Model Configuration Parameters dialog box. To generate code, select `Fixed-step` for the solver type. Certain restrictions apply to the sample rates that you can use:

- The sample rate of a block must be an integer multiple of the base (that is, the fastest) sample period.
- When you set model configuration parameter **Periodic sample time constraint** to `unconstrained`, the base sample period is determined by the setting of parameter **Fixed step size**.
- When you set parameter **Periodic sample time constraint** to `Specified`, the base rate fixed-step size is the first element of the sample time matrix that you specify for parameter **Sample time properties**. The **Solver** pane from the example model `rtwdemo_mrmtbb` shows an example.



- Continuous blocks execute by using an integration algorithm that runs at the base sample rate. The base sample period is the greatest common denominator of rates in the model only when you set model configuration parameter **Periodic sample time constraint** to `Unconstrained` and parameter **Fixed step size** to `Auto`.
- The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part and is an integer multiple of the base sample rate.

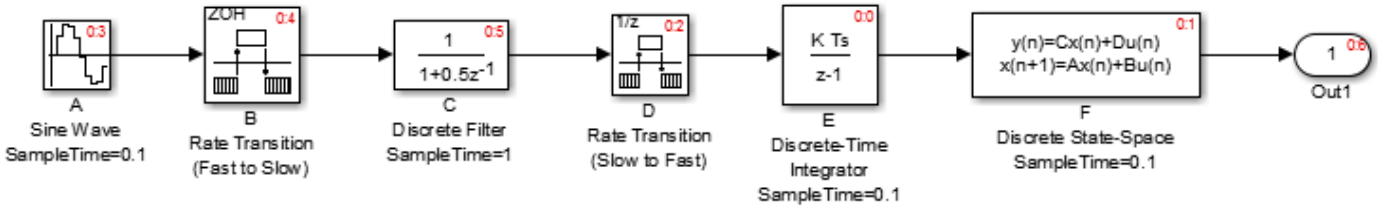
Multitasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting model configuration parameter **Treat each discrete rate as a separate task**.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. For more information, see “Simulation Phases in Dynamic Systems”.

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected) mode, with block parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** selected

This example considers the execution of the preceding model when model configuration parameter **Tasking mode** is set to `MultiTasking`. Block computations are executed under two tasks, prioritized by rate:

- The slower task, which gets the lower priority, is scheduled to run every second. This is called the *1 second task*.
- The faster task, which gets higher priority, is scheduled to run 10 times per second. This is called the *0.1 second task*. The 0.1 second task can preempt the 1 second task.

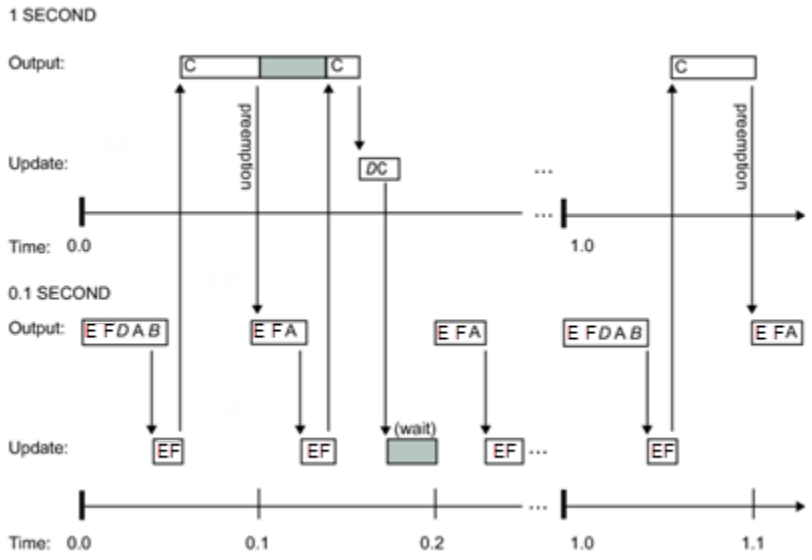
The following table shows, for each block in the model, the execution order, the task under which the block runs, and whether the block has an output or update computation. Blocks A and B do not have discrete states, and accordingly do not have an update computation.

Task Allocation of Blocks in Multitasking Execution

Blocks (in Execution Order)	Task	Output	Update
E	0.1 second task	Y	Y
F	0.1 second task	Y	Y
D	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. Update runs at input port sample time under 1 second task. For more information on port-based sample times, see “Referenced Model Sample Times”.	Y	Y
A	0.1 second task	Y	N
B	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. For more information on port-based sample times, see “Referenced Model Sample Times”.	Y	N
C	1 second task	Y	Y

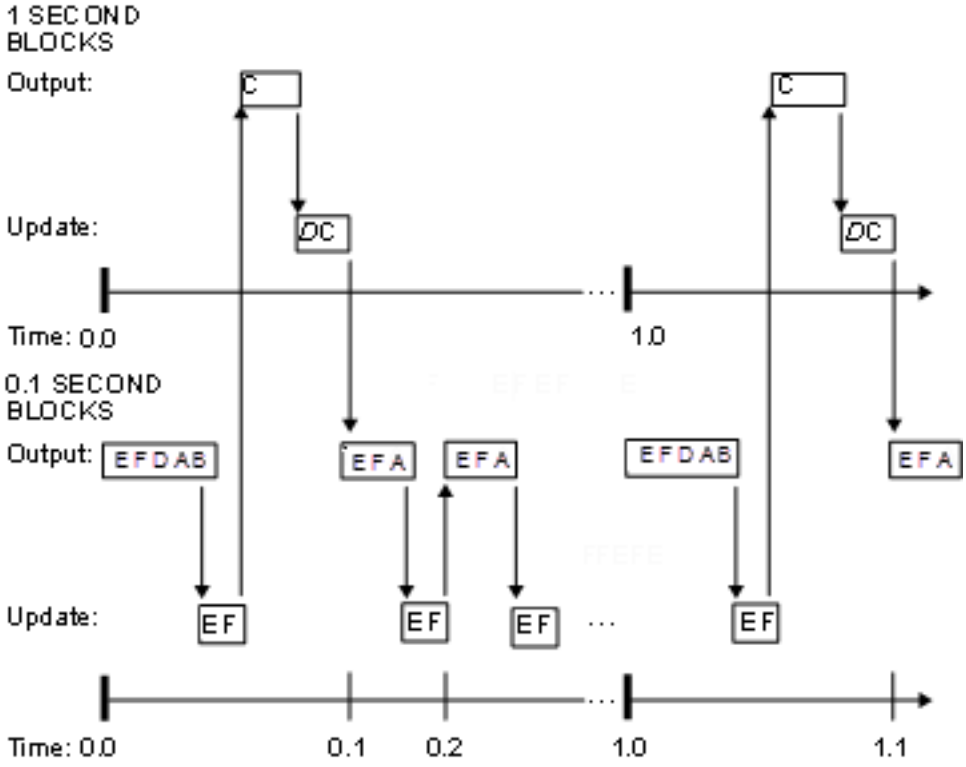
Real-Time Multitasking Execution

The next figure shows the scheduling of computations in `MultiTasking` solver mode when the generated code is deployed in a real-time system. The generated program is shown running in real time, as two tasks under control of interrupts from a 10 Hz timer.



Simulated Multitasking Execution

The next figure shows the Simulink execution of the same model, in `MultiTasking` solver mode. In this case, the Simulink engine runs the blocks in one thread of execution, simulating multitasking. No preemption occurs.



See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Configure Time-Based Scheduling”
- “Resolve Rate Transitions”
- “Handle Rate Transitions”
- “Time-Based Scheduling Example Models”

Handle Rate Transitions

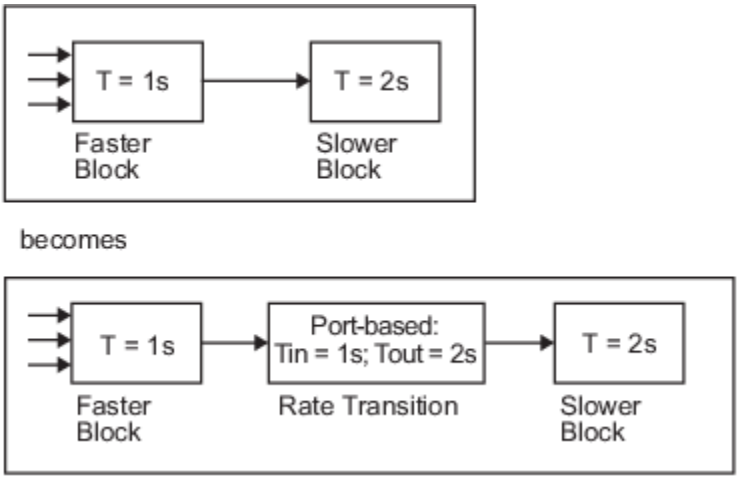
Rate Transitions

Two periodic sample rate transitions can exist within a model:

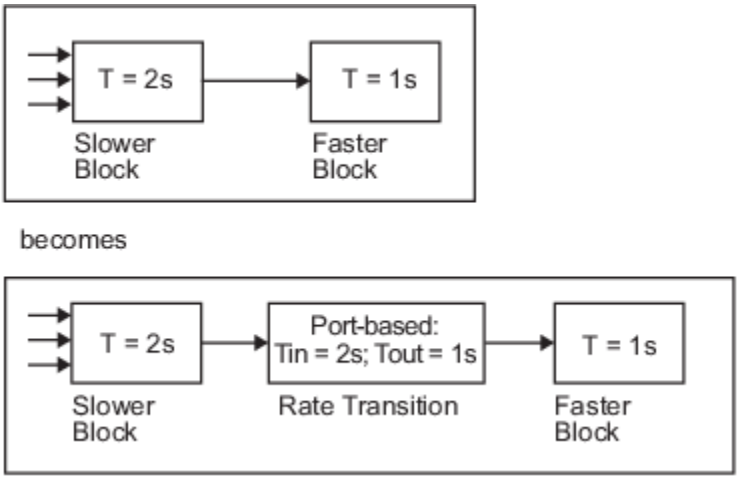
- A faster block driving a slower block
- A slower block driving a faster block

The following sections concern models with periodic sample times with zero offset only. Other considerations apply to multirate models that involve asynchronous tasks. For details on how to generate code for asynchronous multitasking, see “Asynchronous Support”.

In multitasking and pseudomultitasking systems, differing sample rates can cause blocks to be executed in the wrong order. To prevent possible errors in calculated data, you must control model execution at these transitions. When connecting faster and slower blocks, you or the Simulink engine must add Rate Transition blocks between them. Fast-to-slow transitions are illustrated in the next figure.



Slow-to-fast transitions are illustrated in the next figure.



Note Although the Rate Transition block offers a superset of the capabilities of the Unit Delay block (for slow-to-fast transitions) and the Zero-Order Hold block (for fast-to-slow transitions), you should use the Rate Transition block instead of these blocks.

Data Transfer Problems

Rate Transition blocks deal with issues of data integrity and determinism associated with data transfer between blocks running at different rates.

- *Data integrity*: A problem of data integrity exists when the input to a block changes during the execution of that block. Data integrity problems can be caused by preemption.

  Consider the following scenario:

  - A faster block supplies the input to a slower block.
  - The slower block reads an input value V_1 from the faster block and begins computations using that value.
  - The computations are preempted by another execution of the faster block, which computes a new output value V_2.
  - A data integrity problem now arises: when the slower block resumes execution, it continues its computations, now using the "new" input value V_2.

  Such a data transfer is called *unprotected*. "Faster to Slower Transitions in Real Time" on page 29-22 shows an unprotected data transfer.

  In a *protected* data transfer, the output V_1 of the faster block is held until the slower block finishes executing.
- *Deterministic* versus *nondeterministic* data transfer: In a *deterministic* data transfer, the timing of the data transfer is completely predictable, as determined by the sample rates of the blocks.

  The timing of a *nondeterministic* data transfer depends on the availability of data, the sample rates of the blocks, and the time at which the receiving block begins to execute relative to the driving block.

You can use the Rate Transition block to protect data transfers in your application and make them deterministic. These characteristics are considered desirable in most applications. However, the Rate Transition block supports flexible options that allow you to compromise data integrity and determinism in favor of lower latency. The next section summarizes these options.

Data Transfer Assumptions

When processing data transfers between tasks, the code generator makes these assumptions:

- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.

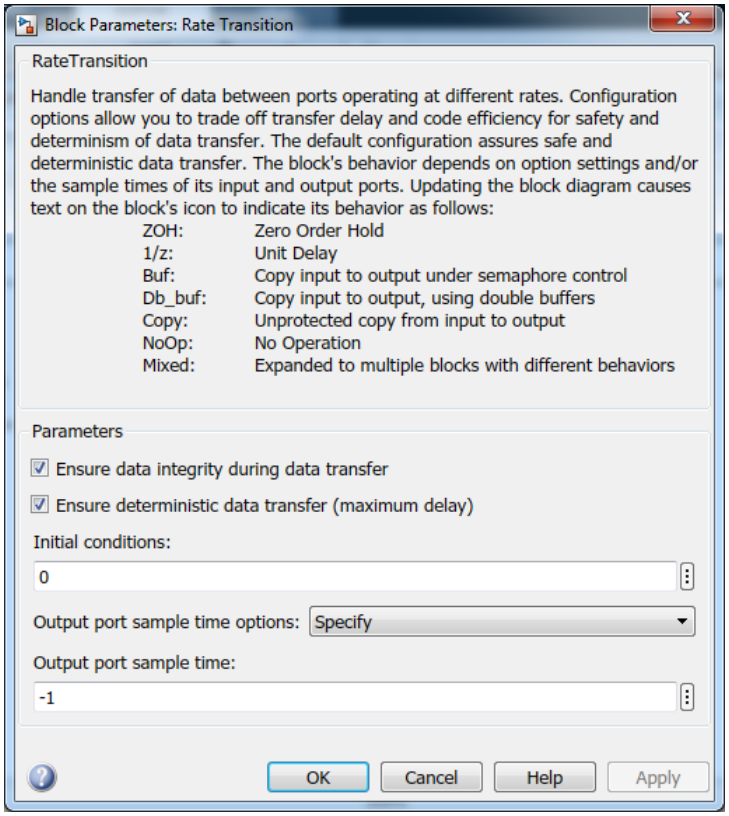
- Processes do not crash or restart (especially while data is transferred between tasks).

Rate Transition Block Options

Several parameters of the Rate Transition block are relevant to its use in code generation for real-time execution, as discussed below. For a complete block description, see Rate Transition.

The Rate Transition block handles periodic (fast to slow and slow to fast) and asynchronous transitions. When inserted between two blocks of differing sample rates, the Rate Transition block automatically configures its input and output sample rates for the type of transition; you do not need to specify whether a transition is slow-to-fast or fast-to-slow (low-to-high or high-to-low priorities for asynchronous tasks).

The critical decision you must make in configuring a Rate Transition block is the choice of data transfer mechanism to be used between the two rates. Your choice is dictated by considerations of safety, memory usage, and performance. As the Rate Transition block parameter dialog box in the next figure shows, the data transfer mechanism is controlled by two options.



- **Ensure data integrity during data transfer**: When you select this parameter, data transferred between rates maintains its integrity (the data transfer is protected). When you clear this parameter, data might not maintain its integrity (the data transfer is unprotected). By default, parameter **Ensure data integrity during data transfer** is selected.
- **Ensure deterministic data transfer (maximum delay)**: This parameter is supported for periodic tasks with an offset of zero and fast and slow rates that are multiples of each other. When this parameter is selected, the Rate Transition block behaves like a Zero-Order Hold block (for fast to slow transitions) or a Unit Delay block (for slow to fast transitions). The Rate Transition

block controls the timing of data transfer in a completely predictable way. When this parameter is cleared, the data transfer is nondeterministic. By default, **Ensure deterministic data transfer (maximum delay)** is selected for transitions between periodic rates with an offset of zero; for asynchronous transitions, it cannot be selected.

The Rate Transition block offers three modes of operation with respect to data transfer. In order of level of safety:

- **Protected/Deterministic (default):** This is the safest mode. The drawback of this mode is that it introduces deterministic latency into the system for the case of slow-to-fast periodic rate transitions. For that case, the latency introduced by the Rate Transition block is one sample period of the slower task. For the case of fast-to-slow periodic rate transitions, the Rate Transition block does not introduce additional latency.
- **Protected/NonDeterministic:** In this mode, for slow-to-fast periodic rate transitions, data integrity is protected by double-buffering data transferred between rates. For fast-to-slow periodic rate transitions, a semaphore flag is used. The blocks downstream from the Rate Transition block use the latest available data from the block that drives the Rate Transition block. Maximum latency is less than or equal to one sample period of the faster task.

  The drawbacks of this mode are its nondeterministic timing. The advantage of this mode is its low latency.
- **Unprotected/NonDeterministic:** This mode is not recommended for mission-critical applications. The latency of this mode is the same as for Protected/NonDeterministic mode, but memory requirements are reduced since neither double-buffering nor semaphores are required. That is, the Rate Transition block does nothing in this mode other than to pass signals through; it simply exists to notify you that a rate transition exists (and can cause generated code to compute incorrect answers). Selecting this mode, however, generates the least amount of code.

  Note In unprotected mode (**Ensure data integrity during data transfer** is selected), the Rate Transition block does nothing other than allow the rate transition to exist in the model.

Rate Transition Blocks and Continuous Time

The sample time at the output port of a Rate Transition block can only be discrete or fixed in minor time step. This means that when a Rate Transition block inherits continuous sample time from its destination block, it treats the inherited sample time as Fixed in Minor Time Step. Therefore, the output function of the Rate Transition block runs only at major time steps. If the destination block sample time is continuous, Rate Transition block output sample time is the base rate sample time (if solver is fixed-step), or zero-order-hold-continuous sample time (if solver is variable-step).

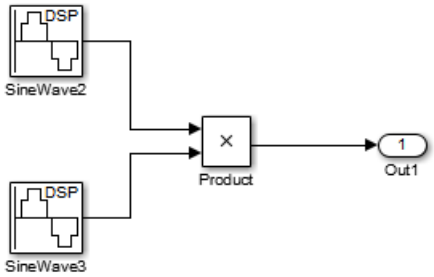
Automatic Rate Transition

The Simulink engine can detect mismatched rate transitions in a multitasking model during an update diagram and automatically insert Rate Transition blocks to handle them. To enable this, select model configuration parameter **Automatically handle rate transition for data transfer**. By default, this parameter is cleared. When you select this parameter:

- Simulink handles transitions between periodic sample times and asynchronous tasks.
- Simulink inserts hidden Rate Transition blocks in the block diagram.

- The code generator produces code for the Rate Transition blocks that were automatically inserted. This code is identical to the code generated for Rate Transition blocks that were inserted manually.
- Automatically inserted Rate Transition blocks operate in protected mode for periodic tasks and asynchronous tasks. You cannot alter this behavior. For periodic tasks, automatically inserted Rate Transition blocks operate with the level of determinism specified by model configuration parameter **Deterministic data transfer**. The default setting is `Whenever possible`, which enables determinism for data transfers between periodic sample-times that are related by an integer multiple. For more information, see Deterministic data transfer. To use other modes, you must insert Rate Transition blocks and set their modes manually.

For example, in this model, SineWave2 has a sample time of 2, and SineWave3 has a sample time of 3.

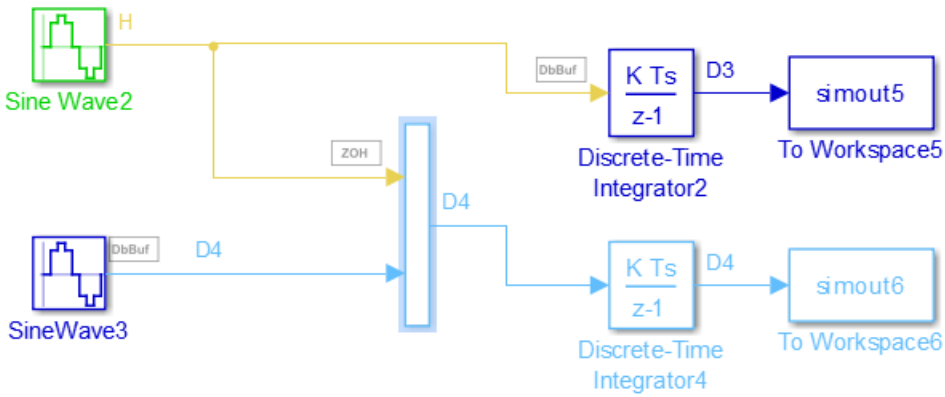


When you select model configuration parameter **Automatically handle rate transition for data transfer**, Simulink inserts a Rate Transition block between each Sine Wave block and the Product block. The inserted blocks have the parameter values to reconcile the Sine Wave block sample times.

If the input port and output port data sample rates in a model are not multiples of each other, Simulink inserts a Rate Transition block whose sample rate is the greatest common divisor (GCD) of the two rates. If no other block in the model contains this new rate, an error occurs during simulation. In this case, you must insert a Rate Transition block manually.

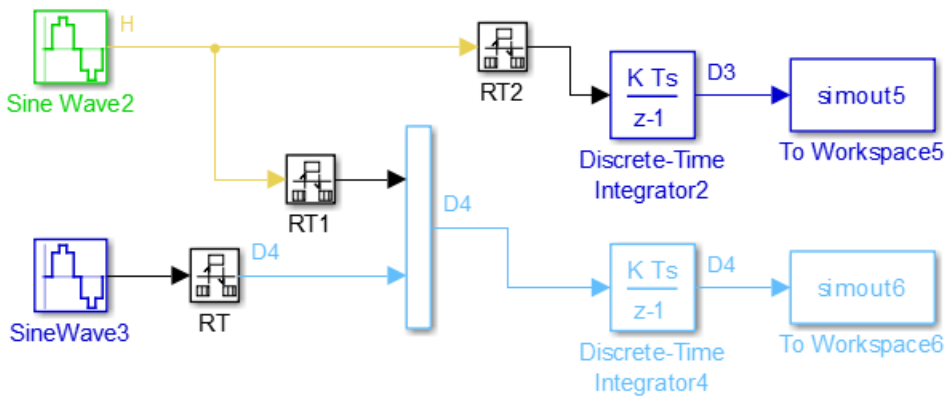
Visualize Inserted Rate Transition Blocks

When you select model configuration parameter **Automatically handle rate transition for data transfer**, Simulink inserts Rate Transition blocks in the paths that have mismatched transition rates. These blocks are hidden by default. To visualize the inserted blocks, update the diagram. Badge labels appear in the model and indicate where Simulink inserted Rate Transition blocks during the compilation phase. For example, in this model, three Rate Transition blocks were inserted between the two Sine Wave blocks and the Multiplexer and Integrator when the model compiled. The ZOH and DbBuf badge labels indicate these blocks.



You can show or hide badge labels. Open the **Debug** tab. In the **Diagnostics** section of the **Information Overlays/Sample Time** gallery select or clear **Automatic Rate Transitions**.

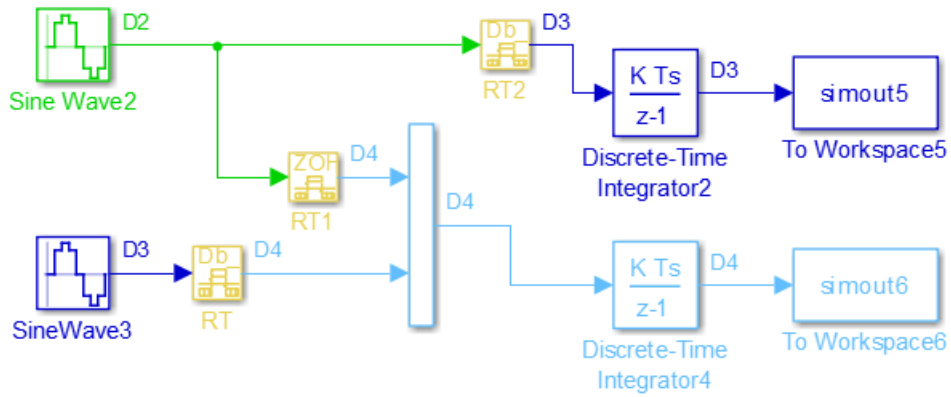
To configure the hidden Rate Transition blocks so that the blocks are visible, right-click a badge label and click **Insert rate transition block**.



When you make hidden Rate Transition blocks visible:

- You can see the type of Rate Transition block inserted and the location of the block in the model.
- You can set the block parameter **Initial Conditions**.
- You can change block parameter settings for rate transfer.

Validate the changes to your model by updating your diagram.



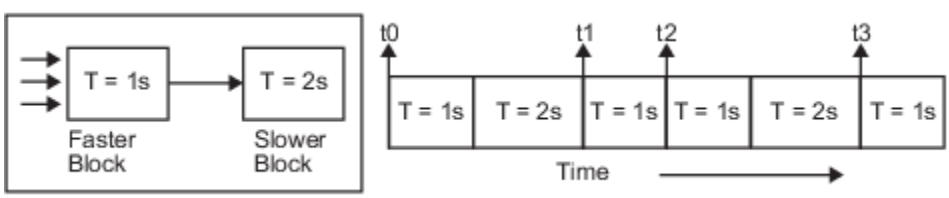
Displaying inserted Rate Transition blocks is not compatible with export-function models.
To learn more about the types of Rate Transition blocks, see Rate Transition.

Periodic Sample Rate Transitions

These sections describe cases in which Rate Transition blocks are required for periodic sample rate transitions. The discussion and timing diagrams in these sections are based on the assumption that the Rate Transition block is used in its default (protected/deterministic) mode. Model configuration parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** are selected. These are the settings used for automatically inserted Rate Transition blocks.

Faster to Slower Transitions in a Simulink Model

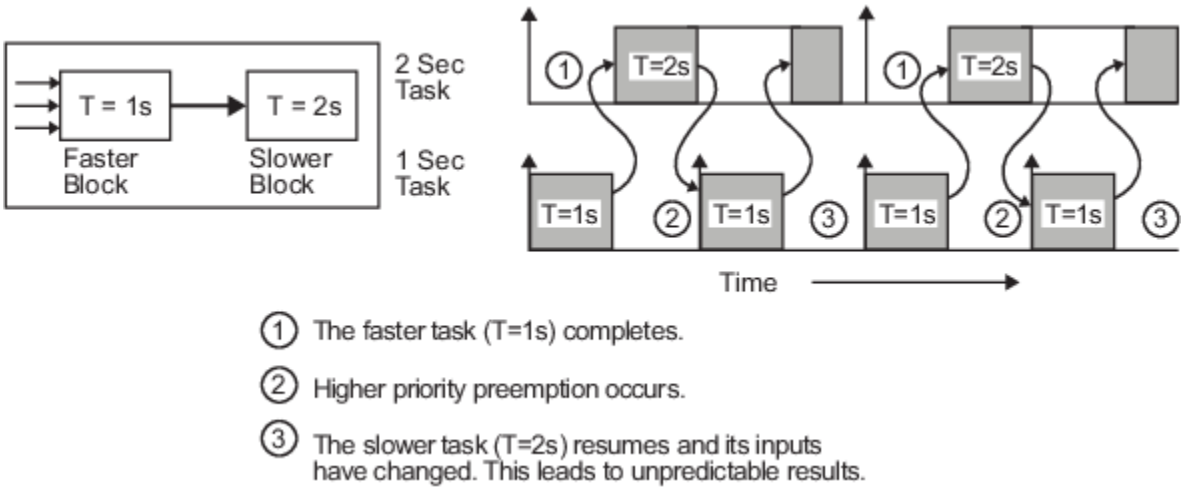
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The next figure illustrates this situation.



A Simulink simulation does not execute in real time, which means that it is not bound by real-time constraints. The simulation waits for, or moves ahead to, whatever tasks are required to complete simulation flow. The actual time interval between sample time steps can vary.

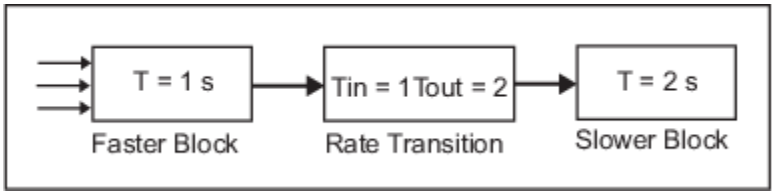
Faster to Slower Transitions in Real Time

In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block might span more than one execution period of the faster block. This means that the outputs of the faster block can change before the slower block has finished computing its outputs. The next figure shows a situation in which this problem arises (T = sample time). Note that lower priority tasks are preempted by higher priority tasks before completion.



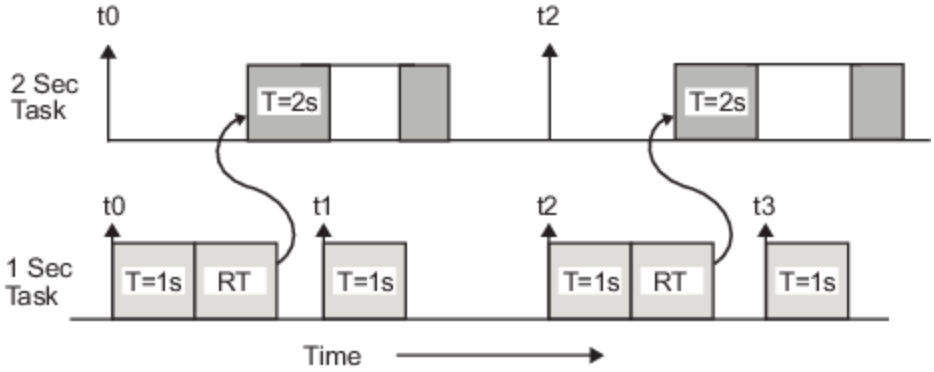
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data might not maintain its integrity in this situation.

To avoid this situation, the Simulink engine must hold the outputs of the 1 second (faster) block until the 2 second (slower) block finishes executing. The way to accomplish this is by inserting a Rate Transition block between the 1 second and 2 second blocks. The input to the slower block does not change during its execution, maintaining data integrity.



It is assumed that the Rate Transition block is used in its default (protected/deterministic) mode.

The Rate Transition block executes at the sample rate of the slower block, but with the priority of the faster block.

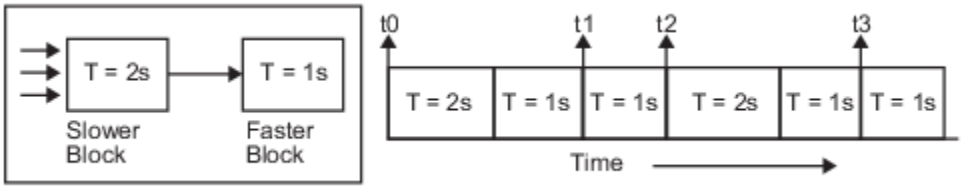


When you add a Rate Transition block, the block executes before the 2 second block (its priority is higher) and its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

Slower to Faster Transitions in a Simulink Model

In a model where a slower block drives a faster block, the Simulink engine again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

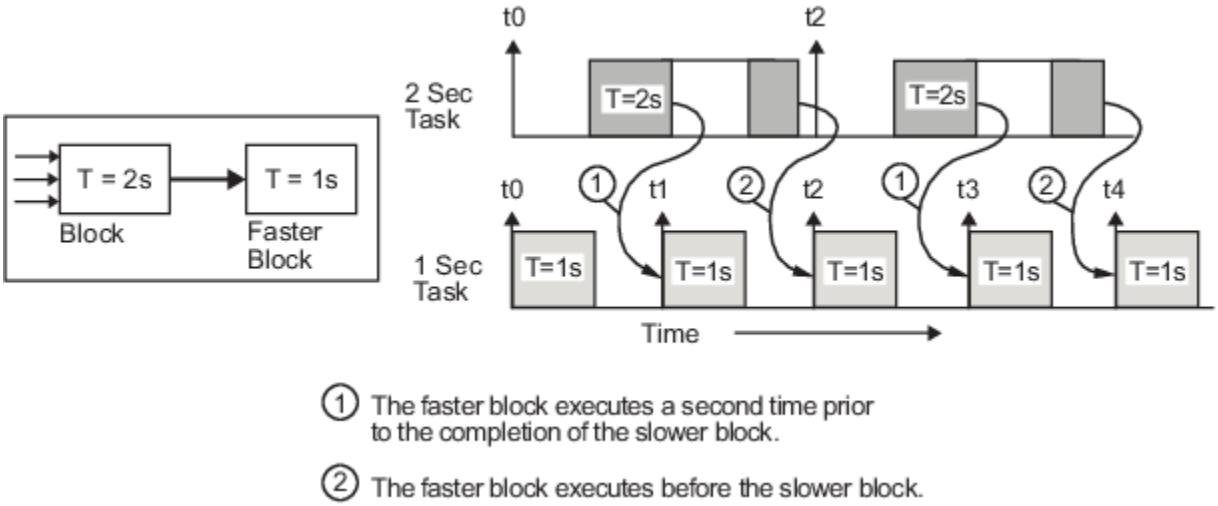
The next figure shows the execution sequence.



As you can see from the preceding figures, the Simulink engine can simulate models with multiple sample rates in an efficient manner. However, a Simulink simulation does not operate in real time.

Slower to Faster Transitions in Real Time

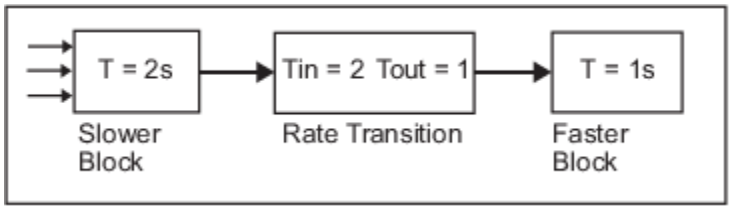
In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.



This timing diagram illustrates two problems:

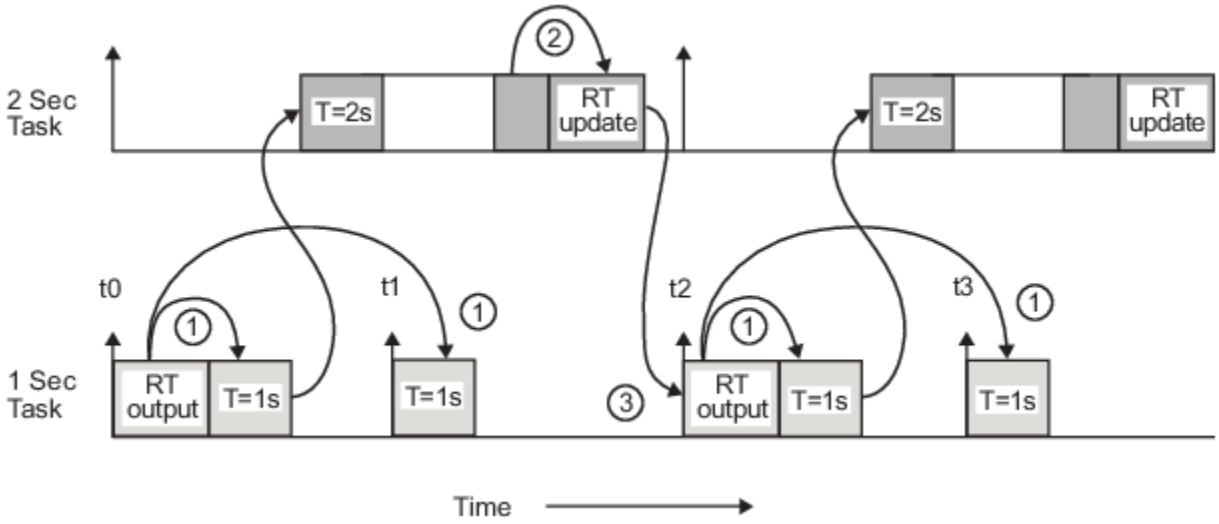
- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the faster task can have incorrect values some of the time.
- The faster block executes before the slower block (which is backward from the way a Simulink simulation operates). In this case, the 1 second block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

To eliminate these problems, you must insert a Rate Transition block between the slower and faster blocks.



It is assumed that the Rate Transition block is used in its default (protected/deterministic) mode.

The next figure shows the timing sequence that results with the added Rate Transition block.



Three key points about transitions in this diagram (refer to circled numbers):

1. The Rate Transition block output runs in the 1 second task, but at a slower rate (2 seconds). The output of the Rate Transition block feeds the 1 second task blocks.
2. The Rate Transition update uses the output of the 2 second task to update its internal state.
3. The Rate Transition output in the 1 second task uses the state of the Rate Transition that was updated in the 2 second task.

The first problem is addressed because the Rate Transition block is updating at a slower rate and at the priority of the slower block. The input to the Rate Transition block (which is the output of the slower block) is read after the slower block completes executing.

The second problem is addressed because the Rate Transition block executes at a slower rate and its output does not change during the computation of the faster block it is driving. The output portion of a Rate Transition block is executed at the sample rate of the slower block, but with the priority of the faster block. Since the Rate Transition block drives the faster block and has effectively the same priority, it is executed before the faster block.

Note This use of the Rate Transition block changes the model. The output of the slower block is now delayed by one time step compared to the output without a Rate Transition block.

Protect Data Integrity with volatile Keyword

When you select model configuration parameter **Ensure data integrity during data transfer**, the code generated for a Rate Transition block defines global buffers and semaphores and uses them to protect the integrity of the transferred data.

Particularly for a multitasking application, the tasks (rates) involved in a data transfer can write to the transferred data, buffers, and semaphores at times that your compiler cannot anticipate. To prevent your compiler from optimizing the assembly code in a manner that compromises the integrity of the transferred data, the code generator applies the keyword `volatile` to the buffers and semaphores. The code generator does not apply `volatile` to the global variable that represents the transferred data because the `volatile` buffers and semaphores typically offer enough protection.

With Embedded Coder, you can explicitly apply `volatile` to the transferred data by applying the built-in custom storage class `Volatile` to the input of the Rate Transition block. For example, you can use this technique to help protect the integrity of data that your external code shares with the generated code.

Alternatively, to protect data that your external code shares with the generated code, you can write your own C functions that read and write the data in a protected manner. Then, you can apply the custom storage class `GetSet` to the data in the model, which causes the generated code to call your functions instead of directly accessing the data.

For more information about applying `volatile`, see “Protect Global Data with const and volatile Type Qualifiers” on page 45-13. For more information about `GetSet`, see “Access Data Through Functions with Storage Class GetSet” on page 40-25.

Separate Rate Transition Block Code and Data from Algorithm Code and Data

You can specify whether the code generator inlines code and data it produces for Rate Transition blocks with model code or places the code and data in separate functions that the model code calls. You control this by selecting the **Rate Transition block code** parameter. Separating Rate Transition block code and data from algorithm code and data enables you to independently analyze, optimize, and test Rate Transition block and algorithm code. By default, Rate Transition block code is set inline with algorithm code and data. You can separate the code and data so that the generated code contains separate `get` and `set` functions that the *`model`*`_step` functions call and a dedicated structure for state data. The generated code also contains separate `start` and `initialize` functions that the *`model`*`_initialize` function calls.

Example Model

Open the example model `rtwdemo_ratetrans`. This multirate, multitasking model contains several Rate Transition blocks operating at different modes.

```
open_system('rtwdemo_ratetrans');
set_param('rtwdemo_ratetrans','SystemTargetFile','ert.tlc');
set_param('rtwdemo_ratetrans','GenerateComments', 'Off');
```

Separate Code for Rate Transition blocks

In the Configuration Parameters dialog box, the **Rate Transition block code** parameter is set to `Function`. Generate code for the model. The code is in the files `rtwdemo_ratetrans.c` and `rtwdemo_ratetrans.h`.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
slbuild('rtwdemo_ratetrans');
```

```
### Starting build procedure for: rtwdemo_ratetrans
### Successful completion of build procedure for: rtwdemo_ratetrans

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
rtwdemo_ratetrans  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 21.251s
```

```
hfile=fullfile(cgDir, 'rtwdemo_ratetrans_ert_rtw','rtwdemo_ratetrans.h');
rtwdemodbtype(hfile,'typedef struct {','} DW;', 1, 1);
```

```
typedef struct {
  real_T OutportBufferForOut3[20];
  real_T Integrator1_DSTATE[20];
  real_T Integrator2_DSTATE[20];
  real_T Integrator3_DSTATE[20];
  real_T Integrator1_PREV_U[20];
  real_T Integrator2_PREV_U[20];
  real_T Integrator3_PREV_U[20];
  uint32_T Algorithm_PREV_T;
  struct {
    uint_T Algorithm_RESET_ELAPS_T:1;
  } bitsForTID1;

  uint8_T Integrator1_SYSTEM_ENABLE;
  uint8_T Integrator2_SYSTEM_ENABLE;
  uint8_T Integrator3_SYSTEM_ENABLE;
} DW;
```

For Rate Transition blocks, the state data is not in the global state structure, `DW_rtwdemo_ratetrans_T`. This data is in its own structure in the file `rtwdemo_ratetrans_rtb.h`.

This code is in the file `rtwdemo_ratetrans.c`.

```
cfile=fullfile(cgDir, 'rtwdemo_ratetrans_ert_rtw','rtwdemo_ratetrans.c');
rtwdemodbtype(cfile,'void rtwdemo_ratetrans_step0','void rtwdemo_ratetrans_terminate(void)', 1, (
```

```
void rtwdemo_ratetrans_step0(void)
{
  (rtM->Timing.RateInteraction.TID0_1)++;
  if ((rtM->Timing.RateInteraction.TID0_1) > 1) {
```

```
    rtM->Timing.RateInteraction.TID0_1 = 0;
  }

  rtwdemo_rate_DetAndIntegS2F_get(rtY.Out1);
  rtwdemo_ratetr_IntegOnlyS2F_get(rtY.Out2);
  memcpy(&rtY.Out3[0], &rtDW.OutportBufferForOut3[0], 20U * sizeof(real_T));
  rtwdemo_rate_DetAndIntegF2S_set(rtU.In1);
  rtwdemo_ratetr_IntegOnlyF2S_set(rtU.In2);
}

void rtwdemo_ratetrans_step1(void)
{
  real_T rtb_DetAndIntegF2S[20];
  real_T rtb_IntegOnlyF2S[20];
  real_T Integrator3_DSTATE;
  real_T tmp;
  int32_T i;
  uint32_T Algorithm_ELAPS_T;
  rtwdemo_rate_DetAndIntegF2S_get(rtb_DetAndIntegF2S);
  rtwdemo_ratetr_IntegOnlyF2S_get(rtb_IntegOnlyF2S);
  if (rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T) {
    Algorithm_ELAPS_T = 0U;
  } else {
    Algorithm_ELAPS_T = rtM->Timing.clockTick1 - rtDW.Algorithm_PREV_T;
  }

  rtDW.Algorithm_PREV_T = rtM->Timing.clockTick1;
  rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T = false;
  tmp = 0.001 * (real_T)Algorithm_ELAPS_T;
  for (i = 0; i < 20; i++) {
    if (rtDW.Integrator1_SYSTEM_ENABLE == 0) {
      rtDW.Integrator1_DSTATE[i] += tmp * rtDW.Integrator1_PREV_U[i];
    }

    if (rtDW.Integrator2_SYSTEM_ENABLE == 0) {
      rtDW.Integrator2_DSTATE[i] += tmp * rtDW.Integrator2_PREV_U[i];
    }

    Integrator3_DSTATE = rtDW.Integrator3_DSTATE[i];
    if (rtDW.Integrator3_SYSTEM_ENABLE == 0) {
      Integrator3_DSTATE += tmp * rtDW.Integrator3_PREV_U[i];
    }

    rtDW.Integrator3_DSTATE[i] = Integrator3_DSTATE;
    rtDW.OutportBufferForOut3[i] = Integrator3_DSTATE;
    rtDW.Integrator1_PREV_U[i] = rtb_DetAndIntegF2S[i];
    rtDW.Integrator2_PREV_U[i] = rtb_IntegOnlyF2S[i];
    rtDW.Integrator3_PREV_U[i] = rtU.In3[i];
  }

  rtDW.Integrator1_SYSTEM_ENABLE = 0U;
  rtDW.Integrator2_SYSTEM_ENABLE = 0U;
  rtDW.Integrator3_SYSTEM_ENABLE = 0U;
  rtwdemo_rate_DetAndIntegS2F_set(rtDW.Integrator1_DSTATE);
  rtwdemo_ratetr_IntegOnlyS2F_set(rtDW.Integrator2_DSTATE);
  rtM->Timing.clockTick1++;
}
```

```
void rtwdemo_ratetrans_initialize(void)
{
  rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T = true;
  rtDW.Integrator1_SYSTEM_ENABLE = 1U;
  rtDW.Integrator2_SYSTEM_ENABLE = 1U;
  rtDW.Integrator3_SYSTEM_ENABLE = 1U;
}
```

The `rtwdemo_ratetrans_step0` and `rtwdemo_ratetrans_step1` functions contain calls to `get` and `set` functions. These functions contain the Rate Transition block code. These function definitions are in the file `rtwdemo_ratetrans_rtb.c`.

Generate Inlined Code for Rate Transition blocks

In the Configuration Parameters dialog box, set the **Rate Transition block code** parameter to `Inline`. Generate code for the model. The code is in the files `rtwdemo_ratetrans.c` and `rtwdemo_ratetrans.h`.

```
set_param('rtwdemo_ratetrans','RateTransitionBlockCode','Inline');
slbuild('rtwdemo_ratetrans')
```

```
### Starting build procedure for: rtwdemo_ratetrans
### Successful completion of build procedure for: rtwdemo_ratetrans

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
rtwdemo_ratetrans  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.388s
```

This code is now in the file `rtwdemo_ratetrans.h`.

```
hfile=fullfile(cgDir, 'rtwdemo_ratetrans_ert_rtw','rtwdemo_ratetrans.h');
rtwdemodbtype(hfile, 'typedef struct {', '} DW;', 1, 1);
```

```
typedef struct {
  real_T Integrator1_DSTATE[20];
  real_T Integrator2_DSTATE[20];
  real_T Integrator3_DSTATE[20];
  real_T DetAndIntegS2F_Buffer0[20];
  volatile real_T IntegOnlyS2F_Buffer[40];
  real_T DetAndIntegF2S_Buffer[20];
  volatile real_T IntegOnlyF2S_Buffer0[20];
  real_T Integrator1_PREV_U[20];
  real_T Integrator2_PREV_U[20];
  real_T Integrator3_PREV_U[20];
  uint32_T Algorithm_PREV_T;
  struct {
    uint_T Algorithm_RESET_ELAPS_T:1;
  } bitsForTID1;

  volatile int8_T IntegOnlyS2F_ActiveBufIdx;
```

```
  volatile int8_T IntegOnlyF2S_semaphoreTaken;
  uint8_T Integrator1_SYSTEM_ENABLE;
  uint8_T Integrator2_SYSTEM_ENABLE;
  uint8_T Integrator3_SYSTEM_ENABLE;
} DW;
```

For Rate Transition blocks, the state data is not in the global state structure, `DW_rtwdemo_ratetrans_T`. This data is in its own structure in the file `rtwdemo_ratetrans_rtb.h`.

This code is now in the file `rtwdemo_ratetrans_rtb.c`.

```
cfile=fullfile(cgDir, 'rtwdemo_ratetrans_ert_rtw','rtwdemo_ratetrans.c');
rtwdemodbtype(cfile,'void rtwdemo_ratetrans_step0','void rtwdemo_ratetrans_terminate(void)', 1, (
```

```
void rtwdemo_ratetrans_step0(void)
{
  int32_T i;
  int32_T i_0;
  (rtM->Timing.RateInteraction.TID0_1)++;
  if ((rtM->Timing.RateInteraction.TID0_1) > 1) {
    rtM->Timing.RateInteraction.TID0_1 = 0;
  }

  if (rtM->Timing.RateInteraction.TID0_1 == 1) {
    memcpy(&rtY.Out1[0], &rtDW.DetAndIntegS2F_Buffer0[0], 20U * sizeof(real_T));
  }

  i = rtDW.IntegOnlyS2F_ActiveBufIdx * 20;
  for (i_0 = 0; i_0 < 20; i_0++) {
    rtY.Out2[i_0] = rtDW.IntegOnlyS2F_Buffer[i_0 + i];
  }

  if (rtM->Timing.RateInteraction.TID0_1 == 1) {
    memcpy(&rtDW.DetAndIntegF2S_Buffer[0], &rtU.In1[0], 20U * sizeof(real_T));
  }

  if (rtDW.IntegOnlyF2S_semaphoreTaken == 0) {
    for (i = 0; i < 20; i++) {
      rtDW.IntegOnlyF2S_Buffer0[i] = rtU.In2[i];
    }
  }
}

void rtwdemo_ratetrans_step1(void)
{
  real_T rtb_IntegOnlyF2S[20];
  real_T Integrator1_DSTATE;
  real_T Out3;
  real_T tmp;
  int32_T i;
  uint32_T Algorithm_ELAPS_T;
  rtDW.IntegOnlyF2S_semaphoreTaken = 1;
  for (i = 0; i < 20; i++) {
    rtb_IntegOnlyF2S[i] = rtDW.IntegOnlyF2S_Buffer0[i];
  }

  rtDW.IntegOnlyF2S_semaphoreTaken = 0;
```

```
  if (rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T) {
    Algorithm_ELAPS_T = 0U;
  } else {
    Algorithm_ELAPS_T = rtM->Timing.clockTick1 - rtDW.Algorithm_PREV_T;
  }

  rtDW.Algorithm_PREV_T = rtM->Timing.clockTick1;
  rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T = false;
  tmp = 0.001 * (real_T)Algorithm_ELAPS_T;
  for (i = 0; i < 20; i++) {
    Integrator1_DSTATE = rtDW.Integrator1_DSTATE[i];
    if (rtDW.Integrator1_SYSTEM_ENABLE == 0) {
      Integrator1_DSTATE += tmp * rtDW.Integrator1_PREV_U[i];
    }

    rtDW.Integrator1_DSTATE[i] = Integrator1_DSTATE;
    if (rtDW.Integrator2_SYSTEM_ENABLE == 0) {
      rtDW.Integrator2_DSTATE[i] += tmp * rtDW.Integrator2_PREV_U[i];
    }

    if (rtDW.Integrator3_SYSTEM_ENABLE != 0) {
      Out3 = rtDW.Integrator3_DSTATE[i];
    } else {
      Out3 = tmp * rtDW.Integrator3_PREV_U[i] + rtDW.Integrator3_DSTATE[i];
    }

    rtY.Out3[i] = Out3;
    rtDW.Integrator1_PREV_U[i] = rtDW.DetAndIntegF2S_Buffer[i];
    rtDW.Integrator2_PREV_U[i] = rtb_IntegOnlyF2S[i];
    rtDW.Integrator3_DSTATE[i] = Out3;
    rtDW.Integrator3_PREV_U[i] = rtU.In3[i];
    rtDW.DetAndIntegS2F_Buffer0[i] = Integrator1_DSTATE;
  }

  rtDW.Integrator1_SYSTEM_ENABLE = 0U;
  rtDW.Integrator2_SYSTEM_ENABLE = 0U;
  rtDW.Integrator3_SYSTEM_ENABLE = 0U;
  for (i = 0; i < 20; i++) {
    rtDW.IntegOnlyS2F_Buffer[i + (rtDW.IntegOnlyS2F_ActiveBufIdx == 0) * 20] =
      rtDW.Integrator2_DSTATE[i];
  }

  rtDW.IntegOnlyS2F_ActiveBufIdx = (int8_T)(rtDW.IntegOnlyS2F_ActiveBufIdx == 0);
  rtM->Timing.clockTick1++;
}

void rtwdemo_ratetrans_initialize(void)
{
  rtDW.bitsForTID1.Algorithm_RESET_ELAPS_T = true;
  rtDW.Integrator1_SYSTEM_ENABLE = 1U;
  rtDW.Integrator2_SYSTEM_ENABLE = 1U;
  rtDW.Integrator3_SYSTEM_ENABLE = 1U;
}
```

The code is inlined in functions `rtwdemo_ratetrans_step0` and `rtwdemo_ratetrans_step1`.

Limitation

The code generator does not separate the code and data for Rate Transition blocks that have variable-size signals or are inside of a For Each Subsystem block.

See Also

- Rate Transition block code

```
bdclose('rtwdemo_ratetrans');
rtwdemoclean;
cd(currentDir)
```

See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Modeling for Multitasking Execution”
- “Configure Time-Based Scheduling”
- “Resolve Rate Transitions”
- “Time-Based Scheduling Example Models”

Configure Time-Based Scheduling

For details about solver options, see “Solver Pane”.

Configure Start and Stop Times

The value of model configuration parameter **Stop time** must be greater than or equal to the value of parameter **Start time**. If the stop time is zero, or if the total simulation time (`Stop` minus `Start`) is less than zero, the generated program runs for one step. If the stop time is set to `inf`, the generated program runs indefinitely.

When using the GRT or ERT targets, you can override the stop time when running a generated program from the Microsoft Windows command prompt or UNIX[4] command line. To override the stop time that was set during code generation, use the `-tf` switch.

```
model -tf n
```

The program runs for `n` seconds. If `n = inf`, the program runs indefinitely.

Certain blocks have a dependency on absolute time. If you are designing a program that is intended to run indefinitely (model configuration parameter **Stop time** = `inf`), and your generated code does not use the `rtModel` data structure (that is, it uses `simstructs` instead), you must not use these blocks. See “Absolute Time Limitations” on page 28-14 for a list of blocks that can potentially overflow timers.

If you know how long an application that depends on absolute time needs to run, you can prevent the timers from overflowing and force the use of optimal word sizes by selecting model configuration parameter **Application lifespan (days)**. See “Control Memory Allocation for Time Counters” on page 76-10 for details.

Configure the Solver Type

For code generation, you must configure a model to use a fixed-step solver for system target files except for S-function and RSim system target files. You can configure the S-function and RSim targets with a fixed-step or variable-step solver.

Configure the Tasking Mode

The code generator supports both single-tasking and multitasking modes for periodic sample times. See “Time-Based Scheduling and Code Generation” on page 29-2 for details.

See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Sample Times in Subsystems”
- “Sample Times in Systems”

4 UNIX is a registered trademark of The Open Group in the United States and other countries.

- “Time-Based Scheduling Example Models”

Time-Based Scheduling Example Models

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

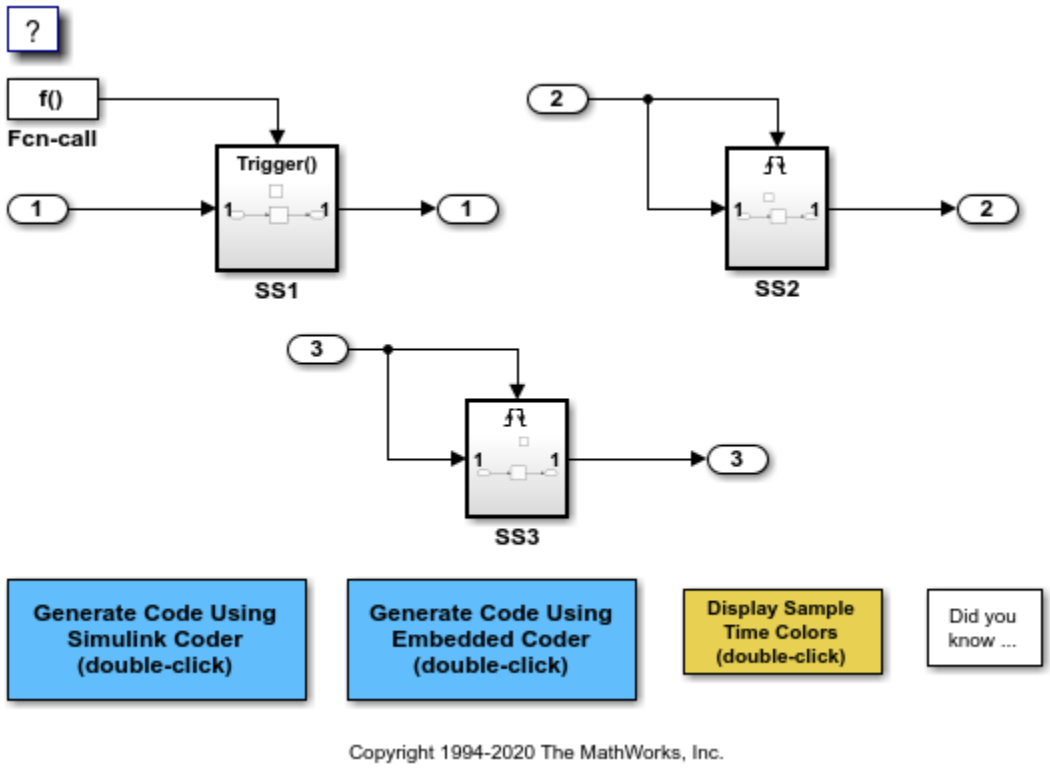
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



The model consists of three subsystems `SS1`, `SS2`, and `SS3`. Open the Model Configuration Parameters dialog box. On the **Math and Data Types** pane, the **Application lifespan (days)** parameter is set to the default, which is `inf`.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.

2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.

3. Build the model.

```
### Starting build procedure for: rtwdemo_abstime
### Successful completion of build procedure for: rtwdemo_abstime

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================================
rtwdemo_abstime  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.921s
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.h`.

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint32_T clockTickH1;
    uint32_T clockTick2;
    uint32_T clockTickH2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};

/* Block states (default storage) */
extern DW_rtwdemo_abstime_T rtwdemo_abstime_DW;

/* Zero-crossing (trigger) state */
extern PrevZCX_rtwdemo_abstime_T rtwdemo_abstime_PrevZCX;

/* External inputs (root inport signals with default storage) */
extern ExtU_rtwdemo_abstime_T rtwdemo_abstime_U;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY_rtwdemo_abstime_T rtwdemo_abstime_Y;

/* Model entry point functions */
extern void rtwdemo_abstime_initialize(void);
extern void rtwdemo_abstime_step0(void);
extern void rtwdemo_abstime_step1(void);
extern void rtwdemo_abstime_step2(void);
```

```
extern void rtwdemo_abstime_terminate(void);

/* Real-time Model object */
extern RT_MODEL_rtwdemo_abstime_T *const rtwdemo_abstime_M;

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model.  The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model.  For example,
 *
 * hilite_system('<S3>')    - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'rtwdemo_abstime'
 * '<S1>'   : 'rtwdemo_abstime/SS1'
 * '<S2>'   : 'rtwdemo_abstime/SS2'
 * '<S3>'   : 'rtwdemo_abstime/SS3'
 */
#endif                                /* RTW_HEADER_rtwdemo_abstime_h_ */
```

Four 32-bit unsigned integers, `clockTick1` , `clockTickH1` , `clockTick2` , and `clockTickH2` are counters for storing the elapsed time of subsystems `SS2` and `SS3`.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.

2. Build the model.

```
### Starting build procedure for: rtwdemo_abstime
### Successful completion of build procedure for: rtwdemo_abstime

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
==============================================================================
rtwdemo_abstime  Code generated and compiled.  Incremental checksum changed.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.449s
```

Review the Regenerated Code

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
```

```
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint16_T clockTick2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};

/* Block states (default storage) */
extern DW_rtwdemo_abstime_T rtwdemo_abstime_DW;

/* Zero-crossing (trigger) state */
extern PrevZCX_rtwdemo_abstime_T rtwdemo_abstime_PrevZCX;

/* External inputs (root inport signals with default storage) */
extern ExtU_rtwdemo_abstime_T rtwdemo_abstime_U;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY_rtwdemo_abstime_T rtwdemo_abstime_Y;

/* Model entry point functions */
extern void rtwdemo_abstime_initialize(void);
extern void rtwdemo_abstime_step0(void);
extern void rtwdemo_abstime_step1(void);
extern void rtwdemo_abstime_step2(void);
extern void rtwdemo_abstime_terminate(void);

/* Real-time Model object */
extern RT_MODEL_rtwdemo_abstime_T *const rtwdemo_abstime_M;

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model.  The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model.  For example,
 *
 * hilite_system('<S3>')    - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'rtwdemo_abstime'
 * '<S1>'   : 'rtwdemo_abstime/SS1'
 * '<S2>'   : 'rtwdemo_abstime/SS2'
 * '<S3>'   : 'rtwdemo_abstime/SS3'
 */
#endif                                /* RTW_HEADER_rtwdemo_abstime_h_ */
```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for `SS2`
- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for `SS3`

Related Information

- "Model Configuration Parameters: Code Generation Optimization"
- "Timers in Asynchronous Tasks"
- "Time-Based Scheduling and Code Generation"

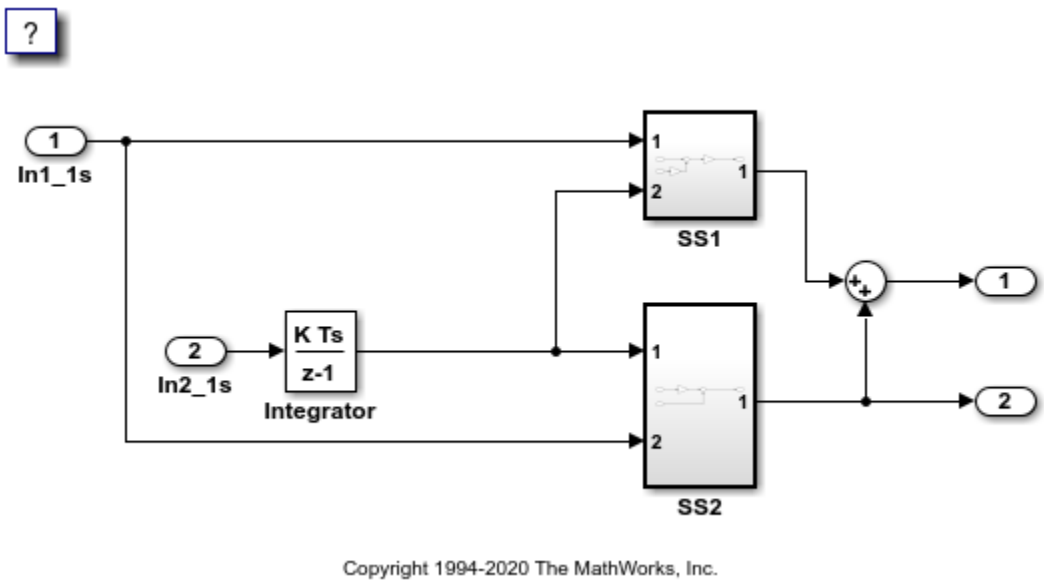
Single-Rate Modeling (Bare Board, No OS)

This model shows code generation for a single-rate discrete-time model configured for a bare-board target (one with no operating system).

Open Example Model

Open the example model `rtwdemo_srbb`.

```
open_system('rtwdemo_srbb')
```



The model uses one sample time and is configured to display sample-time colors when an update diagram occurs. Inport blocks In1_1s and In2_1s specify a 1-second sample time, which is enforced by the setting of model configuration parameter **Periodic sample time constraint**.

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**. Because the model is configured with one sample time, the model appears red, which is the color that represents the fastest discrete sample time in the model.

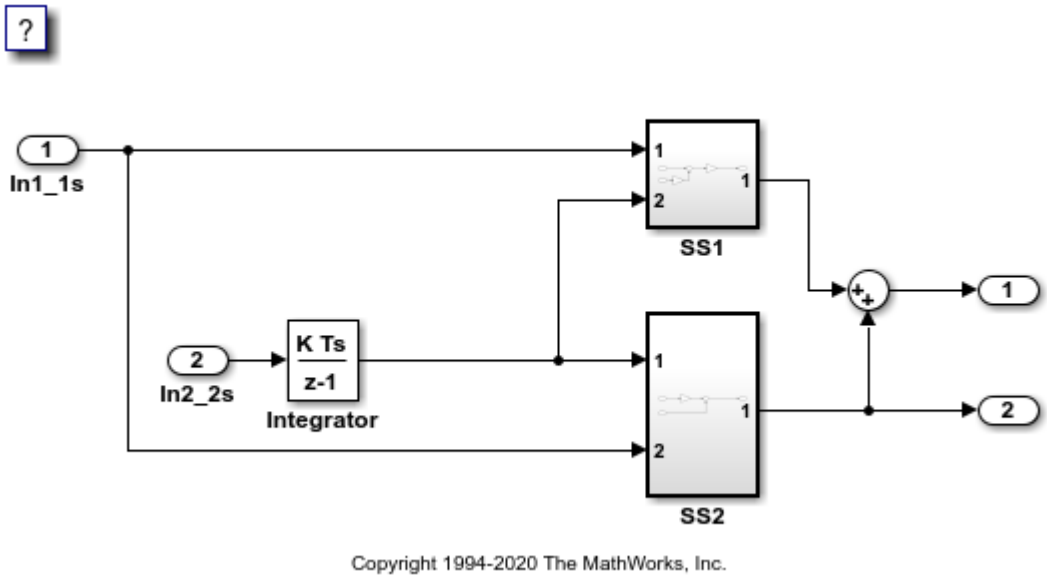
Multirate Modeling in Single-Tasking Mode (Bare Board, No OS)

This model shows the code generated for a multirate discrete-time model configured for single-tasking on a bare-board target (one with no operating system).

Open Example Model

Open the example model `rtwdemo_mrstbb`.

```
open_system('rtwdemo_mrstbb')
```



The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the setting of model configuration parameter **Periodic sample time constraint**. The solver is set for single-tasking operation. Rate transition blocks are, therefore, not included between blocks executing at different sample times because preemption will not occur.

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times.

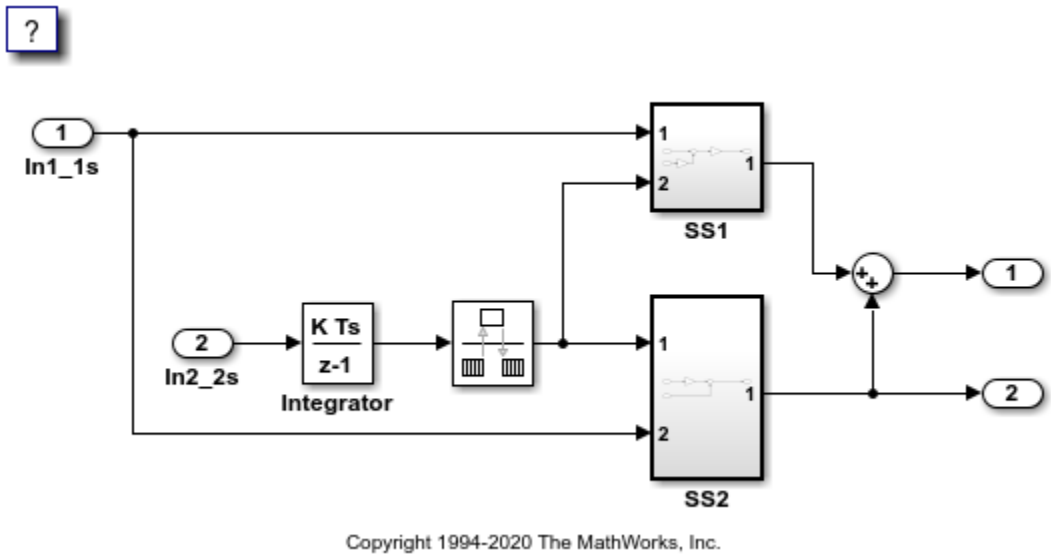
Multirate Modeling in Multitasking Mode (Bare Board, No OS)

This model shows the code generated for a multirate discrete-time model configured for a multitasking bare-board target (one with no operating system).

Open Example Model

Open the example model `rtwdemo_mrmtbb`.

```
open_system('rtwdemo_mrmtbb')
```



Explore Example Model

The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the setting of model configuration parameter **Periodic sample time constraint**. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink® and Simulink Coder™. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink setting model configuration parameter **Automatically handle rate transition for data transfer**.

The model is configured to display sample-time colors upon diagram update. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times.

Data Transfer Assumptions

Basis of operation for data transfers between tasks:

- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash and restart (especially while data is being transferred between tasks)

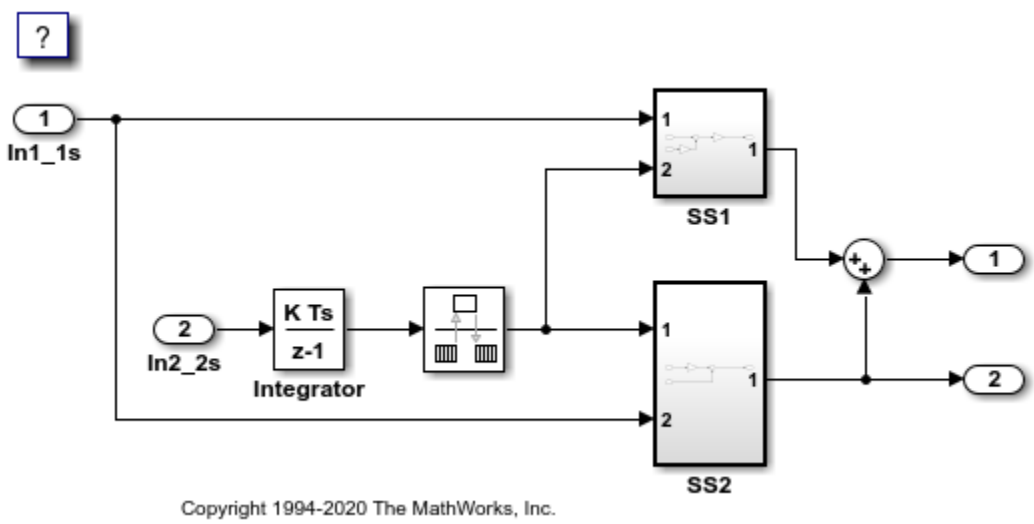
Multirate Modeling in Multitasking Mode (VxWorks® OS)

This example generates code for a multirate discrete-time model configured for a multitasking operating system target (VxWorks®). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the setting of model configuration parameter **Periodic sample time constraint**. The solver is set for multitasking operation, which means a Rate Transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Simulink® and the code generator enforce proper rate transitions. This model specifies an explicit Rate Transition block. Alternatively, you can instruct Simulink® to insert this block for you by setting model configuration parameter **Automatically handle rate transition for data transfer**.

The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing Ctrl+D. To display the legend, press Ctrl+J. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times.

Example Model

```
model = 'rtwdemo_mrmtos';
open_system(model);
```



Trade Determinism and Data Integrity to Improve System Performance

This model shows the differences in the operation modes of the Rate Transition block when used in a multirate, multitasking model. The flexible options for the Rate Transition block allow you to select the mode that is best suited for your application. You can trade levels of determinism and data integrity to improve system performance.

Rate Transition Block Modes of Operation

Ensure data integrity and determinism (DetAndInteg): Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. Additionally, it is ensured that the relative sample time (delay) from which the data is transferred from one rate to another is always the same. Only ANSI® C code is used, no target specific 'critical section' protection is needed.

Ensure integrity (IntegOnly): Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. However, from one transfer of data to the next, the relative sample time (delay) for which the data is transferred can vary. In this mode, the code to read/write the data is run more often than in the DetandInt mode. In the worst case, the delay is equivalent to the DetandInt mode, but the delay can be less which is important is some applications. Also, this mode support data transfers to/from asynchronous rates which the DetandInt mode cannot support. Only ANSI-C code is used, no target specific 'critical section' protection is needed.

No data consistency operations are performed (None): For this case, the Rate Transition block does not generated code. This mode is acceptable in some application where atomic access of scalar data types is guaranteed and when the relative temporal values of the data is not important. This mode does not introduce any delay.

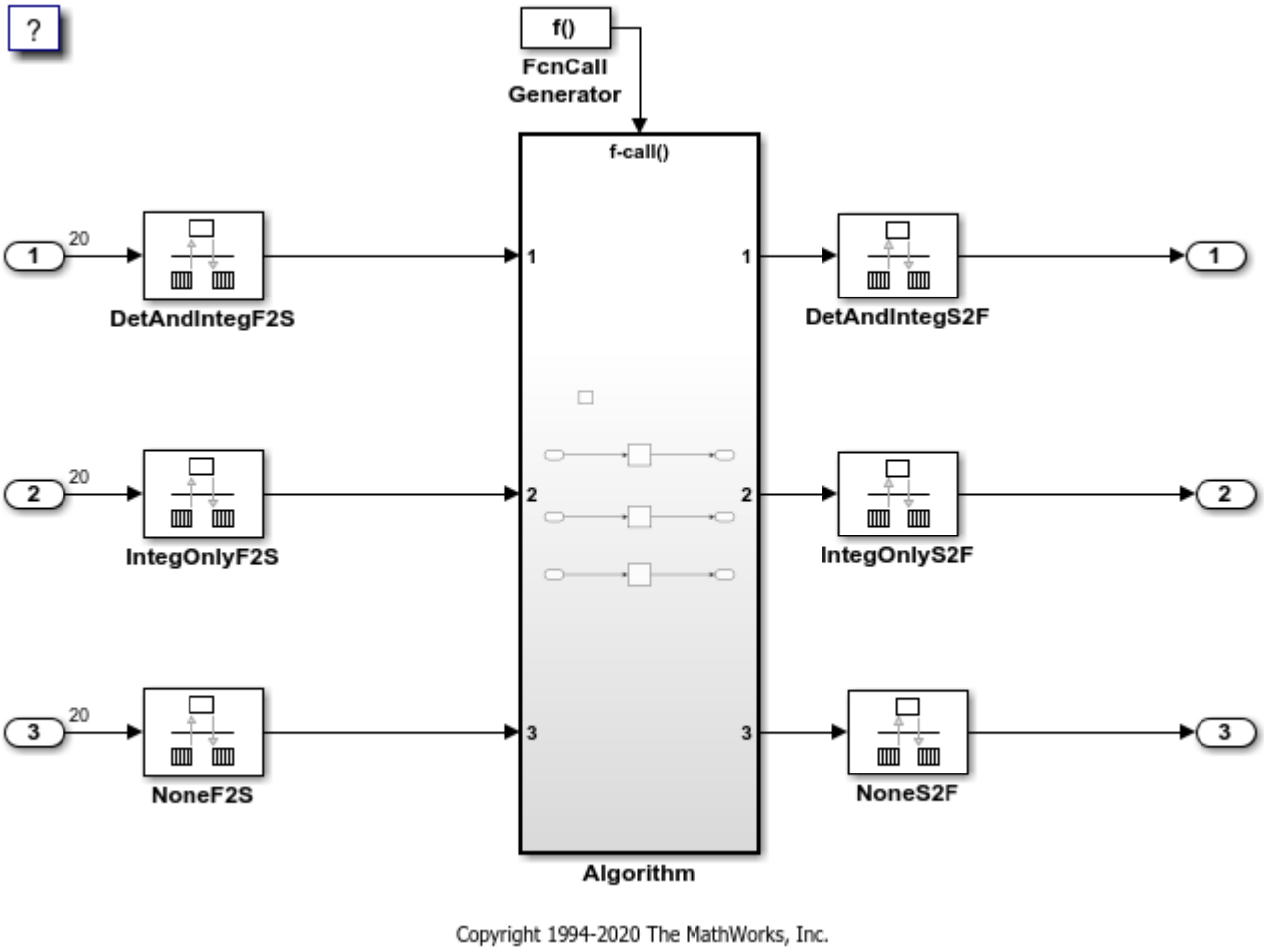
Data Transfer Assumptions

Basis of operation for data transfers between tasks:

- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash/restart (especially while data is being transferred between tasks).

Model `rtwdemo_ratetrans`

```
open_system('rtwdemo_ratetrans')
```



Model `rtwdemo_ratetrans` shows the differences in the operation modes of the following Rate Transition blocks.

Rate Transition Block `DetAndIntegF2S`

Determinism and data integrity (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to output at slower rate but done during the faster rate context.
- Data as seen by the slower rate is always the value when both the faster and slower rate last executed. Any subsequent steps by the faster rate (and associated data updates) while the slower rate is running are not seen by the slower rate.

Rate Transition Block `DetAndIntegS2F`

Determinism and data integrity (slow to fast transition):

- Uses two persistent data buffers, an internal buffer and the blocks output.
- The internal buffer is copied to the output at the slower rate but done during the faster rate context.

- The internal buffer is written at the slower rate and during the slower rate context.
- The data that Fast rate sees is always delayed, i.e. data is from the previous step of the slow rate code.

Rate Transition Block `IntegOnlyF2S`

Data integrity only (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to buffer during the faster rate context if a flag indicates it not in the process of being read.
- The flag is set and data is copied from the buffer to output at the slow rate, the flag is then cleared. This is an additional copy as compared to the deterministic case.
- Data as seen by the slower rate can be from a more recent step of the faster rate than from when the slower rate and faster rate both executed.

Rate Transition Block `IntegOnlyS2F`

Data integrity only (slow to fast transition):

- Uses two persistent data buffers, both are internal buffers.
- One of the 2 buffers is always copied to the output at faster rate.
- One of the 2 buffers is written at the slower rate and during the slower rate context, then the active buffer is switched.
- The data as seen by the faster rate can be more recent than for the deterministic case. Specifically, when both the slower and faster rate have their hits, the faster rate will see a previous value from the slower rate. But, subsequent steps for the faster rate may see an updated value (when the slower rate updates the non-active buffer and switches the active buffer flag.

Rate Transition Block `NoneF2S`

No code is generated for the Rate Transition block when determinism and data integrity is waived.

Rate Transition Block `NoneS2F`

No code is generated for the Rate Transition block when determinism and data integrity is waived.

```
bdclose('rtwdemo_ratetrans');
```

See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Modeling for Single-Tasking Execution”
- “Modeling for Multitasking Execution”

Event-Based Scheduling in Simulink Coder

- “Asynchronous Events” on page 30-2
- “Generate Interrupt Service Routines” on page 30-4
- “Spawn and Synchronize Execution of RTOS Task” on page 30-11
- “Pass Asynchronous Events in RTOS as Input to a Referenced Model” on page 30-28
- “Rate Transitions and Asynchronous Blocks” on page 30-34
- “Timers in Asynchronous Tasks” on page 30-38
- “Create a Customized Asynchronous Library” on page 30-40
- “Import Asynchronous Event Data for Simulation” on page 30-47
- “Asynchronous Support Limitations” on page 30-50

Asynchronous Events

Asynchronous Support

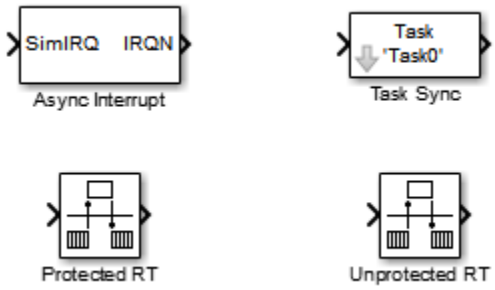
Normally, you time models from which you plan to generate code from a *periodic* interrupt source (for example, a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at multiples of that rate.

Many systems must also support execution of blocks in response to events that are *asynchronous* with respect to the periodic timing source of the system. For example, a peripheral device might signal completion of an input operation by generating an interrupt. The system must service such interrupts, for example, by acquiring data from the interrupting device.

This topic explains how to use blocks to model and generate code for asynchronous event handling, including servicing of hardware-generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under a real-time operating system (RTOS). This block library demonstrates integration with an example RTOS (VxWorks). Although the blocks target an example RTOS, this chapter provides source code analysis and other information you can use to develop blocks that support asynchronous event handling for an alternative target RTOS.[5]

Block Library for Calls to an Example Real-Time Operating System

The next figure shows the blocks in the `vxlib1` block library.



The key blocks in the library are the Async Interrupt and Task Sync blocks. These blocks are targeted for an example RTOS (VxWorks). You can use them, with modification, to support your RTOS applications.

Note Use the blocks in the Interrupt Templates block library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for a target environment.

To implement asynchronous support for an RTOS other than the example RTOS, use these guidelines and the example code in “Create a Customized Asynchronous Library” on page 30-40 to adapt the Interrupt Template library blocks for your RTOS.

5 VxWorks is a registered trademark of Wind River Systems, Inc.

The Interrupt Template library includes blocks that you can use to:

- Generate interrupt-level code — Async Interrupt block
- Spawn an RTOS task that calls a function call subsystem — Task Sync block

The use of protected and unprotected Rate Transition blocks in asynchronous contexts is discussed in “Rate Transitions and Asynchronous Blocks” on page 30-34. For general information on rate transitions, see “Time-Based Scheduling and Code Generation” on page 29-2.

Examples and Additional Information

- Example model `rtwdemo_async` uses the `tornado.tlc` system target file and Interrupt Template block library. To open the model, type `rtwdemo_async` at the MATLAB command prompt.
- Example model `rtwdemo_async_mdlreftop` uses the `tornado.tlc` system target file and Interrupt Template block library. To open the model, type `rtwdemo_async_mdlreftop` at the MATLAB command prompt.
- “Time-Based Scheduling and Code Generation” discusses general multitasking and rate transition issues for periodic models.
- The Embedded Coder documentation discusses the `ert.tlc` system target file, including task execution and scheduling.
- For information about the system calls to the example RTOS (VxWorks) mentioned in this topic, see VxWorks system documentation on the Wind River website.

See Also

More About

- “Time-Based Scheduling and Code Generation”
- “Generate Interrupt Service Routines”
- “Spawn and Synchronize Execution of RTOS Task”
- “Pass Asynchronous Events in RTOS as Input to a Referenced Model”
- “Timers in Asynchronous Tasks”
- “Import Asynchronous Event Data for Simulation”
- “Rate Transitions and Asynchronous Blocks”
- “Create a Customized Asynchronous Library”
- “Asynchronous Support Limitations”

Generate Interrupt Service Routines

To generate an interrupt service routine (ISR) associated with a specific VME interrupt level for an example RTOS (for example, VxWorks), use the Async Interrupt block. The Async Interrupt block enables the specified interrupt level and installs an ISR that calls a connected function call subsystem.

You can also use the Async Interrupt block in a simulation. It provides an input port that you can enable and connect to a simulated interrupt source.

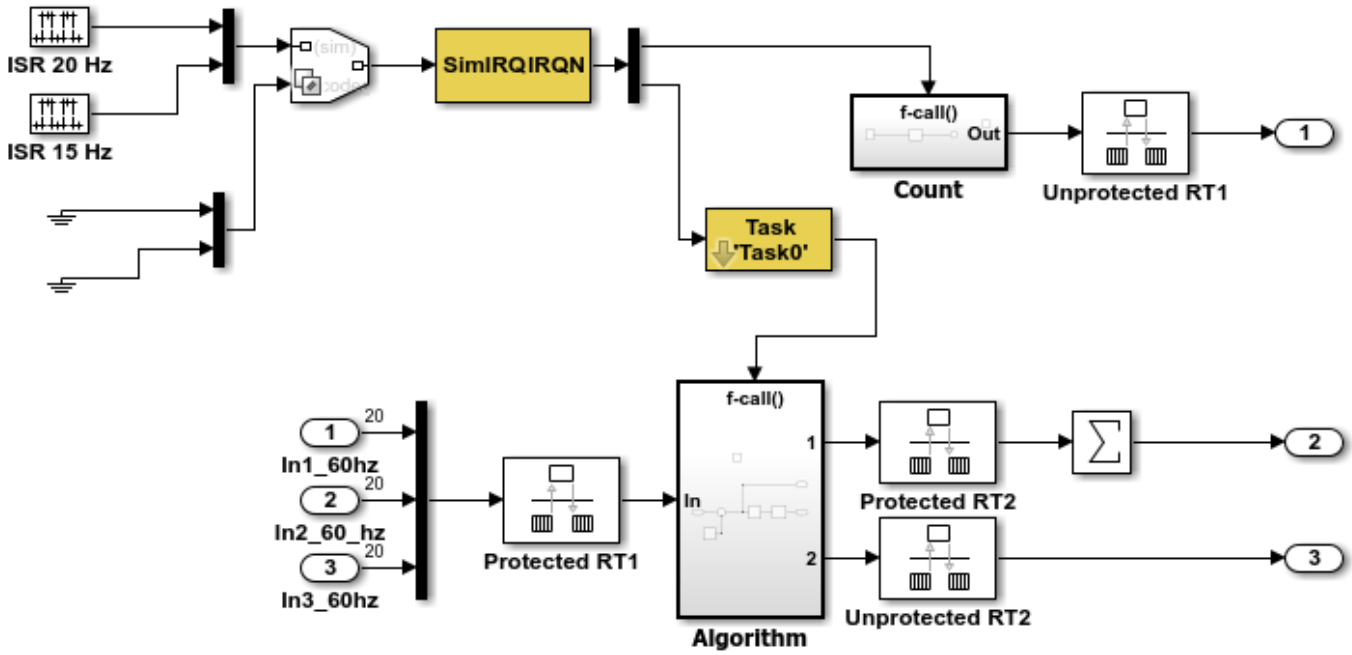
Note The operating system integration techniques that are demonstrated in this section use blocks that are in the Interrupt Templates block library. The blocks in that library provide starting point examples to help you develop custom blocks for a target environment.

Connecting the Async Interrupt Block

To generate an ISR, connect an output of the Async Interrupt block to the control input of:

- A function call subsystem
- The input of a Task Sync block
- The input to a Stateflow chart configured for a function call input event

Example model `rtwdemo_async` shows an Async Interrupt block that is configured to service two interrupt sources. Each of the two output signals is connected to a function call subsystem.



Requirements and Restrictions

- The Async Interrupt block supports VME interrupts 1 through 7.

- The Async Interrupt block uses these system calls to the example RTOS (VxWorks):
  - `sysIntEnable`
  - `sysIntDisable`
  - `intConnect`
  - `intLock`
  - `intUnlock`
  - `tickGet`

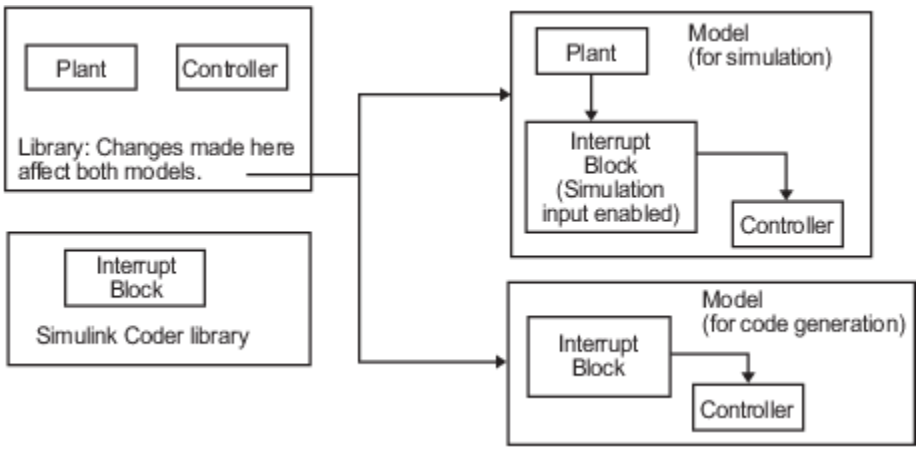
Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. It is best to keep ISRs as short as possible. Connect only function call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function call subsystem to a RTOS task. Place the Task Sync block between the Async Interrupt block and the function call subsystem. The Async Interrupt block installs the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The task is then scheduled and run by the RTOS (VxWorks). See “Spawn and Synchronize Execution of RTOS Task” on page 30-11 for more information.

Dual-Model Approach for Developing Real-Time Systems that Include ISRs

When developing a real-time system that includes ISRs, consider using a *dual-model* approach. Develop one model that includes a plant and a controller for simulation, and another model that includes only the controller for code generation. By using a Simulink library, you can implement changes to both models simultaneously. This figure shows how changes made to the plant or controller, both of which are in a library, are propagated to the models.

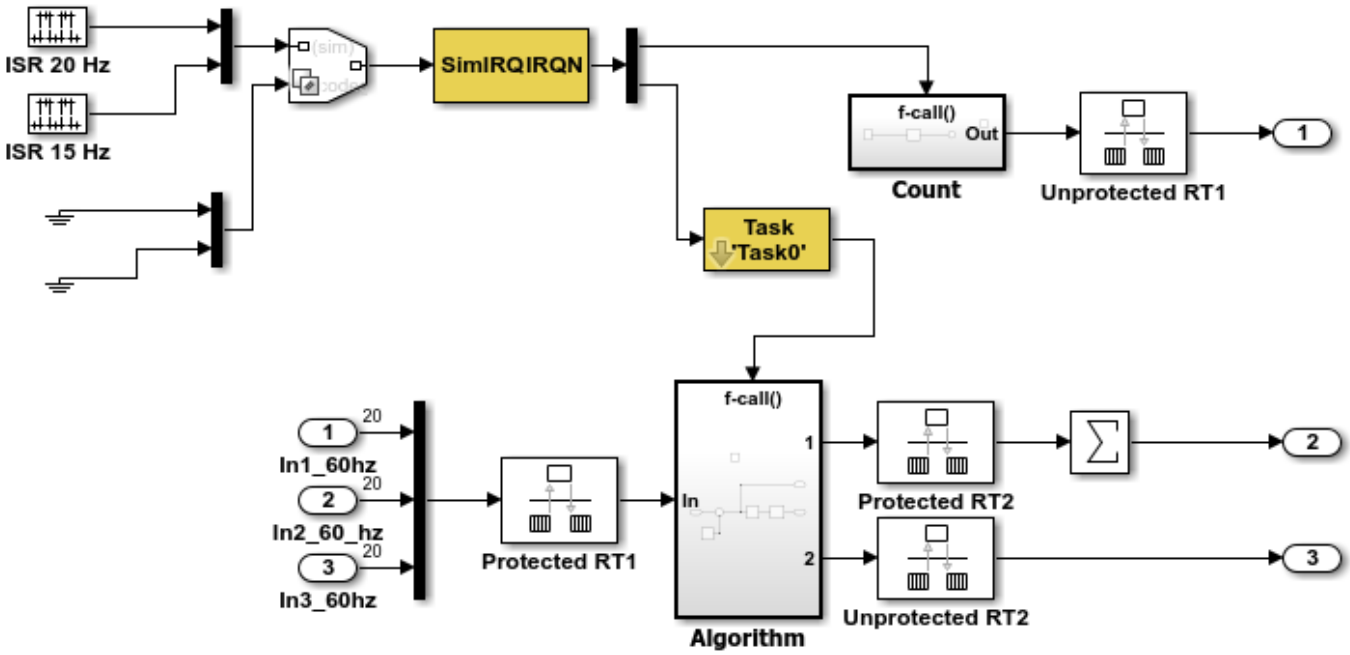


Dual-Model Use of Async Interrupt Block for Simulation and Code Generation

A *single-model* approach is also possible. The Plant component of the model is active only during simulation. For code generation, you switch out the Plant components out of the system and you generate code only for the interrupt block and controller parts of the model.

Dual-Model Approach: Simulation

Example model `rtwdemo_async` shows the dual-model approach to modeling. During simulation, the Pulse Generator blocks provide simulated interrupt signals.



During simulation:

- When the value of the input port is nonzero, Simulink imitates the behavior of each interrupt by calling the destination function unconditionally where the interrupt occurs.
- Subsystems connected to Async Interrupt block output ports execute in order of their priority in the RTOS (in this example, VxWorks). If two or more interrupt signals occur simultaneously, the Async Interrupt block executes the downstream systems in the order specified by their interrupt levels (level 7 gets the highest priority). The first input element maps to the first output element.

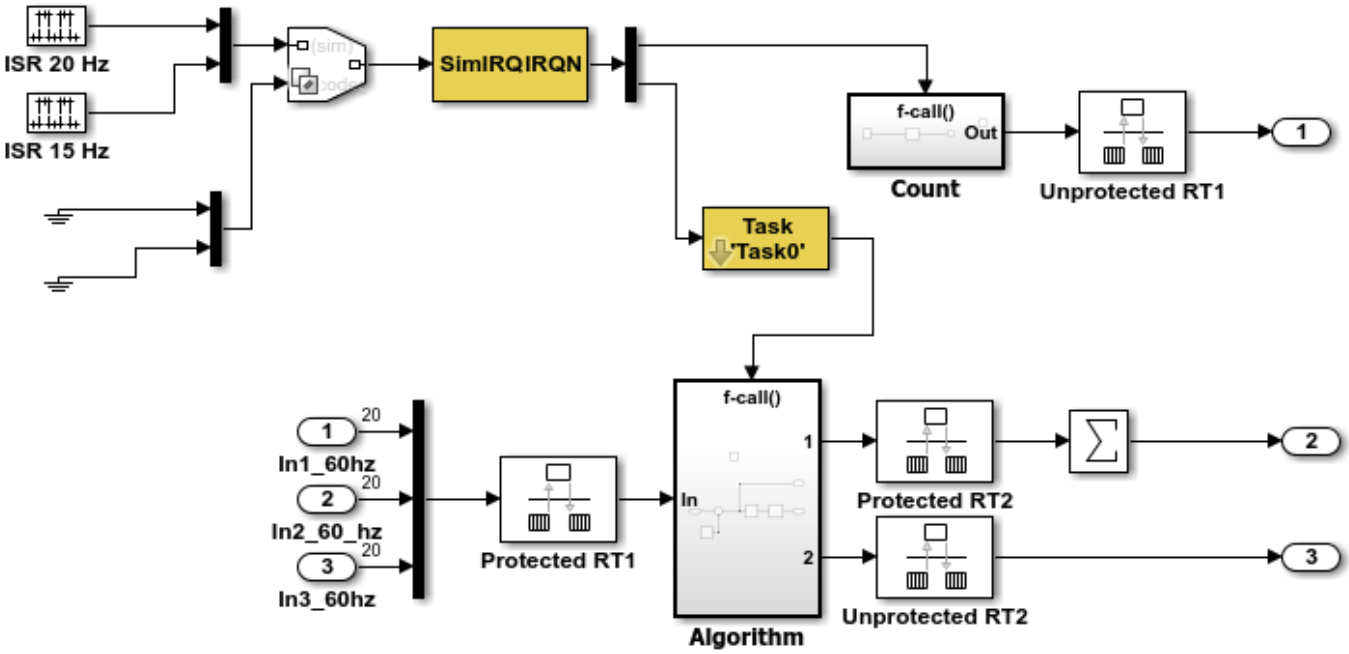
You can use the Async Interrupt block in a simulation without enabling the simulation input. The Async Interrupt block inherits the base rate of the model and calls the connected subsystems in order of their priorities in the RTOS. (In this case, the Async Interrupt block behaves as if inputs receive a 1 simultaneously.)

Dual-Model Approach: Code Generation

In the generated code for the example model:

- Ground blocks provide input signals to the Variant Source block
- The Async Interrupt block does not use its simulation input

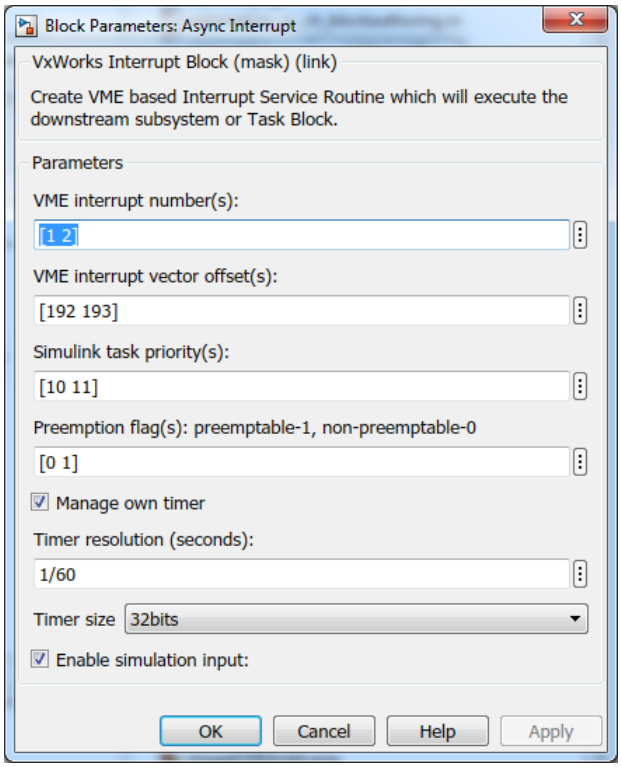
The Ground blocks drive control input of the Variant Source block, so the code generator does not produce code for that signal path. The code generator does not produce code for blocks that drive the simulation control input to the Variant Source block because that path is not selected during code generation. However, the sample times of driving blocks for the simulation input to the Variant Source block contribute to the sample times supported in the generated code. To avoid including unnecessary sample times in the generated code, use the sample times of the blocks driving the simulation input in the model where generated code is intended.



The code generator produces standalone functions for the ISRs and the interrupt vector table is as follows:

Offset	
192	`&isr_num1_vec192()`
193	`&isr_num2_vec193()`

Consider the code generated from this model, assuming that the Async Interrupt block parameters are configured as shown in this figure.



Initialization Code

During code generation, the Async Interrupt block installs interrupt code in the code generated for subsystem blocks as interrupt service routines. The interrupt vectors for `IRQ1` and `IRQ2` are stored at locations `192` and `193`, relative to the base of the interrupt vector table, as specified by the **VME interrupt vector offset(s)** parameter.

Installing an ISR requires two RTOS (VxWorks) calls, `int_connect` and `sysInt_Enable`. The Async Interrupt block inserts these calls in the *`model`*`_initialize` function, as shown in this code fragment.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Connect and enable ISR function: isr_num1_vec192 */
    if( intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
      printf("intConnect failed for ISR 1.\n");
    }
    sysIntEnable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Connect and enable ISR function: isr_num2_vec193 */
    if( intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK)
    {
      printf("intConnect failed for ISR 2.\n");
    }
    sysIntEnable(2);
```

The Async Interrupt block does not configure the hardware that generates the interrupt. Typically, the interrupt source is a VME I/O board, which generates interrupts for specific events (for example, at the end of an analog-to-digital conversion). You set up the VME interrupt level and vector in registers or by using jumpers on the board. You can use the `mdlStart` routine of a user-written device driver (S-function) to set up the registers and enable interrupt generation on the board. You

must match the interrupt level and vector specified in the Async Interrupt block parameter dialog box to the level and vector set up on the I/O board.

Generated ISR Code

The actual ISR generated for `IRQ1` in the RTOS (VxWorks) is listed below.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */

void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

   /* Use tickGet() as a portable tick counter example.
      A much higher resolution can be achieved with a
      hardware counter */
   Async_Code_M->Timing.clockTick2 = tickGet();

   /* disable interrupts (system is configured as non-ive) */
   lock = intLock();

   /* save floating point context */
   fppSave(&context);

   /* Call the system: <Root>/Subsystem A */
   Count(0, 0);

   /* restore floating point context */
   fppRestore(&context);

   /* re-enable interrupts */
   intUnlock(lock);
}
```

Note these features of the ISR:

- Because of the setting of the **Preemption Flag(s)** parameter, the ISR is locked; that is, it cannot be preempted by a higher priority interrupt. The ISR is locked and unlocked in the example RTOS (VxWorks) by calls to the functions `int_lock` and `int_unlock`.
- The connected subsystem, `Count`, is called from within the ISR.
- The `Count` function executes algorithmic (model) code. Therefore, the floating-point context is saved and restored across the call to `Count`.
- The ISR maintains its own absolute time counter, which is distinct from other periodic base rate or subrate counters in the system. Timing data is maintained for use by code generated for blocks executed within the ISR that require absolute or elapsed time.

  See “Timers in Asynchronous Tasks” on page 30-38 for details.

Model Termination Code

The model termination function disables the interrupts in the RTOS (VxWorks):

```
/* Model terminate function */
void Async_Code_terminate(void)
{
```

```
    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num1_vec192 */
    sysIntDisable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num2_vec193 */
    sysIntDisable(2);
}
```

See Also

More About

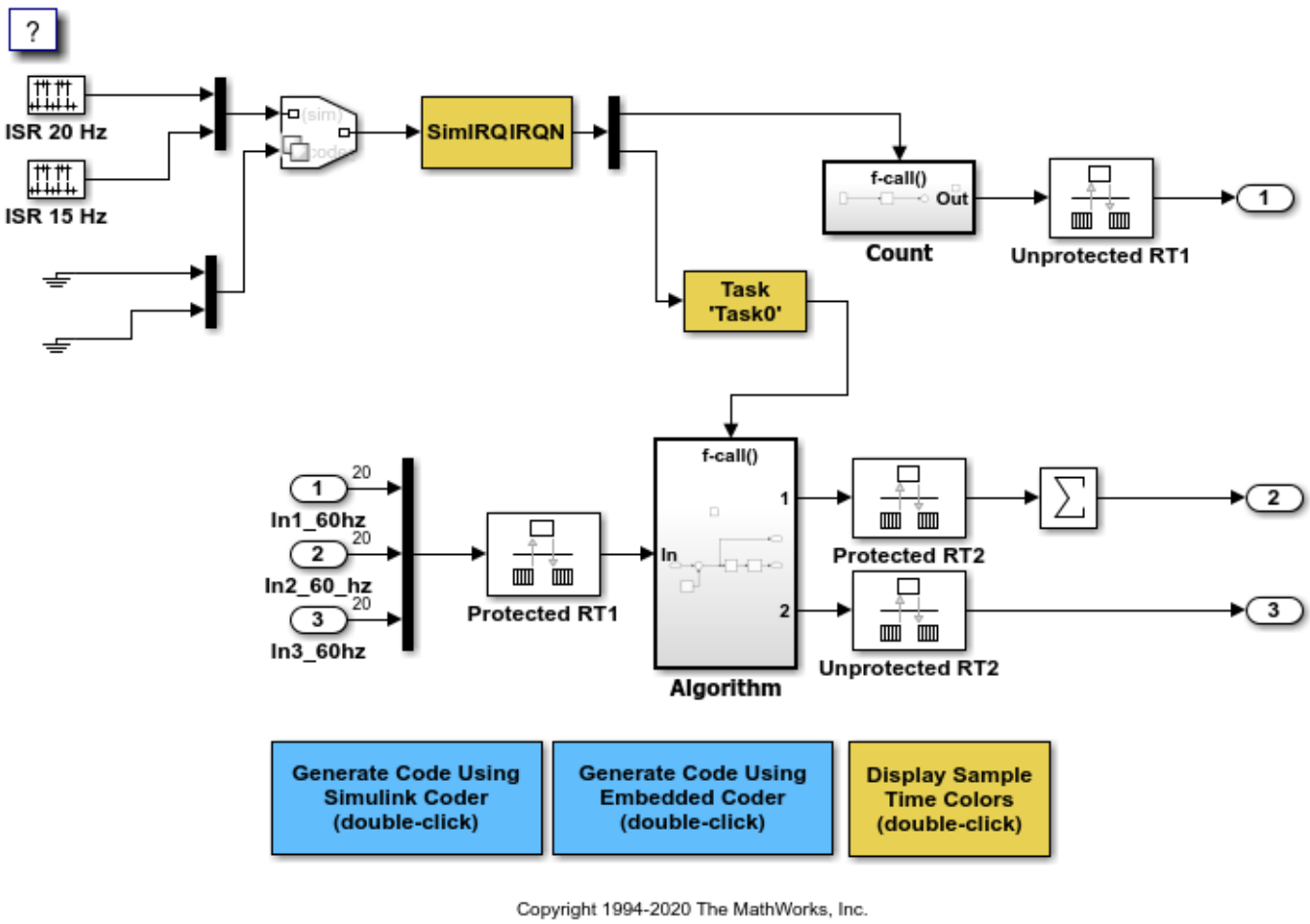
- “Spawn and Synchronize Execution of RTOS Task”
- “Pass Asynchronous Events in RTOS as Input to a Referenced Model”
- “Import Asynchronous Event Data for Simulation”
- “Rate Transitions and Asynchronous Blocks”

Spawn and Synchronize Execution of RTOS Task

This example shows how to simulate and generate code for asynchronous events on a real-time multitasking system. The model contains an Async Interrupt block, a Task Sync block, asynchronously executed subsystems `Count` and `Algorithm`, and Rate Transition blocks. The Async Interrupt block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to subsystem `Count` and the Task Sync block. `Count` executes at interrupt level. `Algorithm` executes as an asynchronous task. The code that the code generator produces for subsystems is tailored for the VxWorks® operating system.

About the Example Model

Open the example model `rtwdemo_async`.



You can place an Async Interrupt block between a simulated interrupt source and one of these model elements:

- Function call subsystem
- Task Sync block

- A Stateflow® chart configured for a function call input event
- A referenced model with an Inport block that connects to one of the preceding model elements

The Async Interrupt and Task Sync blocks enable the subsystems to execute asynchronously. Reconfigure the Async Interrupt and Task Sync blocks to generate code for an alternative run-time environment.

`Count` represents a simple interrupt service routine (ISR) that executes at interrupt level. It is best to keep ISRs as simple as possible. This subsystem includes only a Discrete-Time Integrator block.

`Algorithm` includes more substance. It includes multiple blocks and produces two output values. Execution of larger subsystems at interrupt level can significantly impact response time for interrupts of equal and lower priority in the system. A better solution for larger subsystems is to use the Task Sync block to represent the ISR for the function-call subsystem.

The Async Interrupt block generates calls to ISRs. Place the block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow chart configured for a function call input event

For each specified interrupt level, the block generates a Versa Module Eurocard (VME) ISR that executes the connected subsystem, Task Sync block, or chart.

In the example model, the Async Interrupt block is configured for VME interrupts 1 and 2, by using interrupt vector offsets 192 and 193. Interrupt 1 connects directly to subsystem `Count`. Interrupt 2 connects to a Task Sync block, which serves as the ISR for `Algorithm`. Place a Task Sync block in one of the following locations:

- Between an Async Interrupt block and a function-call subsystem or Stateflow chart.
- At the output port of a Stateflow chart that has an event, `Output to Simulink`, that you configure as a function call.

In the example model, the Task Sync block is between the Async Interrupt block and function-call subsystem `Algorithm`. The Task Sync block is configured with the task name `Task()`, a priority of 50, a stack size of 8192, and data transfers of the task synchronized with the caller task. The spawned task uses a semaphore to synchronize task execution. The Async Interrupt block triggers a release of the task semaphore.

Four Rate Transition blocks handle data transfers between ports that operate at different rates. In two instances, protected Rate Transition blocks protect data transfers (prevent them from being preempted and corrupted). In the other two instances, unprotected Rate Transition blocks introduce no special behavior. Their presence informs Simulink® of a rate transition.

The code generated for the Async Interrupt and Task Sync blocks is tailored for the example RTOS (VxWorks). However, you can modify the blocks to generate code specific to your run-time environment.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-size variable is atomic.

- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks with slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not stop and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are reddish-blue. Asynchronous interrupts and tasks are purple. The Rate Transition Blocks, which are a hybrid rate (their input and output sample times can differ), are yellow.

Generate Code and Report

Generate code and a code generation report for the model. Generated code for the Async Interrupt and Task Sync blocks is for the example RTOS (VxWorks). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.

2. Build the model.

```
### Starting build procedure for: rtwdemo_async
Warning: Simulink Coder: The tornado.tlc target will be removed in a future
release.
### Successful completion of code generation for: rtwdemo_async

Build Summary

Top model targets built:

Model          Action           Rebuild Reason
================================================================================
rtwdemo_async  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.539s
```

Review Initialization Code

Open the generated source file `rtwdemo_async.c`. The initialization code:

1. Creates and initializes the synchronization semaphore `Task0_semaphore`.

```
  *(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID = semBCreate(SEM_Q_PRIORITY,
    SEM_EMPTY);
  if (rtwdemo_async_DW.SFunction_PWORK.SemID == NULL) {
    printf("semBCreate call failed for block Task0.\n");
  }
```

2. Spawns task `task0` and assigns the task priority 50.

```
  rtwdemo_async_DW.SFunction_IWORK.TaskID = taskSpawn("Task0",
```

```
    50.0,
    VX_FP_TASK,
    8192.0,
    (FUNCPTR)Task0,
    0, 0, 0, 0, 0, 0, 0,0, 0, 0);
  if (rtwdemo_async_DW.SFunction_IWORK.TaskID == ERROR) {
    printf("taskSpawn call failed for block Task0.\n");
  }

  /* End of Start for S-Function (vxtask1): '<S4>/S-Function' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Connect and enable ISR function: isr_num1_vec192 */
  if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
    printf("intConnect failed for ISR 1.\n");
  }

  sysIntEnable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Connect and enable ISR function: isr_num2_vec193 */
  if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
    printf("intConnect failed for ISR 2.\n");
  }

  sysIntEnable(2);
```

3. Connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
  {
    int32_T i;

    /* InitializeConditions for RateTransition: '<Root>/Protected RT1' */
    for (i = 0; i < 60; i++) {
      rtwdemo_async_DW.ProtectedRT1_Buffer[i] = 0.0;
    }

    /* End of InitializeConditions for RateTransition: '<Root>/Protected RT1' */

    /* InitializeConditions for RateTransition: '<Root>/Protected RT2' */
    for (i = 0; i < 60; i++) {
      rtwdemo_async_DW.ProtectedRT2_Buffer[i] = 0.0;
    }

    /* End of InitializeConditions for RateTransition: '<Root>/Protected RT2' */

    /* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
     *  SubSystem: '<Root>/Count'
     */
    /* System initialize for function-call system: '<Root>/Count' */

    /* InitializeConditions for DiscreteIntegrator: '<S2>/Integrator' */
    rtwdemo_async_DW.Integrator_DSTATE_l = 0.0;
    rtwdemo_async_DW.Integrator_PREV_U_o = 0.0;

    /* SystemInitialize for Outport: '<Root>/Out1' incorporates:
```

```
   *  Outport: '<S2>/Out'
   */
  rtwdemo_async_Y.Out1 = 0.0;

  /* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
   *  SubSystem: '<S3>/Subsystem'
   */
  /* System initialize for function-call system: '<S3>/Subsystem' */

  /* SystemInitialize for S-Function (vxtask1): '<S4>/S-Function' incorporates:
   *  SubSystem: '<Root>/Algorithm'
   */

  /* System initialize for function-call system: '<Root>/Algorithm' */
  rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;

  /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
  rtwdemo_async_DW.Integrator_DSTATE = 0.0;

  /* InitializeConditions for Sum: '<S1>/Sum1' incorporates:
   *  DiscreteIntegrator: '<S1>/Integrator'
   */
  rtwdemo_async_DW.Integrator_PREV_U = 0.0;

  /* SystemInitialize for Sum: '<S1>/Sum' incorporates:
   *  Outport: '<S1>/Out1'
   */
  memset(&rtwdemo_async_B.Sum[0], 0, 60U * sizeof(real_T));

  /* SystemInitialize for Outport: '<Root>/Out3' incorporates:
   *  Outport: '<S1>/Out2'
   */
  rtwdemo_async_Y.Out3 = 0.0;

  /* End of SystemInitialize for S-Function (vxtask1): '<S4>/S-Function' */

  /* End of SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/* Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
 *  SubSystem: '<Root>/Count'
 */
/* Enable for function-call system: '<Root>/Count' */
rtwdemo_async_DW.Count_RESET_ELAPS_T = true;

/* Enable for DiscreteIntegrator: '<S2>/Integrator' */
rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h = 1U;

/* Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
 *  SubSystem: '<S3>/Subsystem'
 */

/* Enable for function-call system: '<S3>/Subsystem' */

/* Enable for S-Function (vxtask1): '<S4>/S-Function' incorporates:
 *  SubSystem: '<Root>/Algorithm'
 */
```

```
  /* Enable for function-call system: '<Root>/Algorithm' */
  rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;
  rtwdemo_async_DW.Algorithm_RESET_ELAPS_T = true;

  /* Enable for DiscreteIntegrator: '<S1>/Integrator' */
  rtwdemo_async_DW.Integrator_SYSTEM_ENABLE = 1U;

  /* End of Enable for S-Function (vxtask1): '<S4>/S-Function' */

  /* End of Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/* Model terminate function */
static void rtwdemo_async_terminate(void)
{
  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
   *  SubSystem: '<S3>/Subsystem'
   */

  /* Termination for function-call system: '<S3>/Subsystem' */

  /* Terminate for S-Function (vxtask1): '<S4>/S-Function' */

  /* VxWorks Task Block: '<S4>/S-Function' (vxtask1) */
  /* Destroy task: Task0 */
  taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

  /* End of Terminate for S-Function (vxtask1): '<S4>/S-Function' */

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/*=========================================================================*
 * Start of Classic call interface                                         *
 *=========================================================================*/
void MdlOutputs(int_T tid)
{
  rtwdemo_async_output(tid);
}

void MdlUpdate(int_T tid)
{
  rtwdemo_async_update(tid);
}

void MdlInitializeSizes(void)
```

```
{
}

void MdlInitializeSampleTimes(void)
{
}

void MdlInitialize(void)
{
}

void MdlStart(void)
{
  rtwdemo_async_initialize();
}

void MdlTerminate(void)
{
  rtwdemo_async_terminate();
}

/* Registration function */
RT_MODEL_rtwdemo_async_T *rtwdemo_async(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)rtwdemo_async_M, 0,
                sizeof(RT_MODEL_rtwdemo_async_T));

  /* Initialize timing info */
  {
    int_T *mdlTsMap = rtwdemo_async_M->Timing.sampleTimeTaskIDArray;
    mdlTsMap[0] = 0;
    mdlTsMap[1] = 1;

    /* polyspace +2 MISRA2012:D4.1 [Justified:Low] "rtwdemo_async_M points to
       static memory which is guaranteed to be non-NULL" */
    rtwdemo_async_M->Timing.sampleTimeTaskIDPtr = (&mdlTsMap[0]);
    rtwdemo_async_M->Timing.sampleTimes =
      (&rtwdemo_async_M->Timing.sampleTimesArray[0]);
    rtwdemo_async_M->Timing.offsetTimes =
      (&rtwdemo_async_M->Timing.offsetTimesArray[0]);

    /* task periods */
    rtwdemo_async_M->Timing.sampleTimes[0] = (0.016666666666666666);
    rtwdemo_async_M->Timing.sampleTimes[1] = (0.05);

    /* task offsets */
    rtwdemo_async_M->Timing.offsetTimes[0] = (0.0);
    rtwdemo_async_M->Timing.offsetTimes[1] = (0.0);
  }

  rtmSetTPtr(rtwdemo_async_M, &rtwdemo_async_M->Timing.tArray[0]);
```

```
  {
    int_T *mdlSampleHits = rtwdemo_async_M->Timing.sampleHitArray;
    int_T *mdlPerTaskSampleHits = rtwdemo_async_M->Timing.perTaskSampleHitsArray;
    rtwdemo_async_M->Timing.perTaskSampleHits = (&mdlPerTaskSampleHits[0]);
    mdlSampleHits[0] = 1;
    rtwdemo_async_M->Timing.sampleHits = (&mdlSampleHits[0]);
  }

  rtmSetTFinal(rtwdemo_async_M, 0.5);
  rtwdemo_async_M->Timing.stepSize0 = 0.016666666666666666;
  rtwdemo_async_M->Timing.stepSize1 = 0.05;
  rtwdemo_async_M->solverInfoPtr = (&rtwdemo_async_M->solverInfo);
  rtwdemo_async_M->Timing.stepSize = (0.016666666666666666);
  rtsiSetFixedStepSize(&rtwdemo_async_M->solverInfo, 0.016666666666666666);
  rtsiSetSolverMode(&rtwdemo_async_M->solverInfo, SOLVER_MODE_MULTITASKING);

  /* block I/O */
  rtwdemo_async_M->blockIO = ((void *) &rtwdemo_async_B);
  (void) memset(((void *) &rtwdemo_async_B), 0,
               sizeof(B_rtwdemo_async_T));

  /* states (dwork) */
  rtwdemo_async_M->dwork = ((void *) &rtwdemo_async_DW);
  (void) memset((void *)&rtwdemo_async_DW, 0,
               sizeof(DW_rtwdemo_async_T));

  /* external inputs */
  rtwdemo_async_M->inputs = (((void*)&rtwdemo_async_U));
  (void)memset(&rtwdemo_async_U, 0, sizeof(ExtU_rtwdemo_async_T));

  /* external outputs */
  rtwdemo_async_M->outputs = (&rtwdemo_async_Y);
  (void)memset(&rtwdemo_async_Y, 0, sizeof(ExtY_rtwdemo_async_T));

  /* Initialize Sizes */
  rtwdemo_async_M->Sizes.numContStates = (0);/* Number of continuous states */
  rtwdemo_async_M->Sizes.numY = (3);   /* Number of model outputs */
  rtwdemo_async_M->Sizes.numU = (60);  /* Number of model inputs */
  rtwdemo_async_M->Sizes.sysDirFeedThru = (1);/* The model is direct feedthrough */
  rtwdemo_async_M->Sizes.numSampTimes = (2);/* Number of sample times */
  rtwdemo_async_M->Sizes.numBlocks = (19);/* Number of blocks */
  rtwdemo_async_M->Sizes.numBlockIO = (6);/* Number of block outputs */
  return rtwdemo_async_M;
}

/*=========================================================================*
 * End of Classic call interface                                           *
 *=========================================================================*/
```

The order of these operations is important. Before the code generator enables the interrupt that activates the task, it must spawn the task.

Review Task and Task Synchronization Code

In the generated source file `rtwdemo_async.c`, review the task and task synchronization code.

The code generator produces the code for function `Task0` from the Task Sync block. That function includes a small amount of interrupt-level code and runs as an RTOS task.

The task waits in an infinite `for` loop until the system releases a synchronization semaphore. If the system releases the semaphore, the function updates its task timer and calls the code generated for the `Algorithm` subsystem.

In the example model, the **Synchronize the data transfer of this task with the caller task** parameter for the Task Sync block is set. This parameter setting updates the timer associated with the Task Sync block (`rtM->Timing.clockTick2`) with the value of the timer that the Async Interrupt block (`rtM->Timing.clockTick3`) maintains. As a result, code for blocks within the `Algorithm` subsystem use timer values that are based on the time of the most recent interrupt, rather than the most recent activation of `Task0`.

```
{
  /* Wait for semaphore to be released by system: rtwdemo_async/Task Sync */
  for (;;) {
    if (semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID,NO_WAIT) !=
        ERROR) {
      logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);

#if STOPONOVERRUN

      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);

#endif

    } else {
      semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID, WAIT_FOREVER);
    }

    /* Use the upstream clock tick counter for this Task. */
    rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;

    /* Call the system: '<Root>/Algorithm' */
    {
      int32_T i;
      int32_T i_0;

      /* RateTransition: '<Root>/Protected RT1' */
      i = rtwdemo_async_DW.ProtectedRT1_ActiveBufIdx * 60;
      for (i_0 = 0; i_0 < 60; i_0++) {
        rtwdemo_async_B.ProtectedRT1[i_0] =
          rtwdemo_async_DW.ProtectedRT1_Buffer[i_0 + i];
      }

      /* End of RateTransition: '<Root>/Protected RT1' */

      /* S-Function (vxtask1): '<S4>/S-Function' */

      /* Output and update for function-call system: '<Root>/Algorithm' */
      {
        real_T Sum;
        int32_T i;
        rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;
        if (rtwdemo_async_DW.Algorithm_RESET_ELAPS_T) {
          rtwdemo_async_DW.Algorithm_ELAPS_T = 0U;
```

```
        } else {
          rtwdemo_async_DW.Algorithm_ELAPS_T =
            rtwdemo_async_M->Timing.clockTick2 -
            rtwdemo_async_DW.Algorithm_PREV_T;
        }

        rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick2;
        rtwdemo_async_DW.Algorithm_RESET_ELAPS_T = false;

        /* DiscreteIntegrator: '<S1>/Integrator' */
        if (rtwdemo_async_DW.Integrator_SYSTEM_ENABLE == 0) {
          /* DiscreteIntegrator: '<S1>/Integrator' */
          rtwdemo_async_DW.Integrator_DSTATE += 0.016666666666666666 * (real_T)
            rtwdemo_async_DW.Algorithm_ELAPS_T
            * rtwdemo_async_DW.Integrator_PREV_U;
        }

        /* End of DiscreteIntegrator: '<S1>/Integrator' */

        /* Outport: '<Root>/Out3' incorporates:
         *  SignalConversion generated from: '<S1>/Out2'
         */
        rtwdemo_async_Y.Out3 = rtwdemo_async_DW.Integrator_DSTATE;

        /* Sum: '<S1>/Sum1' */
        rtwdemo_async_DW.Integrator_PREV_U = -0.0;
        for (i = 0; i < 60; i++) {
          /* Sum: '<S1>/Sum' incorporates:
           *  Constant: '<S1>/Offset'
           */
          Sum = rtwdemo_async_B.ProtectedRT1[i] + 1.25;
          rtwdemo_async_B.Sum[i] = Sum;

          /* Sum: '<S1>/Sum1' */
          rtwdemo_async_DW.Integrator_PREV_U += Sum;
        }

        /* Update for DiscreteIntegrator: '<S1>/Integrator' */
        rtwdemo_async_DW.Integrator_SYSTEM_ENABLE = 0U;
      }

      /* End of Outputs for S-Function (vxtask1): '<S4>/S-Function' */

      /* RateTransition: '<Root>/Protected RT2' */
      for (i = 0; i < 60; i++) {
        rtwdemo_async_DW.ProtectedRT2_Buffer[i +
          (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0) * 60] =
          rtwdemo_async_B.Sum[i];
      }

      rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx = (int8_T)
        (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0);

      /* End of RateTransition: '<Root>/Protected RT2' */
    }
  }
}
```

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick3 = tickGet();

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
  fppSave(&context);

  /* Call the system: '<Root>/Count' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<Root>/Count' */
    if (rtwdemo_async_DW.Count_RESET_ELAPS_T) {
      rtwdemo_async_DW.Count_ELAPS_T = 0U;
    } else {
      rtwdemo_async_DW.Count_ELAPS_T = rtwdemo_async_M->Timing.clockTick3 -
        rtwdemo_async_DW.Count_PREV_T;
    }

    rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick3;
    rtwdemo_async_DW.Count_RESET_ELAPS_T = false;

    /* DiscreteIntegrator: '<S2>/Integrator' */
    if (rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h == 0) {
      /* DiscreteIntegrator: '<S2>/Integrator' */
      rtwdemo_async_DW.Integrator_DSTATE_l += 0.016666666666666666 * (real_T)
        rtwdemo_async_DW.Count_ELAPS_T * rtwdemo_async_DW.Integrator_PREV_U_o;
    }

    /* End of DiscreteIntegrator: '<S2>/Integrator' */

    /* Outport: '<Root>/Out1' incorporates:
     *  SignalConversion generated from: '<S2>/Out'
     */
    rtwdemo_async_Y.Out1 = rtwdemo_async_DW.Integrator_DSTATE_l;

    /* Update for DiscreteIntegrator: '<S2>/Integrator' incorporates:
     *  Constant: '<S2>/Constant'
     */
    rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h = 0U;
    rtwdemo_async_DW.Integrator_PREV_U_o = 1.0;

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);
```

```
  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{
  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick4 = tickGet();

  /* Call the system: '<S3>/Subsystem' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<S3>/Subsystem' */

    /* S-Function (vxtask1): '<S4>/S-Function' */

    /* VxWorks Task Block: '<S4>/S-Function' (vxtask1) */
    /* Release semaphore for system task: Task0 */
    semGive(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID);

    /* End of Outputs for S-Function (vxtask1): '<S4>/S-Function' */

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }
}
```

The code generator produces code for ISRs `isr_num1_vec192` and `isr_num2_vec293`. ISR `isr_num2_vec192`:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem that connects to the referenced model Inport block, which receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick3 = tickGet();

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
```

```
  fppSave(&context);

  /* Call the system: '<Root>/Count' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<Root>/Count' */
    if (rtwdemo_async_DW.Count_RESET_ELAPS_T) {
      rtwdemo_async_DW.Count_ELAPS_T = 0U;
    } else {
      rtwdemo_async_DW.Count_ELAPS_T = rtwdemo_async_M->Timing.clockTick3 -
        rtwdemo_async_DW.Count_PREV_T;
    }

    rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick3;
    rtwdemo_async_DW.Count_RESET_ELAPS_T = false;

    /* DiscreteIntegrator: '<S2>/Integrator' */
    if (rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h == 0) {
      /* DiscreteIntegrator: '<S2>/Integrator' */
      rtwdemo_async_DW.Integrator_DSTATE_l += 0.016666666666666666 * (real_T)
        rtwdemo_async_DW.Count_ELAPS_T * rtwdemo_async_DW.Integrator_PREV_U_o;
    }

    /* End of DiscreteIntegrator: '<S2>/Integrator' */

    /* Outport: '<Root>/Out1' incorporates:
     *  SignalConversion generated from: '<S2>/Out'
     */
    rtwdemo_async_Y.Out1 = rtwdemo_async_DW.Integrator_DSTATE_l;

    /* Update for DiscreteIntegrator: '<S2>/Integrator' incorporates:
     *  Constant: '<S2>/Constant'
     */
    rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h = 0U;
    rtwdemo_async_DW.Integrator_PREV_U_o = 1.0;

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);

  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

ISR `isr_num2_vec293` maintains a timer that stores the tick count at the time that the interrupt occurs. After updating the timer, the ISR releases the semaphore that activates `Task0`.

```
/* Spawned with priority: 50 */
void Task0(void)
{
  /* Wait for semaphore to be released by system: rtwdemo_async/Task Sync */
  for (;;) {
```

```
    if (semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID,NO_WAIT) !=
        ERROR) {
      logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);

#if STOPONOVERRUN

      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);

#endif

    } else {
      semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID, WAIT_FOREVER);
    }

    /* Use the upstream clock tick counter for this Task. */
    rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;

    /* Call the system: '<Root>/Algorithm' */
    {
      int32_T i;
      int32_T i_0;

      /* RateTransition: '<Root>/Protected RT1' */
      i = rtwdemo_async_DW.ProtectedRT1_ActiveBufIdx * 60;
      for (i_0 = 0; i_0 < 60; i_0++) {
        rtwdemo_async_B.ProtectedRT1[i_0] =
          rtwdemo_async_DW.ProtectedRT1_Buffer[i_0 + i];
      }

      /* End of RateTransition: '<Root>/Protected RT1' */

      /* S-Function (vxtask1): '<S4>/S-Function' */

      /* Output and update for function-call system: '<Root>/Algorithm' */
      {
        real_T Sum;
        int32_T i;
        rtwdemo_async_M->Timing.clockTick2 = rtwdemo_async_M->Timing.clockTick4;
        if (rtwdemo_async_DW.Algorithm_RESET_ELAPS_T) {
          rtwdemo_async_DW.Algorithm_ELAPS_T = 0U;
        } else {
          rtwdemo_async_DW.Algorithm_ELAPS_T =
            rtwdemo_async_M->Timing.clockTick2 -
            rtwdemo_async_DW.Algorithm_PREV_T;
        }

        rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick2;
        rtwdemo_async_DW.Algorithm_RESET_ELAPS_T = false;

        /* DiscreteIntegrator: '<S1>/Integrator' */
        if (rtwdemo_async_DW.Integrator_SYSTEM_ENABLE == 0) {
          /* DiscreteIntegrator: '<S1>/Integrator' */
          rtwdemo_async_DW.Integrator_DSTATE += 0.016666666666666666 * (real_T)
            rtwdemo_async_DW.Algorithm_ELAPS_T
            * rtwdemo_async_DW.Integrator_PREV_U;
        }
```

```
        /* End of DiscreteIntegrator: '<S1>/Integrator' */

        /* Outport: '<Root>/Out3' incorporates:
         *  SignalConversion generated from: '<S1>/Out2'
         */
        rtwdemo_async_Y.Out3 = rtwdemo_async_DW.Integrator_DSTATE;

        /* Sum: '<S1>/Sum1' */
        rtwdemo_async_DW.Integrator_PREV_U = -0.0;
        for (i = 0; i < 60; i++) {
          /* Sum: '<S1>/Sum' incorporates:
           *  Constant: '<S1>/Offset'
           */
          Sum = rtwdemo_async_B.ProtectedRT1[i] + 1.25;
          rtwdemo_async_B.Sum[i] = Sum;

          /* Sum: '<S1>/Sum1' */
          rtwdemo_async_DW.Integrator_PREV_U += Sum;
        }

        /* Update for DiscreteIntegrator: '<S1>/Integrator' */
        rtwdemo_async_DW.Integrator_SYSTEM_ENABLE = 0U;
      }

      /* End of Outputs for S-Function (vxtask1): '<S4>/S-Function' */

      /* RateTransition: '<Root>/Protected RT2' */
      for (i = 0; i < 60; i++) {
        rtwdemo_async_DW.ProtectedRT2_Buffer[i +
          (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0) * 60] =
          rtwdemo_async_B.Sum[i];
      }

      rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx = (int8_T)
        (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0);

      /* End of RateTransition: '<Root>/Protected RT2' */
    }
  }
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick3 = tickGet();

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
  fppSave(&context);
```

```
  /* Call the system: '<Root>/Count' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<Root>/Count' */
    if (rtwdemo_async_DW.Count_RESET_ELAPS_T) {
      rtwdemo_async_DW.Count_ELAPS_T = 0U;
    } else {
      rtwdemo_async_DW.Count_ELAPS_T = rtwdemo_async_M->Timing.clockTick3 -
        rtwdemo_async_DW.Count_PREV_T;
    }

    rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick3;
    rtwdemo_async_DW.Count_RESET_ELAPS_T = false;

    /* DiscreteIntegrator: '<S2>/Integrator' */
    if (rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h == 0) {
      /* DiscreteIntegrator: '<S2>/Integrator' */
      rtwdemo_async_DW.Integrator_DSTATE_l += 0.016666666666666666 * (real_T)
        rtwdemo_async_DW.Count_ELAPS_T * rtwdemo_async_DW.Integrator_PREV_U_o;
    }

    /* End of DiscreteIntegrator: '<S2>/Integrator' */

    /* Outport: '<Root>/Out1' incorporates:
     *  SignalConversion generated from: '<S2>/Out'
     */
    rtwdemo_async_Y.Out1 = rtwdemo_async_DW.Integrator_DSTATE_l;

    /* Update for DiscreteIntegrator: '<S2>/Integrator' incorporates:
     *  Constant: '<S2>/Constant'
     */
    rtwdemo_async_DW.Integrator_SYSTEM_ENABLE_h = 0U;
    rtwdemo_async_DW.Integrator_PREV_U_o = 1.0;

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);

  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

Review Task Termination Code

The Task Sync block generates the following termination code.

```
static void rtwdemo_async_terminate(void)
{
  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

```
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
   *  SubSystem: '<S3>/Subsystem'
   */

  /* Termination for function-call system: '<S3>/Subsystem' */

  /* Terminate for S-Function (vxtask1): '<S4>/S-Function' */

  /* VxWorks Task Block: '<S4>/S-Function' (vxtask1) */
  /* Destroy task: Task0 */
  taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

  /* End of Terminate for S-Function (vxtask1): '<S4>/S-Function' */

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Related Information

- Async Interrupt
- Task Sync
- “Generate Interrupt Service Routines”
- “Timers in Asynchronous Tasks”
- “Create a Customized Asynchronous Library”
- “Import Asynchronous Event Data for Simulation”
- “Load Data to Root-Level Input Ports”
- “Asynchronous Events”
- “Rate Transitions and Asynchronous Blocks”
- “Asynchronous Support Limitations”

See Also

More About

- “Generate Interrupt Service Routines”
- “Pass Asynchronous Events in RTOS as Input to a Referenced Model”
- “Timers in Asynchronous Tasks”
- “Import Asynchronous Event Data for Simulation”
- “Rate Transitions and Asynchronous Blocks”

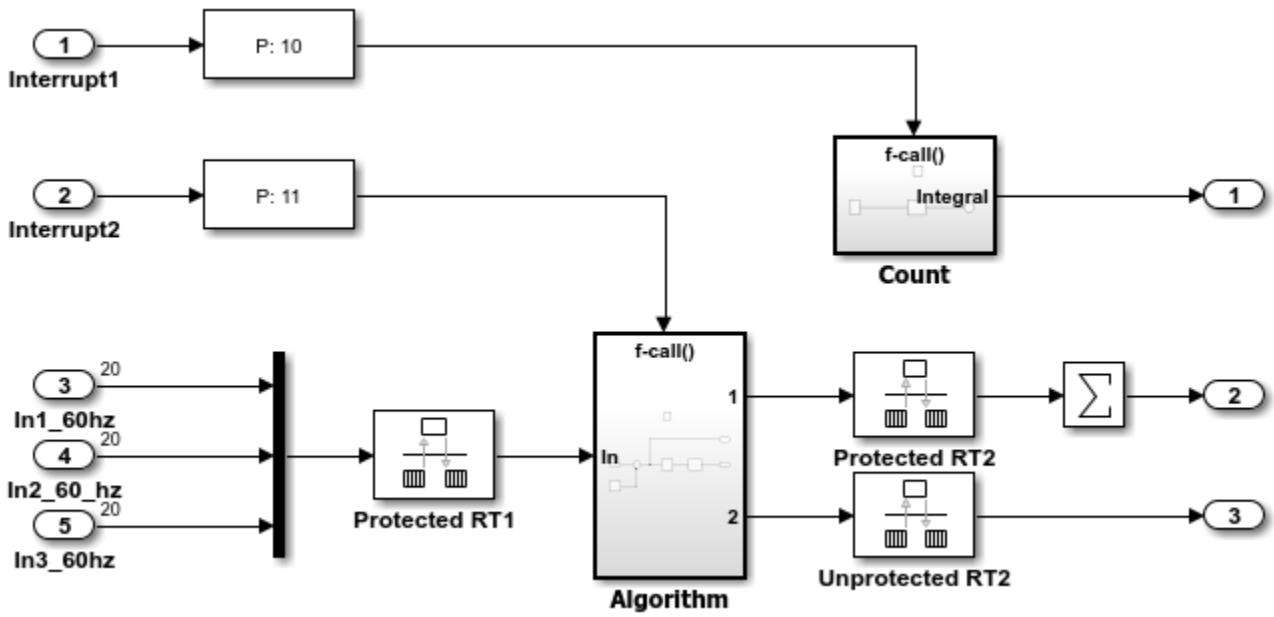
Pass Asynchronous Events in RTOS as Input to a Referenced Model

This example shows how to simulate and generate code for asynchronous events on a real-time multitasking system.

About the Example Model

Open the example model `rtwdemo_async_mdlreftop`.

```
open_system('rtwdemo_async_mdlreftop');
```



The model simulates an interrupt source and includes an `Async Interrupt` block and referenced model. The `Async Interrupt` block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to Inport blocks 1 and 2 of the referenced model.

You can place an `Async Interrupt` block between a simulated interrupt source and one of these model elements:

- Function call subsystem
- `Task Sync` block
- A Stateflow® chart configured for a function call input event
- A referenced model with an `Inport` block that connects to one of the preceding model elements

In this example model, the `Async Interrupt` block passes asynchronous events (function-call trigger signals), `Interrupt1` and `Interrupt2`, to the referenced model through `Inport` blocks 1 and 2.

The code generator produces code that is tailored for the VxWorks operating system. Reconfigure the `Async Interrupt` block to generate code for an alternative application run-time environment.

Open the referenced model. The referenced model includes the two `Inport` blocks that receive the interrupts, each connected to an `Asynchronous Task Specification` block, function-call subsystems Count and Algorithm, and `Rate Transition` blocks. The `Asynchronous Task Specification` block, in combination with a root-level `Inport` block, allows a reference model to receive asynchronous function-call input. To use the block:

1. Connect the `Asynchronous Task Specification` block to the output port of the root-level `Inport` block that outputs a function-call trigger.
2. Select the **Output function call** parameter of the `Inport` block to specify that it accepts function-call signals.
3. On the parameters dialog box for the `Asynchronous Task Specification` block, set the task priority for the asynchronous task associated with an `Inport` block. Specify an integer or []. If you specify an integer, it must match the priority of the interrupt initiated by the `Async Interrupt` block in the parent model. If you specify [], the priorities do not have to match.

The `Asynchronous Task Specification` block for the higher priority interrupt, `interrupt1`, connect to function-call subsystem `Count`. `Count` represents a simple interrupt service routine (ISR). The second `Asynchronous Task Specification` block connect to the subsystem `Algorithm`, which includes more substance. It includes multiple blocks and produces two output values. Both subsystems execute at interrupt level.

For each interrupt level specified for the `Async Interrupt` block in the parent model, the block generates a VME ISR that executes the connected subsystem, `Task Sync` block, or chart.

In the example top model, the `Async Interrupt` block is configured for VME interrupts 1 and 2, using interrupt vector offsets 192 and 193. Interrupt 1 is wired to trigger subsystem `Count`. Interrupt 2 is wired to trigger subsystem `Algorithm`.

The `Rate Transition` blocks handle data transfers between ports that operate at different rates. In two instances, the blocks protect data transfers (prevent them from being preempted and corrupted). In the other instance, no special behavior occurs.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-sized variable is atomic.
- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks with slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are magenta. Asynchronous interrupts are purple. The `Rate Transition` blocks, which are hybrid (input and output sample times can differ), appear yellow.

Generate Code and Report

Generate code and a code generation report for the model. `Async Interrupt` block and `Task Sync` block generated code is for the example RTOS (VxWorks). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

2. Build the model.

```
slbuild('rtwdemo_async_mdlreftop');
```

```
### Starting serial model reference code generation build.

Warning: Simulink Coder: The tornado.tlc target will be removed in a future release.

### Successfully updated the model reference code generation target for: rtwdemo_async_mdlrefbot
### Starting build procedure for: rtwdemo_async_mdlreftop

Warning: Simulink Coder: The tornado.tlc target will be removed in a future release.

### Successful completion of code generation for: rtwdemo_async_mdlreftop

Build Summary

Code generation targets built:

Model                     Action           Rebuild Reason
========================================================================================
rtwdemo_async_mdlrefbot  Code generated.  rtwdemo_async_mdlrefbot.c does not exist.

Top model targets built:

Model                     Action           Rebuild Reason
==========================================================================================
rtwdemo_async_mdlreftop  Code generated.  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 1m 10.11s
```

Review Initialization Code

Open the generated source file `rtwdemo_async_mdlreftop.c`. The initialization code connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
cfile = fullfile(cgDir, 'rtwdemo_async_mdlreftop_tornado_rtw', 'rtwdemo_async_mdlreftop.c');
rtwdemodbtype(cfile, ...
    'static void rtwdemo_async_mdlreftop_initialize(void)', ...
    '/* Model terminate function */', ...
    1, 0);
```

```
static void rtwdemo_async_mdlreftop_initialize(void)
{
  /* Start for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

```
  /* Connect and enable ISR function: isr_num1_vec192 */
  if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
    printf("intConnect failed for ISR 1.\n");
  }

  sysIntEnable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Connect and enable ISR function: isr_num2_vec193 */
  if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
    printf("intConnect failed for ISR 2.\n");
  }

  sysIntEnable(2);

  /* End of Start for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* SystemInitialize for ModelReference: '<Root>/Model' incorporates:
   *  Inport: '<Root>/In1_60hz'
   *  Inport: '<Root>/In2_60_hz'
   *  Inport: '<Root>/In3_60hz'
   *  Outport: '<Root>/Out1'
   *  Outport: '<Root>/Out2'
   *  Outport: '<Root>/Out3'
   */
  rtwdemo_async_mdlrefbot_Init(&rtwdemo_async_mdlreftop_Y.Out1);
  rtwdemo_async_mdlrefbot_Enable();
}
```

Review ISR Code

In the generated source file `rtwdemo_async_mdlreftop.c`, review the code for ISRs `isr_num1_vec192` and `isr_num2_vec293`. Each ISR:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem connected to the referenced model `Inport` block that receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```
cfile = fullfile(cgDir, 'rtwdemo_async_mdlreftop_tornado_rtw', 'rtwdemo_async_mdlreftop.c');
rtwdemodbtype(cfile, ...
    'void isr_num1_vec192(void)', ...
    'time_T rt_SimUpdateDiscreteEvents', ...
    1, 0);
```

```
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
```

```
  fppSave(&context);

  /* Call the system: '<Root>/Model' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* ModelReference: '<Root>/Model' incorporates:
     *  Inport: '<Root>/In1_60hz'
     *  Inport: '<Root>/In2_60_hz'
     *  Inport: '<Root>/In3_60hz'
     *  Outport: '<Root>/Out1'
     *  Outport: '<Root>/Out2'
     *  Outport: '<Root>/Out3'
     */
    rtwdemo_async_mdlrefbot_Interrupt1(&rtwdemo_async_mdlreftop_Y.Out1);

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);

  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{
  FP_CONTEXT context;

  /* save floating point context */
  fppSave(&context);

  /* Call the system: '<Root>/Model' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* ModelReference: '<Root>/Model' incorporates:
     *  Inport: '<Root>/In1_60hz'
     *  Inport: '<Root>/In2_60_hz'
     *  Inport: '<Root>/In3_60hz'
     *  Outport: '<Root>/Out1'
     *  Outport: '<Root>/Out2'
     *  Outport: '<Root>/Out3'
     */
    rtwdemo_async_mdlrefbot_Interrupt2();

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);
}
```

Review Task Termination Code

The `Task Sync` block generates the following termination code.

```
cfile = fullfile(cgDir, 'rtwdemo_async_mdlreftop_tornado_rtw', 'rtwdemo_async_mdlreftop.c');
rtwdemodbtype(cfile, ...
    'static void rtwdemo_async_mdlreftop_terminate(void)', ...
    '/*=========================================================================*', ...
    1, 0);

static void rtwdemo_async_mdlreftop_terminate(void)
{
  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Related Information

- Async Interrupt
- Asynchronous Task Specification
- “Generate Interrupt Service Routines”
- “Timers in Asynchronous Tasks”
- “Create a Customized Asynchronous Library”
- “Import Asynchronous Event Data for Simulation”
- “Load Data to Root-Level Input Ports”
- “Asynchronous Events”
- “Rate Transitions and Asynchronous Blocks”
- “Asynchronous Support Limitations”

```
bdclose('rtwdemo_async_mdlreftop');
rtwdemoclean;
cd(currentDir)
```

See Also

More About

- “Generate Interrupt Service Routines”
- “Spawn and Synchronize Execution of RTOS Task”
- “Timers in Asynchronous Tasks”
- “Import Asynchronous Event Data for Simulation”
- “Rate Transitions and Asynchronous Blocks”

Rate Transitions and Asynchronous Blocks

Because an asynchronous function call subsystem can preempt or be preempted by other model code, an inconsistency arises when more than one signal element is connected to an asynchronous block. The issue is that signals passed to and from the function call subsystem can be in the process of being written to or read from when the preemption occurs. Thus, some old and some new data is used. This situation can also occur with scalar signals in some cases. For example, if a signal is a double (8 bytes), the read or write operation might require two machine instructions.

Note The operating system integration techniques that are demonstrated in this section use blocks that are in the Interrupt Templates block library. The blocks in that library provide starting point examples to help you develop custom blocks for a target environment.

About Rate Transitions and Asynchronous Blocks

The Simulink Rate Transition block is designed to deal with preemption problems that occur in data transfer between blocks running at different rates. These issues are discussed in “Time-Based Scheduling and Code Generation”.

You can handle rate transition issues automatically by selecting model configuratoin parameter **Automatically handle rate transition for data transfer**. This saves you from having to manually insert Rate Transition blocks to avoid invalid rate transitions, including invalid *asynchronous-to-periodic* and *asynchronous-to-asynchronous* rate transitions, in multirate models. For asynchronous tasks, the Simulink engine configures inserted blocks for data integrity but not determinism during data transfers.

For asynchronous rate transitions, the Rate Transition block provides data integrity, but cannot provide determinism. Therefore, when you insert Rate Transition blocks explicitly, you must clear the block parameter **Ensure deterministic data transfer (maximum delay)**.

When you insert a Rate Transition block between two blocks to maintain data integrity and priorities are assigned to the tasks associated with the blocks, the code generator assumes that the higher priority task can preempt the lower priority task and the lower priority task cannot preempt the higher priority task. If the priority associated with task for either block is not assigned or the priorities of the tasks for both blocks are the same, the code generator assumes that either task can preempt the other task.

Priorities of periodic tasks are assigned by the Simulink engine, in accordance with the parameters specified in the **Solver selection** section of the **Solver** pane of the Configuration Parameters dialog box. When the **Periodic sample time constraint** parameter is set to `Unconstrained`, the model base rate priority is set to `40`. Priorities for subrates then increment or decrement by `1` from the base rate priority, depending on the setting of parameter **Higher priority value indicates higher task priority**.

You can assign priorities manually by using parameter **Periodic sample time properties**. The Simulink engine does not assign a priority to asynchronous blocks. For example, the priority of a function call subsystem that connects back to an Async Interrupt block is assigned by the Async Interrupt block.

Block parameter **Simulink task priority** of the Async Interrupt block specifies a priority level (required) for every interrupt number entered for parameter **VME interrupt number(s)**. The priority array sets the priorities of the subsystems connected to each interrupt.

For the Task Sync block, if the example RTOS (VxWorks) is the target, clear block parameter **Higher priority value indicates higher task priority**. Parameter **Simulink task priority** specifies the block priority relative to connected blocks (in addition to assigning an RTOS priority to the generated task code).

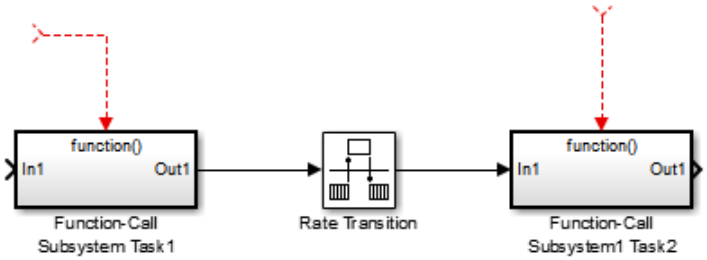
The `vxlib1` library provides two types of rate transition blocks as a convenience. These are simply preconfigured instances of the Rate Transition block:

- Protected Rate Transition block: Rate Transition block that is configured with block parameter **Ensure data integrity during data transfers** selected and **Ensure deterministic data transfer** cleared.
- Unprotected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** cleared.

Handle Rate Transitions for Asynchronous Tasks

For rate transitions that involve asynchronous tasks, you can maintain data integrity. However, you cannot achieve determinism. You have the option of using the Rate Transition block or target-specific rate transition blocks.

Consider the following model, which includes a Rate Transition block.



You can use the Rate Transition block in either of the following modes:

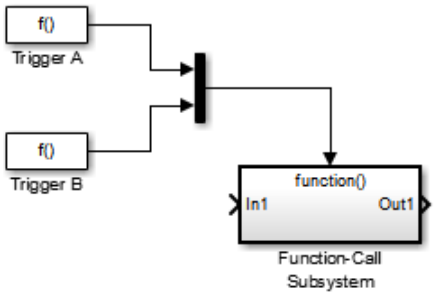
- Maintain data integrity, no determinism
- Unprotected

Alternatively, you can use target-specific rate transition blocks. The following blocks are available for the example RTOS (VxWorks):

- Protected Rate Transition block (reader)
- Protected Rate Transition block (writer)
- Unprotected Rate Transition block

Handle Multiple Asynchronous Interrupts

Consider the following model, in which two functions trigger the same subsystem.



The two tasks must have equal priorities. When priorities are the same, the outcome depends on whether they are firing periodically or asynchronously, and also on a diagnostic setting. The following table and notes describe these outcomes:

Supported Sample Time and Priority for Function Call Subsystem with Multiple Triggers

	Async Priority = 1	Async Priority = 2	Async Priority Unspecified	Periodic Priority = 1	Periodic Priority = 2
Async Priority = 1	Supported (1)				
Async Priority = 2		Supported (1)			
Async Priority Unspecified			Supported (2)		
Periodic Priority = 1				Supported	
Periodic Priority = 2					Supported

1. Control these outcomes by using model configuratoin parameter **Tasks with equal priority** ; set this diagnostic to `none` if tasks of equal priority cannot preempt each other in the target system.
2. For this case, the following warning message is issued unconditionally:

   ```
   The function call subsystem <name> has multiple asynchronous
   triggers that do not specify priority. Data integrity will
   not be maintained if these triggers can preempt one another.
   ```

Empty cells in the above table represent multiple triggers with differing priorities, which are unsupported.

The code generator provides absolute time management for a function call subsystem connected to multiple interrupts in the case where timer settings for `TriggerA` and `TriggerB` (time source, resolution) are the same.

Assume that all of the following conditions are true for the model shown above:

- A function call subsystem is triggered by two asynchronous triggers (TriggerA and TriggerB) having identical priority settings.
- Each trigger sets the source of time and timer attributes by calling the functions `ssSetTimeSource` and `ssSetAsyncTimerAttributes`.

- The triggered subsystem contains a block that needs elapsed or absolute time (for example, a Discrete Time Integrator).

The asynchronous function call subsystem has one global variable, `clockTick#` (where # is the task ID associated with the subsystem). This variable stores absolute time for the asynchronous task. There are two ways timing can be handled:

- If the time source is set to `SS_TIMESOURCE_BASERATE`, the code generator produces timer code in the function call subsystem, updating the clock tick variable from the base rate clock tick. Data integrity is maintained if the same priority is assigned to TriggerA and TriggerB.
- If the time source is `SS_TIMESOURCE_SELF`, generated code for both TriggerA and TriggerB updates the same clock tick variable from the hardware clock.

The word size of the clock tick variable can be set directly or be established according to setting of model configuration parameter **Application lifespan (days)** and the timer resolution set by the TriggerA and TriggerB S-functions (which must be the same). See “Timers in Asynchronous Tasks” on page 30-38 and “Control Memory Allocation for Time Counters” on page 76-10 for more information.

Protect Data Integrity with volatile Keyword

When you select block parameter **Ensure data integrity during data transfer**, the code generated for a Rate Transition block defines `volatile` global buffers and semaphores and uses them to protect the integrity of the transferred data. For additional protection, or for protection without a Rate Transition block, you can explicitly apply `volatile` to the transferred data. For more information, see “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.

See Also

More About

- “Handle Rate Transitions”
- “Time-Based Scheduling and Code Generation”
- “Asynchronous Support Limitations”

Timers in Asynchronous Tasks

An ISR can set a source for absolute time. This is done with the function `ssSetTimeSource`. The function `ssSetTimeSource` cannot be called before `ssSetOutputPortWidth` is called. If this occurs, the program will come to a halt and generate an error message. `ssSetTimeSource` has the following three options:

- `SS_TIMESOURCE_SELF`: Each generated ISR maintains its own absolute time counter, which is distinct from a periodic base rate or subrate counters in the system. The counter value and the timer resolution value (specified with the block parameter **Timer resolution (seconds)** of the Async Interrupt block) are used by downstream blocks to determine absolute time values required by block computations.
- `SS_TIMESOURCE_CALLER`: The ISR reads time from a counter maintained by its caller. Time resolution is thus the same as its caller's resolution.
- `SS_TIMESOURCE_BASERATE`: The ISR can read absolute time from the model's periodic base rate. Time resolution is thus the same as its base rate resolution.

Note The operating system integration techniques that are demonstrated in this section use blocks that are in the Interrupt Templates block library. The blocks in that library provide starting point examples to help you develop custom blocks for a target environment.

By default, the counter is implemented as a 32-bit unsigned integer member of the `Timing` substructure of the real-time model structure. For a target that supports the `rtModel` data structure, when the time data type is not set by using `ssSetAsyncTimeDataType`, the counter word size is determined by the setting for model configuration parameter **Application lifespan (days)**. As an example (from ERT target code),

```
/* Real-time Model Data Structure */
struct _RT_MODEL_elapseTime_exp_Tag {
   const char *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint32_T clockTick2;
  } Timing;
};
```

The example omits unused fields in the `Timing` data structure (a feature of ERT target code not found in GRT). For a target that supports the `rtModel` data structure, the counter word size is determined by the setting of model configuration parameter **Application lifespan (days)**.

By default, the `vxlib1` library blocks for the example RTOS (VxWorks) set the timer source to `SS_TIMESOURCE_SELF` and update their counters by using the system call `tickGet`. `tickGet` returns a timer value maintained by the RTOS kernel. The maximum word size for the timer is `UINT32`. The following example shows a generated call to `tickGet`.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
```

```
{

 /* Use tickGet() as a portable tick counter example. A much
    higher resolution can be achieved with a hardware counter */
 rtM->Timing.clockTick2 = tickGet();
. . .
```

The `tickGet` call is supplied only as an example. It can (and in many instances should) be replaced by a timing source that has better resolution. If you are implementing a custom asynchronous block for an RTOS other than the example RTOS (VxWorks), you should either generate an equivalent call to the target RTOS, or generate code to read a timer register on the target hardware.

Change the setting for the block parameter **Timer resolution (seconds)** for your Async Interrupt block to match the resolution of timing source for you target computer.

The counter is updated at interrupt level. Its value represents the tick value of the timing source at the most recent execution of the ISR. The rate of this timing source is unrelated to sample rates in the model. In fact, typically it is faster than the model's base rate. Select the timer source and set its rate and resolution based on the expected rate of interrupts to be serviced by the Async Interrupt block.

For an example of timer code generation, see “Async Interrupt Block Implementation” on page 30-40.

See Also

Related Examples

- “Generate Interrupt Service Routines” on page 30-4
- “Spawn and Synchronize Execution of RTOS Task” on page 30-11
- “Create a Customized Asynchronous Library” on page 30-40
- “Import Asynchronous Event Data for Simulation” on page 30-47

More About

- “Absolute and Elapsed Time Computation”
- “Time-Based Scheduling and Code Generation”
- “Asynchronous Events”
- “Asynchronous Support Limitations”

Create a Customized Asynchronous Library

This topic describes how to implement asynchronous blocks for use with your target RTOS, using the Async Interrupt and Task Sync blocks as a starting point. Rate Transition blocks are target-independent, so you do not need to develop customized rate transition blocks.

Note The operating system integration techniques that are demonstrated in this section use blocks that are in the Interrupt Templates block library. The blocks in that library provide starting point examples to help you develop custom blocks for a target environment.

About Implementing Asynchronous Blocks

You can customize the asynchronous library blocks by modifying the block implementation. These files are

- The block's underlying S-function MEX-file
- The TLC files that control code generation of the block

In addition, you need to modify the block masks to remove references specific to the example RTOS (VxWorks) and to incorporate parameters required by your target RTOS.

Custom block implementation is an advanced topic, requiring familiarity with the Simulink MEX S-function format and API, and with the Target Language Compiler (TLC). These topics are covered in the following documents:

- Simulink topics “What Is an S-Function?”, “Use S-Functions in Models”, “How S-Functions Work”, and “Implementing S-Functions” describe MEX S-functions and the S-function API in general.
- The “Inlining S-Functions”, “Inline C MEX S-Functions”, and “S-Functions and Code Generation” describe how to create a TLC block implementation for use in code generation.

The following sections discuss the C/C++ and TLC implementations of the asynchronous library blocks, including required `SimStruct` macros and functions in the TLC asynchronous support library (`asynclib.tlc`).

Async Interrupt Block Implementation

The source files for the Async Interrupt block are located in *`matlabroot`*`/rtw/c/tornado/devices` (open):

- `vxinterrupt1.c`: C MEX-file source code, for use in configuration and simulation
- `vxinterrupt1.tlc`: TLC implementation, for use in code generation
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in “asynclib.tlc Support Library” on page 30-45.

C MEX Block Implementation

Most of the code in `vxinterrupt1.c` performs ordinary functions that are not related to asynchronous support (for example, obtaining and validating parameters from the block mask, marking parameters nontunable, and passing parameter data to the *`model`*`.rtw` file).

The `mdlInitializeSizes` function uses special `SimStruct` macros and `SS_OPTIONS` settings that are required for asynchronous blocks, as described below.

Note that the following macros cannot be called before `ssSetOutputPortWidth` is called:

- `ssSetTimeSource`
- `ssSetAsyncTimerAttributes`
- `ssSetAsyncTimerResolutionEl`
- `ssSetAsyncTimerDataType`
- `ssSetAsyncTimerDataTypeEl`
- `ssSetAsyncTaskPriorities`
- `ssSetAsyncTaskPrioritiesEl`

If one of the above macros is called before `ssSetOutputPortWidth`, the following error message appears:

```
SL_SfcnMustSpecifyPortWidthBfCallSomeMacro {
S-function '%s' in '%<BLOCKFULLPATH>'
must set output port %d width using
ssSetOutputPortWidth before calling macro %s
 }
```

ssSetAsyncTimerAttributes

`ssSetAsyncTimerAttributes` declares that the block requires a timer, and sets the resolution of the timer as specified for block parameter **Timer resolution (seconds)**.

The function prototype is

```
ssSetAsyncTimerAttributes(SimStruct *S, double res)
```

where

- S is a Simstruct pointer.
- Parameter **Timer resolution (seconds)** is set to `res`.

The following code excerpt shows the call to ssSetAsyncTimerAttributes.

```
/* Setup Async Timer attributes */
ssSetAsyncTimerAttributes(S,mxGetPr(TICK_RES)[0]);
```

ssSetAsyncTaskPriorities

`ssSetAsyncTaskPriorities` sets the Simulink task priority for blocks executing at each interrupt level, as specified for block parameter **Simulink task priority**.

The function prototype is

```
ssSetAsyncTaskPriorities(SimStruct *S, int numISRs,
                         int *priorityArray)
```

where

- `S` is a `SimStruct` pointer.

- `numISRs` is the number of interrupts specified for block parameter **VME interrupt number(s)** .
- `priorityarray` is an integer array containing the interrupt numbers specified for block parameter **VME interrupt number(s)**.

The following code excerpt shows the call to `ssSetAsyncTaskPriorities`:

```
/* Setup Async Task Priorities */
    priorityArray = malloc(numISRs*sizeof(int_T));
    for (i=0; i<numISRs; i++) {
        priorityArray[i] = (int_T)(mxGetPr(ISR_PRIORITIES)[i]);
    }
    ssSetAsyncTaskPriorities(S, numISRs, priorityArray);
    free(priorityArray);
    priorityArray = NULL;
}
```

SS_OPTION Settings

The code excerpt below shows the `SS_OPTION` settings for `vxinterrupt1.c`. `SS_OPTION_ASYNCHRONOUS_INTERRUPT` should be used when a function call subsystem is attached to an interrupt. For more information, see the documentation for `SS_OPTION` and `SS_OPTION_ASYNCHRONOUS` in *`matlabroot`*`/simulink/include/simstruc.h`.

```
ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
              SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
              SS_OPTION_ASYNCHRONOUS_INTERRUPT));
```

If an S-function specifies the `SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME` option and inherits a sample time of `Inf`, the code generator determines how to produce code for the block based on whether the block is invariant. A block is invariant if its port signals are invariant. A signal is invariant if it has a constant value during the entire simulation. If you specify a Constant block sample time, do not assume that the port signals are invariant. For more information, see “Inline Invariant Signals”. If the block is not invariant, the code generator produces code only in the initialize entry-point function. If the block is invariant, the code generator does not produce code for the block.

TLC Implementation

This section discusses each function of `vxinterrupt1.tlc`, with an emphasis on target-specific features that you will need to change to generate code for your target RTOS.

Generate #include Directives

`vxinterrupt1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

For each connected output of the Async Interrupt block, `BlockInstanceSetup` defines a function name for the corresponding ISR in the generated code. The functions names are of the form

`isr_`*`num`*`_vec_`*`offset`*

where *num* is the ISR number defined for block parameter **VME interrupt number(s)**, and *offset* is an interrupt table offset defined by block parameter **VME interrupt vector offset(s)**.

In a custom implementation, this naming convention is optional.

The function names are cached for use by the `Outputs` function, which generates the actual ISR code.

Outputs Function

`Outputs` iterates over the connected outputs of the Async Interrupt block. An ISR is generated for each such output.

The ISR code is cached in the `"Functions"` section of the generated code. Before generating the ISR, `Outputs` does the following:

- Generates a call to the downstream block (cached in a temporary buffer).
- Determines whether the ISR should be locked or not (as specified by block parameter **Preemption Flag(s)**).
- Determines whether the block connected to the Async Interrupt block is a Task Sync block. (This information is obtained by using the `asynclib` calls `LibGetFcnCallBlock` and `LibGetBlockAttribute`.) If so,
  - The preemption flag for the ISR must be set to `1`. An error results otherwise.
  - The RTOS (VxWorks) calls to save and restore floating-point context are generated, unless the user has configured the model for integer-only code generation.

When generating the ISR code, `Outputs` calls the `asynclib` function `LibNeedAsyncCounter` to determine whether a timer is required by the connected subsystem. If so, and if the time source is set to be `SS_TIMESOURCE_SELF` by `ssSetTimeSource`, `LibSetAsyncCounter` is called to generate an RTOS (VxWorks) `tickGet` function call and update the counter. In your implementation, you should generate either an equivalent call to the target RTOS, or generate code to read the a timer register on the target hardware.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls (`int_connect` and `sysInt_Enable`) to connect and enable each ISR. You should replace this with calls to your target RTOS.

Terminate Function

The `Terminate` function generates the call `sysIntDisable` to disable each ISR. You should replace this with calls to your target RTOS.

Task Sync Block Implementation

The source files for the Task Sync block are located in *matlabroot*`/rtw/c/tornado/devices` (open). They are

- `vxtask1.cpp`: MEX-file source code, for use in configuration and simulation.
- `vxtask1.tlc`: TLC implementation, for use in code generation.
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in “asynclib.tlc Support Library” on page 30-45.

C MEX Block Implementation

Like the Async Interrupt block, the Task Sync block sets up a timer, in this case with a fixed resolution. The priority of the task associated with the block is obtained from block parameter **Simulink task priority**. The `SS_OPTION` settings are the same as those used for the Async Interrupt block.

```
ssSetAsyncTimerAttributes(S, 0.01);

priority  = (int_T) (*(mxGetPr(PRIORITY)));
ssSetAsyncTaskPriorities(S,1,&priority);

ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_ASYNCHRONOUS |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME));
}
```

TLC Implementation

Generate #include Directives

`vxtask1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

The `BlockInstanceSetup` function derives the task name, block name, and other identifiers used later in code generation. It also checks for and warns about unconnected block conditions, and generates a storage declaration for a semaphore (`stopSem`) that is used in case of interrupt overflow conditions.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls to define storage for the semaphore that is used in management of the task spawned by the Task Sync block. Depending on the value of the `CodeFormat` TLC variable of the target, either a static storage declaration or a dynamic memory allocation call is generated. This function also creates a semaphore (`semBCreate`) and spawns an RTOS task (`taskSpawn`). You should replace these with calls to your target RTOS.

Outputs Function

The `Outputs` function generates an example RTOS (VxWorks) task that waits for a semaphore. When it obtains the semaphore, it updates the block's tick timer and calls the downstream subsystem code, as described in “Spawn and Synchronize Execution of RTOS Task” on page 30-11. `Outputs` also generates code (called from interrupt level) that grants the semaphore.

Terminate Function

The `Terminate` function generates the example RTOS (VxWorks) call `taskDelete` to end execution of the task spawned by the block. You should replace this with calls to your target RTOS.

Note also that if the target RTOS has dynamically allocated memory associated with the task, the `Terminate` function should deallocate the memory.

asynclib.tlc Support Library

`asynclib.tlc` is a library of TLC functions that support the implementation of asynchronous blocks. Some functions are specifically designed for use in asynchronous blocks. For example, `LibSetAsyncCounter` generates a call to update a timer for an asynchronous block. Other functions are utilities that return information required by asynchronous blocks (for example, information about connected function call subsystems).

The following table summarizes the public calls in the library. For details, see the library source code and the `vxinterrupt1.tlc` and `vxtask1.tlc` files, which call the library functions.

Summary of asynclib.tlc Library Functions

Function	Description
`LibBlockExecuteFcnCall`	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem.
`LibGetBlockAttribute`	Returns a field value from a block record.
`LibGetFcnCallBlock`	Given an S-Function block and call index, returns the block record for the downstream function call subsystem block.
`LibGetCallerClockTickCounter`	Provides access to the time counter of an upstream asynchronous task.
`LibGetCallerClockTickCounter-HighWord`	Provides access to the high word of the time counter of an upstream asynchronous task.
`LibManageAsyncCounter`	Determines whether an asynchronous task needs a counter and manages its own timer.
`LibNeedAsyncCounter`	If the calling block requires an asynchronous counter, returns `TLC_TRUE`, otherwise returns `TLC_FALSE`.
`LibSetAsyncClockTicks`	Returns code that sets `clockTick` counters that are to be maintained by the asynchronous task.
`LibSetAsyncCounter`	Generates code to set the tick value of the block's asynchronous counter.
`LibSetAsyncCounterHighWord`	Generates code to set the tick value of the high word of the block's asynchronous counter

See Also

More About

- “Asynchronous Events”
- “Generate Interrupt Service Routines”
- “Spawn and Synchronize Execution of RTOS Task”
- “Pass Asynchronous Events in RTOS as Input to a Referenced Model”
- “Timers in Asynchronous Tasks”
- “Import Asynchronous Event Data for Simulation”
- “Rate Transitions and Asynchronous Blocks”

- “Asynchronous Support Limitations”

Import Asynchronous Event Data for Simulation

Capabilities

You can import asynchronous event data into a function-call subsystem via an Inport block. For standalone fixed-step simulations, you can specify:

- The time points at which each asynchronous event occurs
- The number of asynchronous events at each time point

Input Data Format

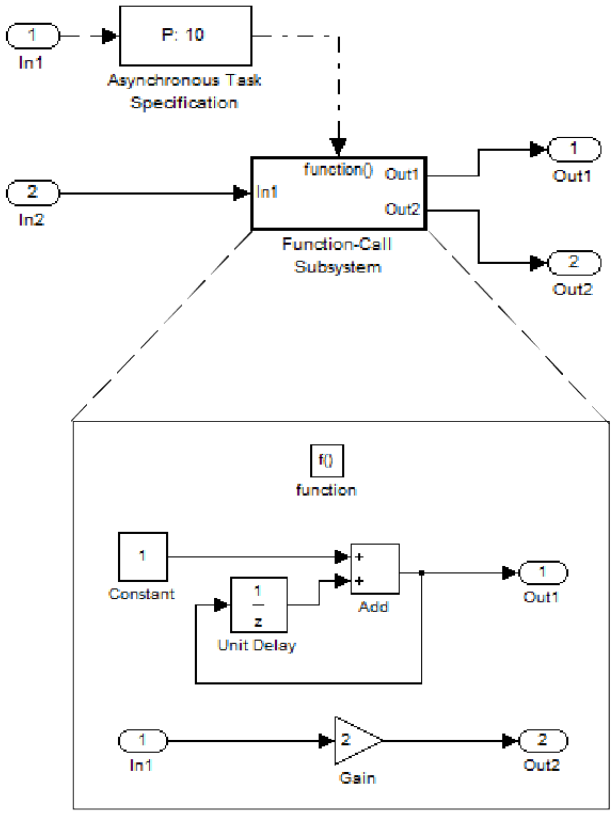
You can enter your asynchronous data at the MATLAB command line or on the **Data Import/Export** pane of the Configuration Parameters dialog box. In either case, a number of restrictions apply to the data format.

- The expression for parameter model configuration parameter **Input** must be a comma-separated list of tables.
- The table corresponding to the input port outputting asynchronous events must be a column vector containing time values for the asynchronous events.
  - The time vector of the asynchronous events must be of double data type and monotonically increasing.
  - Time data must be integer multiples of the model step size.
  - To specify multiple function calls at a given time step, you must repeat the time value accordingly. In other words, if you wish to specify three asynchronous events at $t = 1$ and two events at $t = 9$, then you must list `1` three times and `9` twice in your time vector. (`t = [1 1 1 9 9]'`)
- The table corresponding to normal data input port can be of another supported format.

See “Load Data to Root-Level Input Ports” for more information.

Example

In this model, a function-call subsystem is used to track the total number of asynchronous events and to multiply a set of input values by 2.



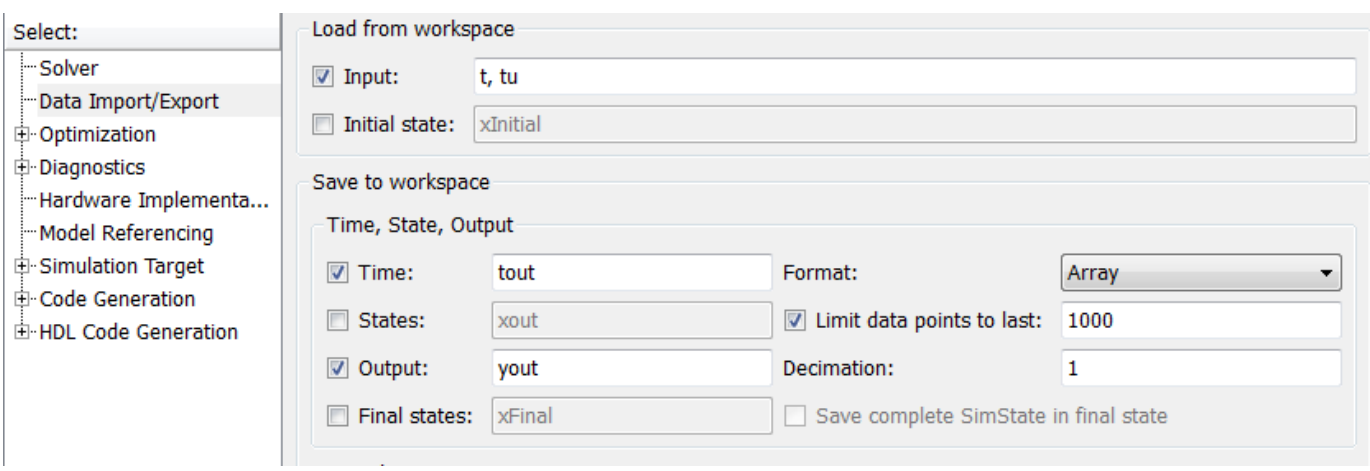
1 To input data by using the Model Configuration Parameters dialog box,

   a Select **Input**.

   b For this example, enter the following command in the MATLAB window:

```
>> t = [1 1 5 9 9 9]', tu = [[0:10]' [0:10]']
```

   Alternatively, you can enter the data as *t, tu* in the Data Import/Export pane:



   Here, *t* is a column vector containing the times of asynchronous events for Inport block In1 while *tu* is a table of input values versus time for Inport block In2.

2. By default, the **Time** and **Output** parameters are selected and the output variables are named *tout* and *yout*.
3. Simulate the model.
4. Display the output by entering `[tout yout]` at the MATLAB command line and obtain:

```
ans =

     0     0    -1
     1     2     2
     2     2     2
     3     2     2
     4     2     2
     5     3    10
     6     3    10
     7     3    10
     8     3    10
     9     6    18
    10     6    18
```

Here the first column contains the simulation times.

The second column represents the output of Out1 — the total number of asynchronous events. Since the function-call subsystem is triggered twice at $t = 1$, the output is 2. It is not called again until $t = 5$, and so does not increase to 3 until then. Finally, it is called three times at 9, so it increases to 6.

The third column contains the output of Out2 obtained by multiplying the input value at each asynchronous event time by 2. At any other time, the output is held at its previous value

See Also

More About

- "Asynchronous Events"
- "Pass Asynchronous Events in RTOS as Input to a Referenced Model"
- "Load Data to Root-Level Input Ports"
- "Rate Transitions and Asynchronous Blocks"

Asynchronous Support Limitations

In this section...

"Asynchronous Task Priority" on page 30-50
"Convert an Asynchronous Subsystem into a Model Reference" on page 30-50

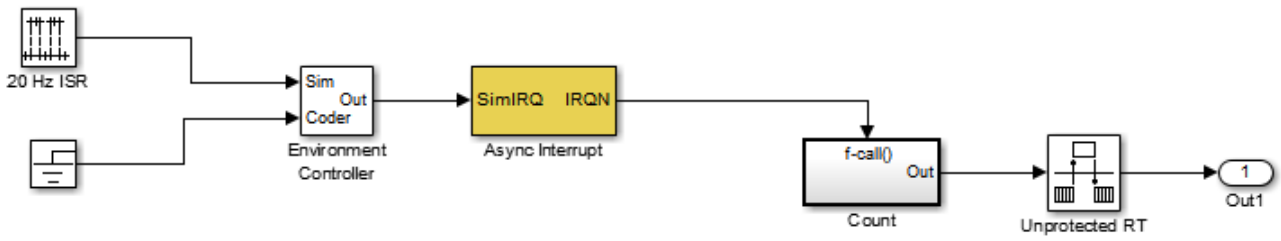
Asynchronous Task Priority

The Simulink product does not simulate asynchronous task behavior. Although you can specify a task priority for an asynchronous task represented in a model with the Task Sync block, the priority setting is for code generation purposes only and is not honored during simulation.

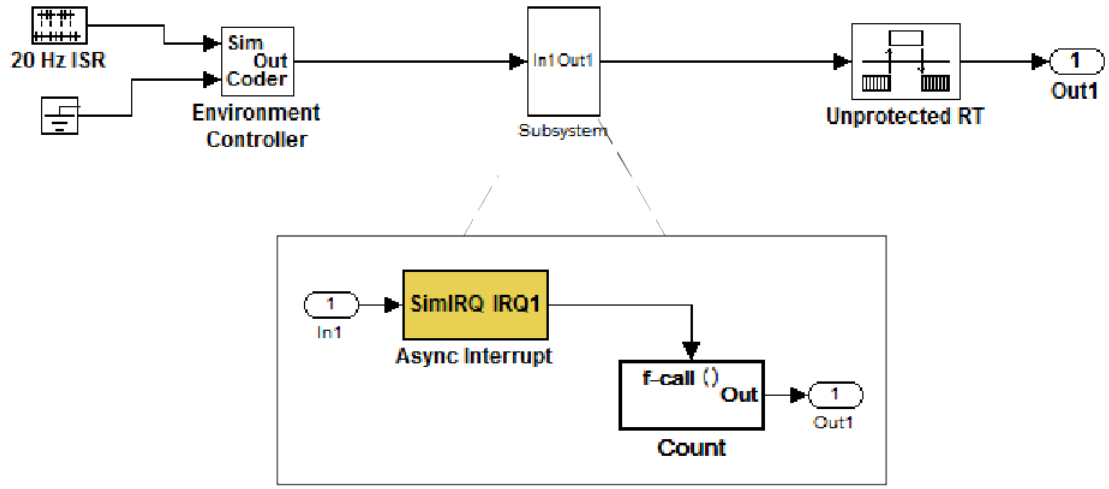
Convert an Asynchronous Subsystem into a Model Reference

You can use the Asynchronous Task Specification block to specify an asynchronous function-call input to a referenced model. However, you must convert the Async Interrupt and Function-Call blocks into a subsystem and then convert the subsystem into a referenced model.

This example shows how to complete a conversion.



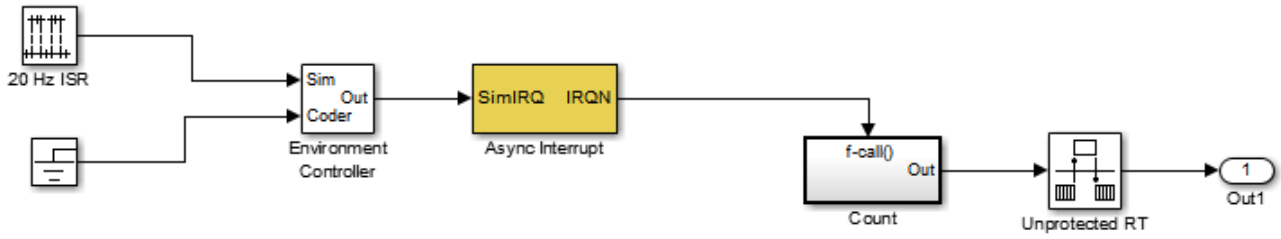
1. Convert the Async Interrupt and Count blocks into a subsystem. Select both blocks and right-click Count. From the menu, select **Subsystem & Model Reference > Create Subsystem from Selection**.

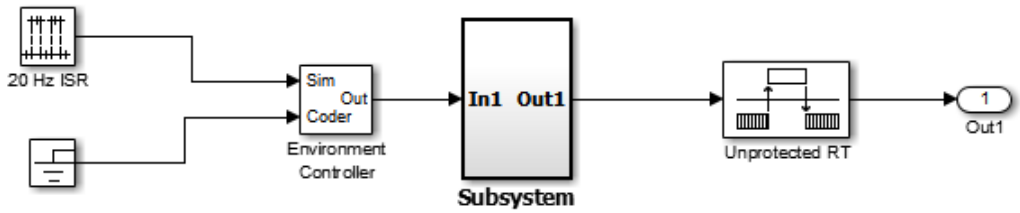


2 To prepare for converting the new subsystem to a Model block, set these model configuration parameters in the top model.

- Set **Multitask data transfer** to `error`.
- Set **Multitask conditionally executed subsystem** to `error`.
- Set **Bus signal treated as vector** to `error`.
- Set **Context-dependent inputs** to `error`.
- Set **Multitask data store** to `error`.
- Set **Underspecified initialization detection** to `Simplified`.
- If your model is large or complex, run the Model Advisor checks in the folder “Migrating to Simplified Initialization Mode Overview” and make the suggested changes.

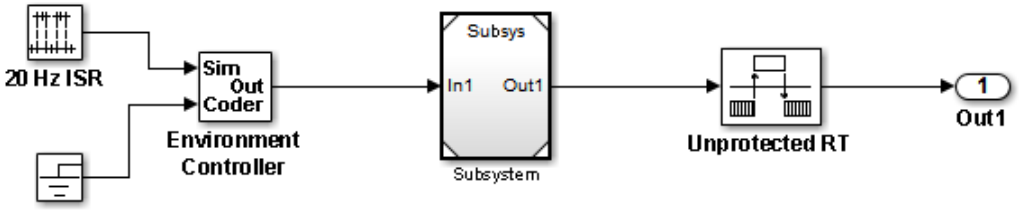
3 Convert the subsystem to an atomic subsystem. Select the subsystem block parameter **Treat as atomic unit**.



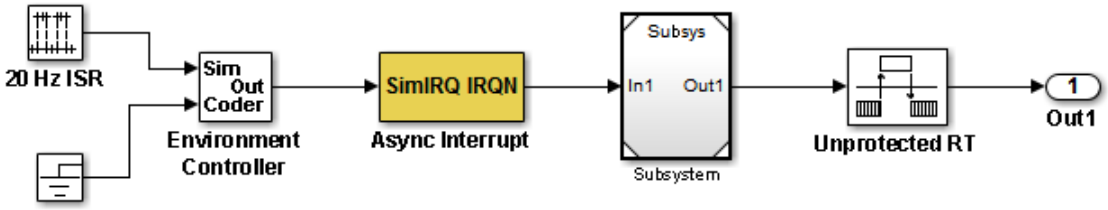


4 Convert the subsystem to a Model block. Right-click the subsystem and select **Subsystem & Model Reference > Convert Subsystem to > Referenced Model**. The Model Reference Conversion Advisor opens.

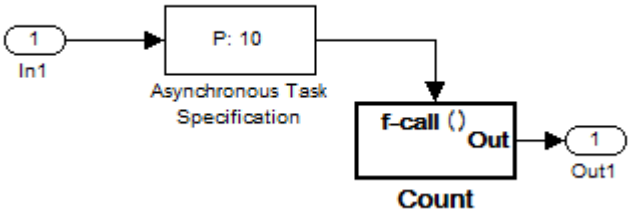
5 Replace the subsystem in the top model with the new Model block.



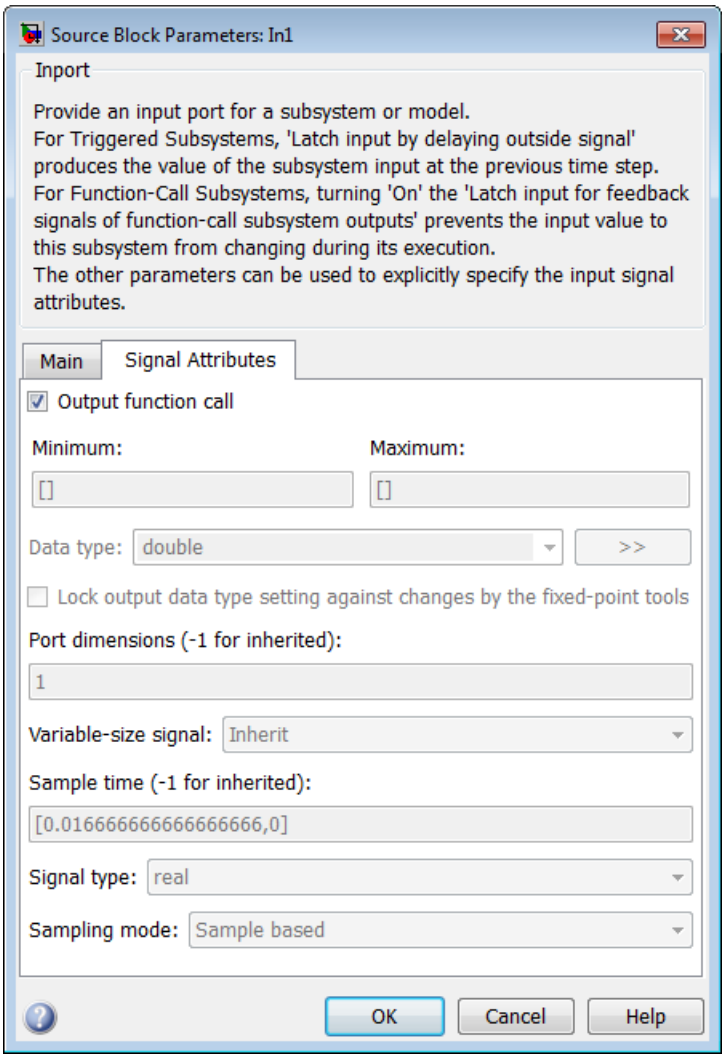
6 Move the Async Interrupt block from the referenced model to the top model, before the Model block.



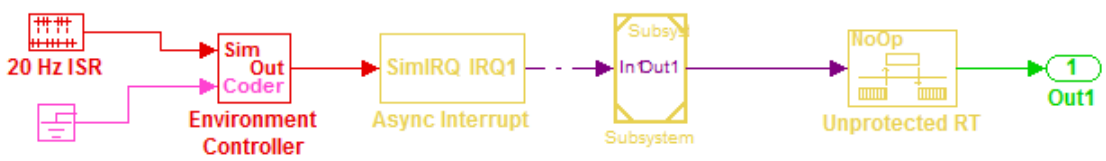
7 Insert an Asynchronous Task Specification block in the referenced model. Set the priority of the Asynchronous Task Specification block. (For more information on setting the priority, see Asynchronous Task Specification.)



8 In the referenced model, double-click the input port to open its Source Block Parameters dialog box. In the **Signal Attributes** tab, select the **Output function call** parameter. Click **OK**.



9 Save your model and then update the diagram to verify your settings.



See Also

More About

- “Asynchronous Events”

Scheduling Considerations in Embedded Coder

- “Use Discrete and Continuous Time” on page 31-2
- “Optimize Multirate Multitasking Execution for RTOS Run-Time Environments” on page 31-4

Use Discrete and Continuous Time

In this section...

"Support for Discrete and Continuous Time Blocks" on page 31-2
"Support for Continuous Solvers" on page 31-2
"Support for Stop Time" on page 31-2

Support for Discrete and Continuous Time Blocks

The ERT system target file supports code generation for discrete and continuous time blocks. If you select model configuration parameter **Support: continuous time**, you can use these blocks in your models, without restriction.

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

```
showblockdatatypetable
```

Support for Continuous Solvers

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

Note Custom targets must be modified to support continuous time. The required modifications are described in "Customize System Target Files".

Support for Stop Time

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

- **External mode**
- **MAT-file logging**
- **Classic call interface**

Otherwise, the executable runs indefinitely.

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

See Also

More About

- “Time-Based Scheduling and Code Generation” on page 29-2
- “Configure Time-Based Scheduling” on page 29-33
- “Sample Times in Subsystems”
- “Sample Times in Systems”

Optimize Multirate Multitasking Execution for RTOS Run-Time Environments

Using the `rtmStepTask` macro, run-time environments that employ task management mechanisms of an real-time operating system (RTOS)—for example, VxWorks—can improve performance of generated code by eliminating redundant scheduling calls during the execution of tasks in a multirate, multitasking model, The following sections describe implementation details.

Use rtmStepTask

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

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

The arguments are:

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

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

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

Schedule Code for Real-time Model without an RTOS

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

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

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

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

For bareboard targets that cannot rely on an external RTOS, the event flags are mandatory to allow overlapping task preemption. However, an RTOS target uses the operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant.

Schedule Code for Multirate Multitasking on an RTOS

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

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

Suppress Redundant Scheduling Calls

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

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

See Also

More About

- “Time-Based Scheduling and Code Generation” on page 29-2
- “Modeling for Multitasking Execution” on page 29-9

Representing a Software Architecture by Creating Code Generation Definitions

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Create Data Interface Configuration Programmatically” on page 32-5
- “Create a Service Interface Configuration” on page 32-11
- “Deploy Code Generation Definitions” on page 32-16
- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24
- “Manage an Embedded Coder Dictionary” on page 32-27
- “Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings” on page 32-31
- “Flexible Storage Class for Different Model Hierarchy Contexts” on page 32-34
- “Separate C Code Interface Configuration Definition from Model-Specific Configurations” on page 32-38

Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture

In a team or large organization, to enable multiple users to generate code that conforms to a standard architecture, you can create and share code generation definitions, such as service interfaces, storage classes, with those users. When configuring code generation settings for different models, the users can apply the definitions to data and function elements in the models, generating standardized code.

Create Code Definitions for Use in the Code Mappings Editor

To control the appearance of model elements in the generated code, you apply code definitions to the elements by using the Code Mappings editor. To create code definitions that you and other users can use in the Code Mappings editor, use a code interface configuration in an Embedded Coder Dictionary. Decide what type of code interface configuration to use by considering what type of code you want to generate from your model and how you want to deploy the code in the target execution environment.

- Data interface configuration — The code generator creates a program intended to run on the target device. The generated code uses the data interfaces that you define.
- Service interface configuration — The code generator creates an algorithm that you intend to deploy within a larger application for a target platform. The platform middleware calls the algorithm code and provides the services for the target device. The generated code calls the services according to your definitions in the service interface configuration. To create a service interface configuration, you must use an Embedded Coder Dictionary that is saved in a SLDD file.

When you create definitions in an Embedded Coder Dictionary, decide where to save the definitions by considering how you intend to use the definitions and the type of code interface configuration. This table lists where to store the code definitions.

Code Interface Configuration Type	Usage of Definitions	Location for Embedded Coder Dictionary
Service interface configuration	Multiple models or one model	SLDD file that all of the models are configured to use for the coder dictionary
Data interface configuration	Multiple models — sharing a set of code definitions to standardize code	SLDD file that all of the models are configured to use for the coder dictionary
Data interface configuration	One model — prototyping generated code by using one model	Local Embedded Coder Dictionary of the model.

To use an Embedded Coder Dictionary and for its limitations with respect to code generation definitions, see Embedded Coder Dictionary.

Create Code Definitions for External Data Objects

To create definitions for data objects that you store in the base workspace or in a Simulink data dictionary, use the Custom Storage Class designer to create a definition in a package. For these data

objects, you apply code generation settings to the data object instead of applying them in the mapping of a model. When an item in a model uses the data object and its code generation settings, the code mapping for the item reflects the specification of the data object. For more information, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14 and “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Use Package Code Definitions in Embedded Coder Dictionaries

If you want to use the same code generation definition in the Code Mappings editor and in data objects, you do not need to store one copy of the definition in an Embedded Coder Dictionary and another copy in a package. Instead, store the definition in a package (by using the Custom Storage Class Designer). Then, configure one or more Embedded Coder Dictionaries to refer to the definition in the package. With this technique, when you want to make changes to the definition, you make the changes in only one place, the package.

To create a code generation definition in a package by using the Custom Storage Class Designer, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14. Then, to configure an Embedded Coder Dictionary to refer to the package, see “Refer to Code Generation Definitions in a Package”.

For more information about maintaining definitions in packages and dictionaries, see “Migrate Package Definitions to Embedded Coder Dictionary to Use in Code Mappings” on page 32-27.

Constrain Use of Storage Class Code Mappings

When defining a storage class in the Embedded Coder Dictionary, you can specify whether users can map the storage class to parameters, signals, or parameters and signals. To constrain the use of a storage class, the **Data Initialization** property must be set to `Auto` or `None`. Then, in the Property Inspector, under **Allowed Usage**, select **Parameters**, **Signals**, or **Parameters** and **Signals**. For more information, see **Embedded Coder Dictionary**.

Deploy Code Generation Definitions to Users

In a large organization with multiple models and users, you can share code generation definitions by storing them in a mutually accessible location. The process of sharing code definitions is called deployment. For more information, see “Deploy Code Generation Definitions” on page 32-16.

Interact with Code Definitions in Data Interface Configuration Programmatically

When you use a data interface configuration, you can create and interact with code definitions by using the Embedded Coder Dictionary API.

1. Represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use the object to access the sections of the dictionary: Storage Classes, Memory Sections, and Function Customization Templates. If you have a package of code definitions, use the `coder.Dictionary` object to load and refresh the package in the dictionary.
2. Represent a section of the dictionary by using a `coder.dictionary.Section` object. Use the section to access code definitions.
3. Represent a code definition by using a `coder.dictionary.Entry` object. Use the entry to interact with the code definition and its properties.

You can also use functions such as `coder.dictionary.copy` and `coder.dictionary.move` to perform operations on Embedded Coder Dictionaries. For more information, see “Create Data Interface Configuration Programmatically” on page 32-5.

See Also

Embedded Coder Dictionary | **Code Mappings Editor – C**

Related Examples

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14
- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24
- “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

Create Data Interface Configuration Programmatically

This example shows how to programmatically create a data interface configuration in an Embedded Coder Dictionary. To automate the creation of an Embedded Coder Dictionary in a script or to create a data interface configuration programmatically, use this programming interface.

In this example, you create a shared Embedded Coder Dictionary in an SLDD file so that multiple models can use the code definitions. In the data interface configuration, you create code definitions that control the naming and placement of global data and functions. You also use a memory section definition that includes a pragma to place the data and functions in fast memory. To verify the architecture that you set up through the code definitions, apply the definitions to a model, and then generate code.

Set Up Embedded Coder Dictionary

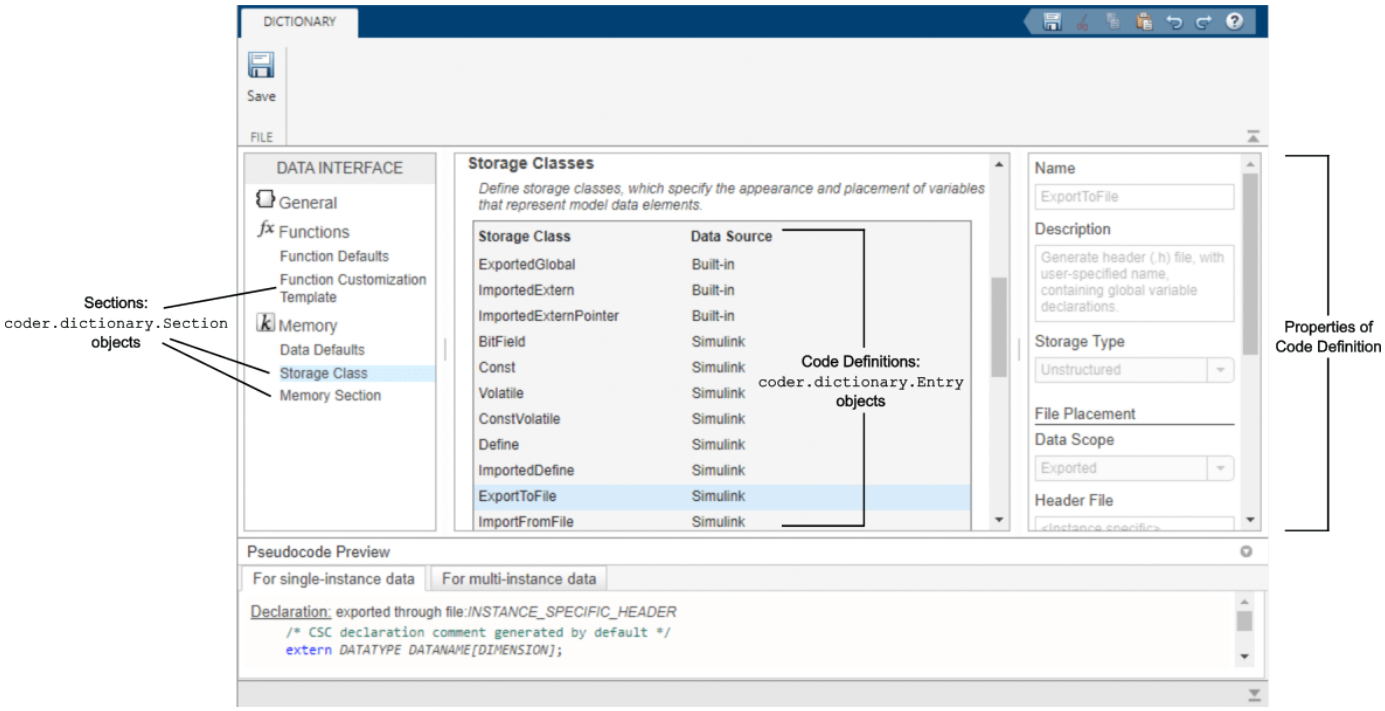
Create a Simulink data dictionary (SLDD file) to store the code definitions. Storing the definitions in a shared SLDD file enables you to use the definitions in multiple models.

```
dictionaryFile = Simulink.data.dictionary.create('codeDefinitions.sldd');
```

Create an Embedded Coder Dictionary in the SLDD file. When you create the dictionary, represent it with a `coder.Dictionary` object. Use the object to perform operations on the entire Embedded Coder Dictionary and to access sections of it.

```
codeDictionary = coder.dictionary.create('codeDefinitions.sldd');
```

The `coder.Dictionary` object contains three `coder.dictionary.Section` objects that represent the sections of an Embedded Coder Dictionary: Storage Classes, Memory Sections, and Function Customization Templates. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects that represent the definitions in that section. To interact with a definition and access its properties, use the `coder.dictionary.Entry` object that represents it.



When you create an Embedded Coder Dictionary, the dictionary loads definitions from the `Simulink` package so that the Embedded Coder Dictionary then contains built-in definitions. If you have custom code definitions stored in a package, load that package to the dictionary. When you use the Embedded Coder Dictionary to configure the code interface for a model, you can apply definitions from packages that you loaded.

Memory Sections

To create memory section definitions, add entries to the Memory Sections section. Storage classes and function customization templates residing in the same dictionary can use these memory sections. For this example, add a memory section named `FastMem`, which allocates memory by using a pragma. When you apply the memory section to a storage class or function template, their generated definitions and declarations are in the memory section.

```
memorySections = getSection(codeDictionary,'MemorySections');
msFast = addEntry(memorySections,'FastMem');
set(msFast,'PreStatement','#pragma begin FAST');
set(msFast,'Comment','/*Fast onchip RAM*/');
set(msFast,'PostStatement','#pragma end FAST');
```

Storage Classes

For this example, create a storage class named `ExportToPrivateHeader`, which generates a global variable declaration in the header file `$R_private.h` and definition in `$R_private.c`. The token `$R` signifies the name of the root model for which you generate code. To use a variable in external code, apply this storage class to the data element and include the header file in the external code.

```
storageClasses = getSection(codeDictionary,'StorageClasses');
exportToPrivateH = addEntry(storageClasses,'ExportToPrivateHeader');
set(exportToPrivateH,'HeaderFile','$R_private.h','DataScope','Exported');
```

To apply the memory section `FastMem` to the storage class, use the `coder.dictionary.Entry` object that represents the memory section.

```
set(exportToPrivateH,'MemorySection',msFast);
```

Create another storage class definition named `ImportFromHeader` for data that external code defines. Because the storage class has `'DataScope'` set to `'Imported'`, the generated code reads and writes to the variable defined by your external code.

```
importFromH = addEntry(storageClasses,'ImportFromHeader');
set(importFromH,'DataScope','Imported','HeaderFile','$R_input.h','DataInit','Dynamic');
```

Function Customization Templates

To create a function customization template, create another `coder.dictionary.Section` object that represents the Function Customization Templates section. Add an entry to this section, which represents a definition that controls the appearance of generated entry-point functions. Apply the memory section `FastMem` to the function template.

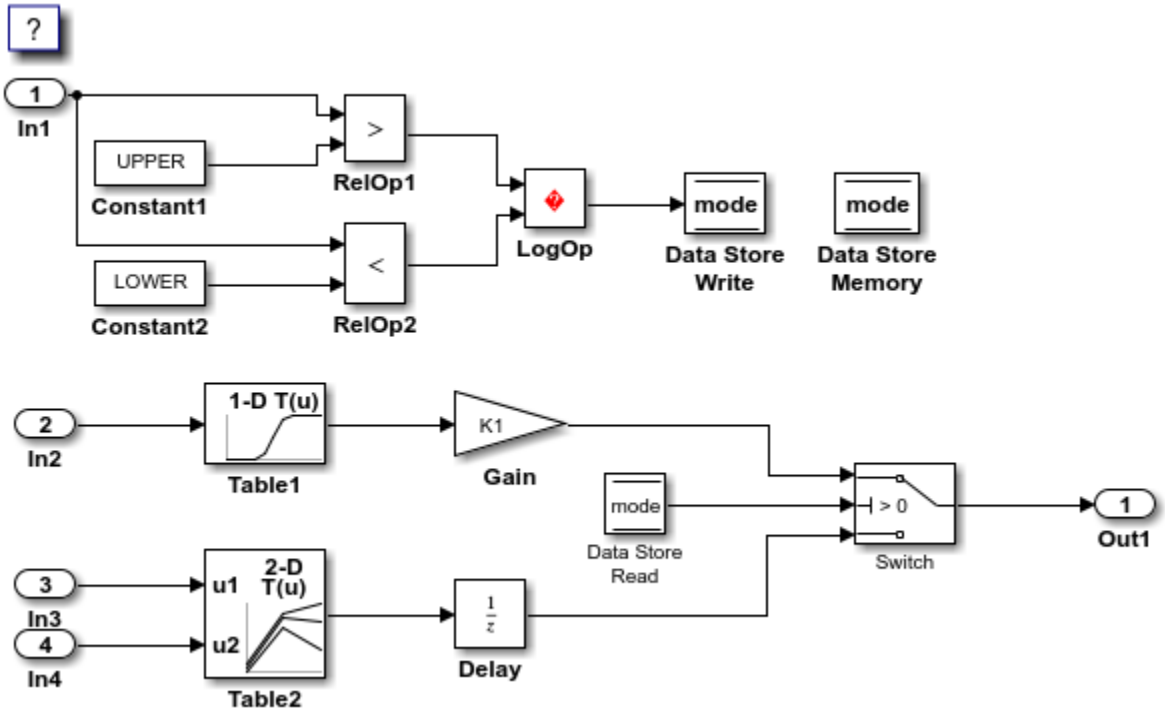
```
functionTemplates = getSection(codeDictionary,'FunctionCustomizationTemplates');
fcGlobal = addEntry(functionTemplates,'GlobalFunctions');
set(fcGlobal,'FunctionName','$R$N');
set(fcGlobal,'MemorySection',msFast);
saveChanges(dictionaryFile);
```

Apply Code Generation Definitions

Confirm the settings of your code definitions by applying them to a model and generating code. You can configure the model code interface to use the code definitions programmatically or by using the Code Mappings editor.

Open the model `rtwdemo_configinterface` and attach the data dictionary that you created to the model.

```
rtwdemo_configinterface
set_param('rtwdemo_configinterface','EmbeddedCoderDictionary','codeDefinitions.sldd');
```



The model contains four inports. `In1` is configured to use the storage class `ImportFromFile`. For this example, you configure the other inports to read data from external code by default. The model also contains state data for the Delay block that you configure to be accessible by external code. When you apply the storage classes `ImportFromHeader` and `ExportToPrivateHeader` to the model elements, the generated code conforms to this architecture.

To configure the model with the code generation definitions, get the code mapping for the model.

```
cm = coder.mapping.api.get('rtwdemo_configinterface');
```

Use the code mapping to configure data to use your storage class definitions. For the state data in the Delay block, specify the storage class `ExportToPrivateHeader`.

```
setState(cm,'rtwdemo_configinterface/Delay','StorageClass','ExportToPrivateHeader');
```

Specify the default storage class `ImportFromHeader` for root inport data. For inports of the model, by default, the generated code uses data that external code defines in the file `rtwdemo_configinterface_input.c`. To compile and link this external file when you build the generated code, set the configuration parameter `CustomSource` to `rtwdemo_configinterface_input.c`.

```
setDataDefault(cm,'Inports','StorageClass','ImportFromHeader');
set_param('rtwdemo_configinterface','CustomSource','rtwdemo_configinterface_input.c');
```

Specify the function customization template `GlobalFunctions` for the default code generation of execution functions.

```
setFunctionDefault(cm,'Execution','FunctionCustomizationTemplate','GlobalFunctions');
```

Verify that the code definitions reflect your specifications.

1. Open the Code Mappings editor. In the Simulink Editor, open the Embedded Coder app. On the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
2. In the Code Mappings editor, on the **Data Defaults** tab, expand the **Inports and Outports** section. The **Inports** category shows the storage class `ImportFromHeader`. On the **Inports** tab, the inports `In2`, `In3`, and `In4` use the model default storage class `ImportFromHeader`. The inport `In1` has a different storage class specified, `ImportFromFile`. This setting overrides the default storage class so that `In1` reads data from a different external file.
3. On the **Signals/States** tab, the state X shows the storage class `ExportToPrivateHeader`.
4. On the **Function Defaults** tab, the **Execution** category shows the template `GlobalFunctions`.
5. Open the Embedded Coder Dictionary. On the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**. The code definitions that you added to the dictionary appear in the tabs.

Generate and Verify the Code

Generate code for the model.

```
slbuild('rtwdemo_configinterface', 'generateCodeOnly', true);
```

```
### Starting build procedure for: rtwdemo_configinterface
### Successful completion of code generation for: rtwdemo_configinterface

Build Summary

Top model targets built:

Model                     Action           Rebuild Reason
========================================================================================
rtwdemo_configinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 47.194s
```

The header file `rtwdemo_configinterface_private.h` defines the Delay block state data according to the storage class `ExportToPrivateHeader`. External code can access this state data.

```
file = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface_private.h');
rtwdemodbtype(file,'/* Storage class', ...
   '#endif');
```

```
/* Storage class 'ExportToPrivateHeader' */

/*Fast onchip RAM*/

#pragma begin FAST

extern MYTYPE X;                        /* '<Root>/Delay' */

#pragma end FAST
```

Because the inports use the storage class specifications, the generated header files do not define inport data. Instead, the generated file `rtwdemo_configinterface.h` includes the external header file that you specified in the storage class definition. The generated code reads inport data from the variables defined in the external file `rtwdemo_configinterface_inputs.c`.

In the generated source file `rtwdemo_configinterface.c`, the execution (step) function reflects the settings of the function customization template `GlobalFunctions`. Because the template

`GlobalFunctions` uses the memory section `FastMem`, the execution function is stored in the memory section and reflects the prestatement, poststatement, and comments that you set in the `FastMem` definition.

```
file2 = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface.c');
rtwdemodbtype(file2,'/* Model step function */', ...
   '/* Logic:');
```

```
/* Model step function */

/*Fast onchip RAM*/
#pragma begin FAST

void rtwdemo_configinterface_step(void)
{
```

After you configure the Embedded Coder Dictionary, you can share the dictionary with your team or organization. You and other users can apply your definitions to multiple models so that the models use the same software architecture.

See Also

Embedded Coder Dictionary | **Code Mappings Editor – C** | `coder.Dictionary` | `coder.mapping.api.CodeMapping` | `coder.dictionary.Section` | `coder.dictionary.Entry`

Related Examples

- “Deploy Code Generation Definitions” on page 32-16
- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24
- “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

Create a Service Interface Configuration

In this section...

"Identify Software Platform Requirements" on page 32-11

"Create a Service Interface Configuration in a Standalone Embedded Coder Dictionary" on page 32-12

"Create Code Definitions for the Service Interface Configuration" on page 32-13

"Test the Service Interface Configuration" on page 32-14

"Deploy Code Interface Configuration Definition to Models" on page 32-14

When you generate component code for deployment in a target execution environment, you can configure the generated code to use the services that the target execution environment provides. To describe the services to the code generator, you create a *service interface configuration* in an Embedded Coder Dictionary. To generate code for the target environment, you share the dictionary among your models and apply the service interface configuration to the models. Separating your models from the service interface configuration in this manner enables you to:

- Develop large and complex embedded systems using a scalable approach.
- Easily deploy the same service interface configuration to many models simultaneously.
- Quickly change the service interface configuration for a system without editing the individual models.

After you create a service interface configuration, you share it with the models from which you want to generate code for the target execution environment. By default, the generated code interacts with the platform middleware, which is the set of services provided by the target execution environment. You can further configure individual models to suit your code generation objectives by mapping model elements to specific interface definitions from the configuration..

Identify Software Platform Requirements

This table lists the code definitions that you can create for a service interface configuration. Consider how your code interacts with the platform middleware and identify the code definitions that you need to create to specify that behavior.

Code Definition	Purpose
Execution function customization template	Controls the naming of model entry-point functions and optionally applies memory sections. Execution function templates for a service interface configuration include: • Initialize and terminate functions • Periodic and aperiodic functions including model step functions

Code Definition	Purpose
Service interface	Defines properties such as the name and data communication method for model elements that use platform services. Service interfaces include: • Receiver service interfaces • Sender service interfaces • Data transfer service interfaces • Timer service interfaces • Parameter tuning service interfaces • Parameter argument tuning service interfaces • Measurement service interfaces
Internal function customization template	Controls the naming of functions that the component calls and optionally applies memory sections. Internal function templates for a service interface configuration include: • Subcomponent functions • Shared utility functions
Memory	Define the appearance, placement, and storage of the data and functions in generated code. • Storage class — You apply a storage class to receiver, sender, parameter, and measurement interfaces. • Memory section — Controls the placement of data and function definitions in memory by using pragmas and other code decorations. You apply a memory section to storage classes and function templates.

Create a Service Interface Configuration in a Standalone Embedded Coder Dictionary

To create a service interface configuration, you must create a standalone Embedded Coder Dictionary that contains the configuration. Storing the service interface configuration in a standalone Embedded Coder Dictionary enables you to share the configuration with multiple models.

1. Open the Model Explorer.
2. Click **File** > **New** > **Embedded Coder Dictionary**.
3. Specify a name for the Embedded Coder Dictionary. For this example, name the dictionary `InterfaceCoderDictionary`. The dictionary is created in the current folder. You can optionally select the location to create the dictionary by clicking **Browse**. Click **Next**.
4. Select the **Service Interface** configuration option. Click **Create**.

The Embedded Coder Dictionary opens and shows the service interface configuration.

Create Code Definitions for the Service Interface Configuration

Define the interfaces of the generated code by creating code definitions in the service interface configuration. For this example, create a function customization template and a sender service interface definition.

Define Prototypes for Callable Functions

To control the naming of functions in the generated code, define function customization templates. For this example, define a function template named `StepFunctions` for model step functions and define the naming rule for the functions.

1. In the Embedded Coder Dictionary, on the left pane, click **Execution**.
2. For the **Periodic and Aperiodic Functions** list, click **Create**.
3. On the right pane, set these properties:
   - **Name** — `StepFunctions`
   - **Function Naming Rule** — `$R$N_func`

   Confirm the appearance of the generated function declaration and definition in the **Pseudocode Preview** pane.
4. In the table, select the new function template as the dictionary default.

For models that use the service interface configuration, the code generator creates the periodic and aperiodic functions by using the template that you created by default. The other function templates are available for use in the Code Mappings editor.

Define Services for a Service Interface Configuration

Specify how the generated code calls the services of the platform middleware by creating service interface definitions. You can define these types of service interfaces:

- Receiver
- Sender
- Data transfer
- Timer
- Parameter tuning
- Parameter argument tuning
- Measurement

For this example, define a sender service interface that communicates outside of function execution.

1. On the left pane, click **Service Interfaces**.
2. In the **Sender** section, click **Create**.
3. On the right pane, set these properties:
   - **Name** — `MySender`
   - **Data Communication Method** — `Outside Execution`
   - **Function Naming Rule** — `$R_set_$X$N`

Confirm the appearance of the generated sender declaration and definition in the **Pseudocode Preview** pane.

4. In the table, select the new sender service interface as the dictionary default.

Configure Default Settings

For each category of definitions in a service interface configuration, there is one default definition that the generated code uses when you apply the configuration to a model. The full list of definitions for each category appears in the code mapping for the models that use the configuration. For an individual model, you can override the default definition by applying a different definition to the category or to an individual model element.

To specify the dictionary default, navigate to the interface category in the dictionary. On the row for the interface that you want to select, click the circle in the **Dictionary Default** column.

Test the Service Interface Configuration

To see how the data and functions appear in the generated code, apply the service interface configuration to the model and generate code.

1. Open the model from which you want to generate code using the service interface configuration.
2. Open the configuration parameters. on the **Modeling** tab, click **Settings**.
3. Set the parameter **Shared coder dictionary** to the name of the data dictionary that contains the service interface configuration. For this example, use the dictionary `InterfaceCoderDictionary.sldd`. Click **OK**.

Click **Generate Code**. Confirm that the generated code shows the service interfaces that you defined in the dictionary.

Deploy Code Interface Configuration Definition to Models

To configure the code interface for a large system with many models, share the code interface configuration with the models and configure them to use the platform.

1. Attach the shared coder dictionary that contains the interface configuration to the models, as shown in the previous section.
2. Configure the deployment type of each model. See “Deploy Code Generation Definitions” on page 32-16.
3. Apply model-specific configurations by using the Code Mappings table in each model. See “C Code Generation Configuration for Model Interface Elements” on page 33-2.

For more information about sharing the code interface configuration among your models, see “Deploy Code Generation Definitions” on page 32-16.

See Also

Embedded Coder Dictionary | **Code Mappings Editor – C**

Related Examples

- “Deploy Code Generation Definitions” on page 32-16

- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24

Deploy Code Generation Definitions

As described in “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2, you can create code generation definitions that you and other users can apply to model elements such as data and functions. The code definitions control the appearance of those elements in the generated code.

In a large organization with multiple models and users, you can share code generation definitions by storing them in a mutually accessible location. The process of sharing code definitions is called deployment.

Share Code Generation Definitions Between Multiple Models and Users

When you create custom code generation definitions, to share the definitions between models and other users:

- When you create definitions in an Embedded Coder Dictionary, save the definitions in a SLDD file. Then, attach the SLDD file that contains the code generation definitions to each model. For an example, see “Share Code Interface Configuration Between Models” on page 32-19.
- When you create definitions in a package by using the Custom Storage Class Designer, for each user, place the folder containing the package folder on the MATLAB path. For more information, see “What Is the MATLAB Search Path?”.

Dictionary Usage for Models Created with Different Versions of Simulink

Simulink provides version handling for Embedded Coder Dictionaries. When these events occur, Simulink synchronizes data in a data dictionary, including the Embedded Coder Dictionaries, for use with a model regardless of the Simulink version used to create the model.

- You attach a data dictionary that was saved in a previous version of Simulink to a model (for example, you attach a dictionary saved in R2018a to a model that you develop in R2018b).
- You open a model, which has a local Embedded Coder Dictionary, in a version of Simulink that is older than the current version (for example, you developed a model that uses a local dictionary in R2018a and you open that model in R2018b to continue development).

You also have the option to export (save) an Embedded Coder Dictionary for use with models created with a different version (previous or older) of the code generator.

For more information, see “Dictionary Usage for Models Created with Different Versions of Simulink”.

Configure Default Code Mapping in a Shared Dictionary

When you set up a shared Embedded Coder Dictionary, you can map default code definitions for categories of model data and functions in the dictionary. The dictionary default definitions apply in the mapping by default for the models that use the dictionary. In this example, you create the Embedded Coder Dictionary `exSharedCodeDefs.sldd` and map default definitions in the dictionary. If you have not created an Embedded Coder Dictionary, create one.

1. In the Model Explorer, click **File** > **New** > **Embedded Coder Dictionary**.
2. Specify the dictionary file name `exSharedCodeDefs.sldd`.
3. Choose to create either a data interface or service interface configuration. For this example, create a *service interface* configuration. Click **Create**.
4. Create code interface definitions in the dictionary. For this example, create a receiver service definition with these settings:
   - **Name** — `SharedReceiverService`
   - **Data Communication Method** — Outside Execution
   - **Function Naming Rule** — `get_data_$X$N`

The type of code interface configuration that you choose determines what categories are available for you to set the default definitions.

Set Dictionary Defaults for a Service Interface Configuration

For a service interface configuration, you can select dictionary defaults for the function categories and service interface categories.

1. In the Embedded Coder Dictionary dialog box, on the left pane, click the category of service interface that you want to configure. On the row for the service interface that you want to select, click the circle in the **Dictionary Default** column. You can select default service interface definitions for these interface categories.

Service Interface Category	Description
Receiver	Calls to the target platform receiver service, represented as root-level inports in the model
Sender	Calls to the target platform sender service, represented as root-level outports in the model
Data transfer	Calls to the target platform data transfer service, represented as signal lines between callable functions within a component model
Timer	Calls to the target platform clock tick, represented as Discrete-time integrator, Weighted sample time, and Digital clock blocks in aperiodic export-function models
Parameter tuning	Representation of parameters that are accessible in memory for tuning
Parameter argument tuning	Representation of model parameter arguments that are accessible in memory for tuning
Measurement	Representation of signals, states, and data stores so that data is preserved in generated code for measurement

   For this example, set the service interface for receivers to `SharedReceiverService`, which you created in the previous section.
2. On the left pane, click the category of function template that you want to configure. On the row for the function template that you want to select, click the circle in the **Dictionary Default** column. You can select default function templates for these function categories.

Model Function Category	Description
Initialize and Terminate	Entry-point functions generated for Initialize function and Terminate function blocks
Periodic and Aperiodic	Entry-point functions generated for Function-call subsystem blocks, function-call models, Simulink function blocks, and S-function blocks
Subcomponent Functions	Function calls between subcomponents within a component model
Shared Utility	Shared utility functions

3 Apply your changes by clicking **Save** and close the Embedded Coder Dictionary and Model Explorer.

Set Dictionary Defaults for a Data Interface Configuration

For a data interface configuration, you can select dictionary defaults for categories of model data elements and model functions.

1 In the Embedded Coder Dictionary dialog box, on the left pane, click **Data Defaults**.

2 In the **Data Defaults** table, configure default code definitions for categories of data and functions.

Model Element Category	Description
Inports	Root-level input ports of a model, such as Inport and In Bus Element blocks.
Outports	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.
Shared local data stores	Data Store Memory blocks that have the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model.
Global data stores	Data stores that are defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. To view and configure these data stores in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Model parameter arguments	Parameters in the model workspace that you configure as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the **Model Data Editor** > **Parameters** > **Argument** check box.

Model Element Category	Description
Model parameters	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
External parameters	Parameters that you define as objects in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. To view and configure these parameters in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Constants	Constant-value block output and parameters that could not be inlined. These values are stored in variables for one of the following reasons. • The value is an array larger than the loop unrolling threshold. • The value address is needed in the code.

The default mapping for a category applies to elements in a category throughout a model.

For this example, set the storage class for data categories **Inports**, **Outports**, and **Global data stores** to `ExportedGlobal`.

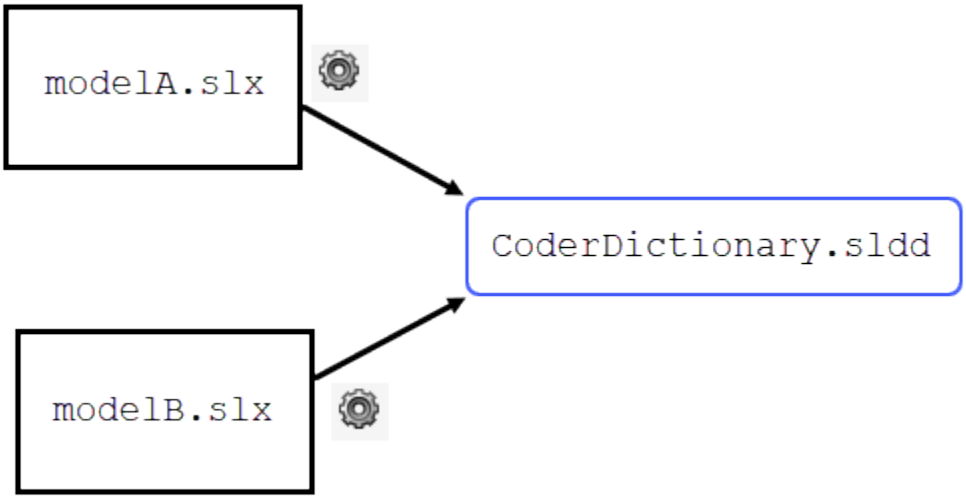
3 In the Embedded Coder Dictionary dialog box, on the left pane, click **Function Defaults**. Configure the function customization template and, if applicable, a memory section, for these categories of functions:

Model Function Category	Description
InitializeTerminate	Entry-point functions for initialization and termination
Execution	Entry-point functions for initiating execution and resets
Shared utility	Shared utility functions

4 Apply your changes by clicking **Save** and close the Embedded Coder Dictionary and Model Explorer.

Share Code Interface Configuration Between Models

You can generate standardized code for the same target execution environment from different models by sharing the same code interface configuration with the models through a shared Embedded Coder Dictionary. Attach the `.sldd` file that contains the Embedded Coder Dictionary to the models by setting the model configuration parameter **Shared coder dictionary**.



For this example, configure the models `RollAxisAutopilot` and `rtwdemo_configinterface` to use the shared coder dictionary that you created in “Configure Default Code Mapping in a Shared Dictionary” on page 32-16.

1. Open the models.

   ```
   rtwdemo_configinterface
   openExample('RollAxisAutopilot');
   ```
2. In `RollAxisAutopilot`, open the **Embedded Coder** app.
3. On the **C Code** tab, click **Code Interface** > **Set up shared Embedded Coder Dictionary**.
4. In the dialog box, you can either open an existing Embedded Coder Dictionary or create a new one. For this example, select **Open Embedded Coder Dictionary** and specify the dictionary file `exSharedCodeDefs.sldd`. Click **Next**.

   The model now uses the shared coder dictionary.
5. Repeat this process for the model `rtwdemo_configinterface`.

Now, for each model, the configuration parameter **Shared coder dictionary** is set to `exSharedCodeDefs.sldd`. The models use the default definitions from the shared dictionary, and the definitions from the dictionary are available in the Code Mappings panel for each model. You can alternatively configure multiple models to use the same dictionary by sharing the configuration set that has the parameter **Shared coder dictionary** set to the name of the dictionary file.

Apply Code Generation Definition in Models

Confirm that the shared definitions are available for use in the models by reviewing the code mapping settings and inspecting the generated code.

1. In `RollAxisAutopilot`, open the Code Mappings panel. on the **C Code** tab, click **Code Interface** > **Component Interface**.
2. On the **Inports** tab, confirm that the receiver services are set to `Dictionary default: SharedReceiverService`. The other receiver service definitions from the dictionary are available for selection in the **Receiver Service** lists.
3. Generate code from `RollAxisAutopilot`.

4. In the **Code** view, inspect the generated file `RollAxisAutopilot.c`. The code calls receiver services according to the default definition that you specified. For example, the code for the Switch block calls the receiver service `get_data_RollAxisAutopilot_step_AP_Eng`.

```
/* Switch: '<S4>/Enable' incorporates:
 *  Inport: '<Root>/AP_Eng'
 *  Logic: '<S3>/NotEngaged'
 */
if (!get_data_RollAxisAutopilot_step_AP_Eng()) {
  /* Switch: '<S3>/RefSwitch' incorporates:
   *  Constant: '<S3>/LoThr'
   *  Constant: '<S3>/UpThr'
   *  Constant: '<S3>/Zero'
   *  Inport: '<Root>/Phi'
   *  Logic: '<S3>/Or'
   *  RelationalOperator: '<S3>/RefThreshold1'
   *  RelationalOperator: '<S3>/RefThreshold2'
   *  UnitDelay: '<S4>/FixPt Unit Delay1'
   */
  if ((get_data_RollAxisAutopilot_step_Phi() >= 6.0F) ||
      (get_data_RollAxisAutopilot_step_Phi() <= -6.0F)) {
    rtDW.FixPtUnitDelay1_DSTATE = get_data_RollAxisAutopilot_step_Phi();
  } else {
    rtDW.FixPtUnitDelay1_DSTATE = 0.0F;
  }

  /* End of Switch: '<S3>/RefSwitch' */
}
```

5. Optionally, generate code from `rtwdemo_configinterface` and confirm that the code appears as expected.

Migrate Data Interface Configuration from Model File to Shared Data Dictionary

If you create code generation definitions in the Embedded Coder Dictionary of a model, the definitions are stored in the model file. If you want to share the definitions between models:

1. Create a shared data dictionary as shown in “Share Code Interface Configuration Between Models” on page 32-19. Before you proceed to the next step, make sure all of the target models have access to the shared dictionary.
2. Use the `coder.dictionary.move` function to migrate the definitions from the source model file to the shared data dictionary.
3. Optionally, copy Code Mappings editor settings from the source model to other models by using the programmatic interface of the Code Mappings editor (see “Configure Default Data and Function Code Generation Programmatically” on page 33-24).

In this example, you convert a subsystem to a referenced model. Then, you create a storage class in the Embedded Coder Dictionary of the top model and use the storage class in the Code Mappings editor. To use the same Code Mappings editor settings for the top model and for the referenced model, you must extract the storage class into a shared data dictionary (`.sldd`).

Create Referenced Model

1. Open the example model `RollAxisAutopilot`.

```
openExample('RollAxisAutopilot');
```

2. In the model, right-click the `BasicRollMode` subsystem and select **Subsystem & Model Reference** > **Convert to** > **Referenced Model**.
3. Click **Convert**.

   The new referenced model, `BasicRollMode`, opens. The model file is in your current folder.

Create and Apply Storage Class

1. In the top model `RollAxisAutopilot`, click the **C Code** tab. If the toolstrip does not show the **C Code** tab, open the Embedded Coder app.
2. Click **Code Interface** > **Embedded Coder Dictionary**.
3. In the Embedded Coder Dictionary dialog box, on the **Storage Class** table, click **Create**.
4. For the new storage class, set these property values:
   - **Name** to `internalsInFile`.
   - **Header File** to `$R_internal.h`.
   - **Definition File** to `$R_internal.c`.
5. In the model, in the Code Mappings editor, for the **Data Defaults** > **Signals, states, and internal data** row, set **Storage Class** to `internalsInFile`.

The referenced model cannot access the code generation definitions in the Embedded Coder Dictionary of the parent model. Now, move the storage class out of the parent model and into a shared data dictionary.

Move Storage Class to Shared Data Dictionary

To create the shared, standalone dictionary and store the code generation definition in it:

1. In the Model Explorer, select **File** > **New** > **Embedded Coder Dictionary**.
2. Create a coder dictionary named `coderDefs.sldd`. For this example, select the option to create a *data interface* configuration.
3. Configure `RollAxisAutopilot` to refer to `coderDefs.sldd`. For the model, open the model configuration parameters and set **Shared coder dictionary** to `coderDefs.sldd`.
4. Set the configuration parameter **Shared coder dictionary** for the model `BasicRollMode` to `coderDefs.sldd`.
5. At the command prompt, move the code generation definitions from the `RollAxisAutopilot` model file to the `coderDefs.sldd` data dictionary.

   ```
   coder.dictionary.move('RollAxisAutopilot','coderDefs.sldd');
   ```

Now, both models have access to the storage class `internalsInFile`.

Copy Code Mappings Editor Settings from Parent Model to Referenced Model

The new referenced model does not use the Code Mappings editor settings that the parent model uses. To copy the settings from the parent model to the referenced model, use the programmatic interface of the Code Mappings editor.

At the command prompt, copy the storage class setting for **Signals, states, and internal data** from `RollAxisAutopilot` to `BasicRollMode`.

```
roll_mapping = coder.mapping.api.get('RollAxisAutopilot');
roll_sc = getDataDefault(roll_mapping,'InternalData','StorageClass');
```

```
mode_mapping = coder.mapping.api.get('BasicRollMode');
setDataDefault(mode_mapping,'InternalData','StorageClass',roll_sc);
```

Now, both models use the same Code Mappings editor settings. For more information about the programmatic interface of the Code Mappings editor, see “Configure Default Data and Function Code Generation Programmatically” on page 33-24.

See Also

Embedded Coder Dictionary

Related Examples

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24
- “Partition Data for Model Reference Hierarchy Using Data Dictionaries”
- “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14

Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models

In a project where you generate code from many models, you can configure the models with the same default code settings so that they generate code that conforms to your software architecture standards. These settings include:

- Configuration parameter settings
- Code interface definitions such as storage classes
- Code mappings that specify code definitions for categories of model data and functions

As projects expand and you and your colleagues create more models, manually configuring the default settings for each new model takes time. To reduce this manual data entry, you can use tools to configure many models by default. You can copy settings between models, enabling each model to later change the settings independently of each other. Alternatively, you can share settings, enabling you to change the settings for multiple models simultaneously with minimal effort.

Share Common Code Generation Settings among Multiple Models

One way to make models conform to code generation standards is by storing the settings in one location and sharing them among the models. Share settings that you might want to change so that when you do change the settings, the models are updated. To share code generation settings among multiple models, store the settings in a standalone Simulink data dictionary. Then, attach one or more additional, intermediate dictionaries, to each model. Configure the intermediate dictionaries to refer to the standalone dictionary. With this modular dictionary hierarchy, each model can store other data, such as design data, separately from the shared settings. For an example that shows how to store shared data in a dictionary, see “Partition Data for Model Reference Hierarchy Using Data Dictionaries”. Once you set up the data dictionary hierarchy for your models, choose which code generation settings that you want to share by using the dictionary.

Model Configuration Parameter Settings

Configuration parameters control model-wide settings including code generation options such as the system target file, target language, and custom code files. A model's configuration parameters are stored in the configuration set. For code generation parameters on the **Code Generation** pane and links to the pages for the code generation sub-panes, see “Model Configuration Parameters: Code Generation”. Some systems that include multiple models require that many of the configuration parameter settings be consistent across the models. For example, when you generate code for a model reference hierarchy, the models in the hierarchy must have the same settings for many of the code generation parameters. In this type of system, share one configuration set with the models.

To share configuration settings among multiple models, store the settings in a standalone configuration set object. Save the configuration object in a Simulink data dictionary that the models share. For each model, activate a configuration reference that points to the shared configuration set. For an example of how to share a configuration set among multiple models, see “Share a Configuration with Multiple Models”. When you want to change parameter settings, edit the shared configuration set. The models that reference it use the new settings. To quickly share a configuration set among referenced models in a hierarchy, see “Share a Configuration Across Referenced Models”.

Code Interface Definitions

For the code interface of many models to conform to the same specifications, define the interface in a shared code interface configuration in an Embedded Coder Dictionary. Use code definitions in the

configuration to specify many detailed code properties. Then you can control the code interface of your data items and functions by applying the code definitions using the code mapping. Code interface definitions include:

- Storage classes
- Memory sections
- Function customization templates
- Service interfaces

Once you create your code interface definitions, apply them to a model by mapping model data and functions to the definitions. Sharing one set of code definitions among multiple models enables you to quickly apply the same set of interface specifications by mapping the data and functions in each model to the definitions.

To share a set of code definitions among multiple models, create the definitions in a shared Embedded Coder Dictionary, which is save in an SLDD file. Configure the models to use the dictionary by setting the model configuration parameter **Shared coder dictionary**. Then, map the data and functions in the model to the shared definitions. For an example of how to share definitions among multiple models, see “Share Code Interface Configuration Between Models” on page 32-19.

Code Mappings

Models can also share the default mapping configuration that applies the definitions to data and function categories. In the shared Embedded Coder Dictionary, you configure the dictionary default mapping. When you attach the dictionary to a model, the model uses the dictionary mapping by default. You can still change the default mapping and individual mapping for an individual model if the model requires a different configuration. For more information about sharing the dictionary default mapping among multiple models, see “Configure Default Code Mapping in a Shared Dictionary” on page 32-16.

Copy Code Generation Settings to New Models

If you do not want to share code generation settings from one source, you can use other tools to configure models by copying the code generation settings. When you use these techniques, you can easily change the settings for an individual model after the initial configuration. If you want to change a setting for many models that are already configured, you must change each model individually. In the model development process, consider when you want to apply the settings to your models and use one of these techniques.

Template Models

If you want new models to be configured at the time they are created, use a template model. You can create a template model that has preconfigured settings such as configuration parameter settings and Code Mappings editor settings. To create a model, you and others can instantiate the template. Each new model can later change their settings independently of other models. See “Create Template from Model”.

When you create a model by instantiating a template, no data dictionary is attached to the new model. If the preconfigured settings in the template model depend on shared data that you store in a data dictionary, you must attach the dictionary that contains the shared data to each new model. Alternatively, you can attach a different dictionary to the model, and configure this dictionary to reference the dictionary with the shared data.

Template Projects

If you want new models to use data that you store outside of the models, such as in a data dictionary, use template projects. You can create a template project that contains models, data dictionaries, and other artifacts. You can also write project shortcuts that execute custom code to create models, copy settings, and make shared settings stored in a dictionary available to new models. For more information about using template projects, see “Create Templates for Standard Project Settings”.

Programmatic Configurations

If you want to copy settings to a model after you create it, you can write scripts, callbacks, and shortcuts that use the programmatic interfaces to configure code generation settings. Configure these code generation settings by using the corresponding programmatic interfaces:

- For configuration parameter settings, use the `set_param` and `get_param` functions. You can write a script that sets many parameters, as described in “Automate Model Configuration by Using a Script”.
- For code interface definitions in an Embedded Coder Dictionary, see the example “Create Data Interface Configuration Programmatically” on page 32-5.
- For Code Mappings editor settings, see “Configure Default Data and Function Code Generation Programmatically” on page 33-24.

Choose how to execute the code by considering when you want to configure the models. You can execute this code in:

- A script. You can configure the models at any time by running the script manually.
- A model callback. A callback uses your code to configure the model each time the callback is called. For example, you can use the `PreLoadFcn` callback, which executes whenever you open a model. See “Model Callbacks”.
- A project shortcut. The shortcut can create a model and copy settings into the model. See “What Can You Do with Project Shortcuts?”.

After you use the code to configure the models, you can change the settings on each individual model as needed.

See Also

Related Examples

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Deploy Code Generation Definitions” on page 32-16
- “Partition Data for Model Reference Hierarchy Using Data Dictionaries”

Manage an Embedded Coder Dictionary

After you create code definitions for standardization and software architecture conformance and deploy them to your users, keep the definitions and their usage up to date. When you and other users create models for code generation, make your code definitions available to the models. If the standards for your generated code change, you might need to update your code definitions. Consider how these standards changes impact models that use your code definitions and update the models accordingly.

Share Available Code Definitions with New Models

If you share code definitions in an Embedded Coder Dictionary, make the definitions available to new models that you or your users create for code generation. A model that you create from the Simulink Start Page or by using the `new_system` function does not have a shared coder dictionary attached to it by default, and the new model cannot access code generation definitions.

To make code definitions available to a new model automatically, write callbacks and other code that immediately configures the model to use the shared coder dictionary that contains the definitions.

- For a simple example that shows how to attach a shared coder dictionary to a model programmatically and for information about configuring Embedded Coder Dictionaries programmatically, see “Create Data Interface Configuration Programmatically” on page 32-5.
- If you use a project, consider creating a project shortcut that creates a model and immediately configures the new model to use the shared coder dictionary. For information about projects and project shortcuts, see “What Are Projects?”.
- Consider using a shared configuration set for the models. Define the configuration set you want to share inside an Embedded Coder Dictionary. When you create a new model, open the Model Configuration Parameters dialog box and click on **Code Generation**. Then, set the parameter **Shared coder dictionary** to the name of the Embedded Coder Dictionary. For more information, see “Share a Configuration with Multiple Models”.

To manually share the definitions with a new model, see “Share Code Interface Configuration Between Models” on page 32-19.

Migrate Package Definitions to Embedded Coder Dictionary to Use in Code Mappings

If you create a storage class in a package and you want to use it in the Code Mappings editor, refer to the package from one or more Embedded Coder Dictionaries. To refer to a package from an Embedded Coder Dictionary, see “Refer to Code Generation Definitions in a Package”. Then, for models that have access to the Embedded Coder Dictionary, you can apply storage classes from the package by using the Code Mappings editor.

When you want to make changes to the definition, make changes only in the package. To apply the changes to the Embedded Coder Dictionaries, refresh the package. To refresh the package, open each Embedded Coder Dictionary.

To apply a storage class to a model data element that is not resolved to an external data object, use the Code Mappings editor. To make the package definitions appear in the Code Mappings editor, refer to the package from the Embedded Coder Dictionary that the model uses.

These unresolved model elements include:

- Inport blocks
- Outport blocks
- Signal lines
- Block states
- Data stores
- Parameter objects in the model workspace

To map these model elements to a package-based definition, refer to the package from the Embedded Coder Dictionary that the model uses.

Beginning in R2020b, if an existing model includes these model elements and they are configured with package-based definitions, Simulink references the package from the Embedded Coder Dictionary. If a shared coder dictionary is attached to the model, Simulink uses the shared coder dictionary to reference the package.

Change Code Generation Definitions

If the requirements for your generated code change, you might need to update your code definitions to reflect the latest interface standards. Changing or deleting a code definition can affect models that use the code definition. When you make changes to code definitions, assess the impact of the changes. Then, deploy the changes and fix the affected usage points.

To make changes to definitions in a package, use the Custom Storage Class designer. For more information, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.

To make changes to definitions in an Embedded Coder Dictionary:

- Use the Embedded Coder Dictionary tool, as described in **Embedded Coder Dictionary**.
- Use a script, as described in “Create Data Interface Configuration Programmatically” on page 32-5. You can write a script to automatically make a change to many code definitions in an Embedded Coder Dictionary.

If you store the Embedded Coder Dictionary in a shared SLDD file, save the file after you update the code definitions. If you store the Embedded Coder Dictionary in a model, save the model after you update the code definitions.

Assess the Impact of Code Definition Changes

Depending on the storage location of the code definition, changing or deleting the definition can break usage points. This table shows the impact of different changes based on the location of the definition. Check if usage points are affected by the change and fix them by making the corresponding updates.

Definition Changes	Storage Location	Updates
Change the name of a definition	Package	For a memory section, update other definitions in the package that use the memory section. In the Custom Storage Class designer, specify the new memory section name.
		For a storage class, update data objects that use the storage class. Specify the new storage class name for the data object.
		In Embedded Coder Dictionaries that reference the package, refresh the package.
		In models that use the definition in the Code Mappings, specify the new definition name. Use either the Code Mappings editor or a script that uses the `coder.mapping.api.CodeMapping` object.
	Embedded Coder Dictionary	None
Change properties of a definition	Package	In Embedded Coder Dictionaries that reference the package, refresh the package.
	Embedded Coder Dictionary	None
Delete a definition	Package	For a memory section, update other definitions in the package that use the memory section. In the Custom Storage Class designer, specify a different memory section or no memory section.
		For a storage class, update data objects that use the storage class. Specify a different storage class or no storage class for the data object.
		In Embedded Coder Dictionaries that reference the package, refresh the package.
		In models that use the definition in the Code Mappings, specify a different storage class or no storage class. Use either the Code Mappings editor or a script.

Definition Changes	Storage Location	Updates
	Embedded Coder Dictionary	For a memory section, update other definitions in the dictionary that use the memory section. Specify a different memory section or no memory section. Use either the Embedded Coder Dictionary or a script.
		In models that use the definition in the Code Mappings, specify a different storage class or no storage class. Use either the Code Mappings editor or a script.
Add a definition	Package	In Embedded Coder Dictionaries that reference the package, refresh the package.
	Embedded Coder Dictionary	None

See Also

Embedded Coder Dictionary

More About

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Deploy Code Generation Definitions” on page 32-16

Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings

For a model created in a release before R2018a, when you open the Embedded Coder app or use the `coder.mapping.create` function, Simulink migrates memory section and shared utility settings from the configuration parameters to the code mappings of the model. As part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses:

- To refer to the package that defines the memory sections, as described in “Refer to Code Generation Definitions in a Package”.
- To contain a function customization template that specifies the naming rule and, if applicable, memory section settings that you specified for shared utilities.

Effects of Migration

- The migration process makes changes to the model file, which you must save. The changes include:
  - Translation of the configuration parameter settings into code mappings.
  - If no data dictionary is attached to the model, configuration of the internal Embedded Coder Dictionary in the model file to refer to the memory section package and to contain a function customization template.
- If a data dictionary is attached to the model, the migration process makes changes to the dictionary, which you must save. The changes include:
  - If the configuration parameters are stored in the model file (the model does not refer to a `Simulink.ConfigSet` object), configuring the Embedded Coder Dictionary of the data dictionary to refer to the memory section package and to contain a function customization template.

    If the data dictionary references other dictionaries, the migration process configures the dictionary at the bottom of the reference hierarchy so other models and dictionaries in the hierarchy can use the memory section and function template definitions.
  - If the model refers to a `Simulink.ConfigSet` object, configuring the Embedded Coder Dictionary of the data dictionary that stores the object.

Considerations Before Migrating

- When you open the Embedded Coder app, Simulink executes the migration process. Do not open the Embedded Coder app unless you are ready to migrate memory section and shared utility naming settings.

  If a model has multiple configuration sets (one active and one or more inactive) or refers to a `Simulink.ConfigSet` object, the Embedded Coder app displays an informational message indicating that information from the referenced configuration set has been imported into the Embedded Coder Dictionary and code mappings.
- If you use Simulink data dictionaries, before you or other users open the Embedded Coder app for a model, consider creating and configuring the dictionaries first.

  In a hierarchy of referenced data dictionaries (for example, a dictionary hierarchy that parallels a model hierarchy), only one dictionary can contain an Embedded Coder Dictionary. Setting up data

dictionaries before migrating associated models enables you to control where the Embedded Coder Dictionary resides in the dictionary hierarchy. Create or identify a data dictionary at the bottom of the hierarchy so that models in the hierarchy can access the memory section and function template definitions. Then, when you initiate migration by opening Embedded Coder app for a model, Simulink configures that data dictionary.

For information about sharing an Embedded Coder Dictionary between models by using referenced dictionaries, see “Share Code Interface Configuration Between Models” on page 32-19.

- The migration process makes changes based on the active configuration set of the model (see “Manage Configuration Sets for a Model”) when you open the model in the Embedded Coder app. Before you open the model in the Embedded Coder app, activate the configuration set whose settings you want the migration process to use.

  The process ignores settings in inactive configuration sets. If different configuration sets of the model specify different settings, you must choose one set to migrate.

Considerations After Migrating

- In a branching hierarchy of referenced models and referenced data dictionaries, after you or other users migrate referenced models in different branches, opening the Embedded Coder app in a parent model can generate an error due to the presence of multiple Embedded Coder Dictionaries in the data dictionary hierarchy. To resolve these errors, use `coder.dictionary.move` and `coder.dictionary.remove` to transfer and delete Embedded Coder Dictionaries until only one dictionary remains. Place the remaining Embedded Coder Dictionary in a data dictionary at the bottom of the hierarchy so that the models in the hierarchy can access the memory section and function template definitions.

  For information about sharing an Embedded Coder Dictionary between models by using referenced dictionaries, see “Share Code Interface Configuration Between Models” on page 32-19.
- After you migrate a model, memory section and shared utility model configuration parameters are not accessible. If you use `set_param` or `get_param` to access these configuration parameter settings programmatically, the functions generate errors. To change these settings, use the Code Mappings editor or the programmatic interface as described in “C Data and Function Interfaces”.

  If a model contains atomic subsystems that have nondefault memory section settings, you can select the memory sections from built-in packages that are loaded into the Embedded Coder Dictionary. See “Override Default Memory Placement for Individual Data Elements” on page 45-6 and “Override Default Memory Placement for Subsystem Functions and Data” on page 45-7.
- If a model refers to a `Simulink.ConfigSet` object, after migration, the model does not use the memory section and shared utility settings in the object. If you used the object to set the configuration parameters of multiple models:

  - Changing memory section and shared utility settings in the object affects only models that you have not migrated.
  - Migrated models do not acquire memory section and shared utility settings from a single location (the `Simulink.ConfigSet` object). To change the settings for these models, use the Code Mappings editor of each model.

See Also

Embedded Coder Dictionary | **Code Mappings Editor – C**

Related Examples

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2
- “Generate Shared Utility Code” on page 8-52
- “C++ Data and Function Interfaces”

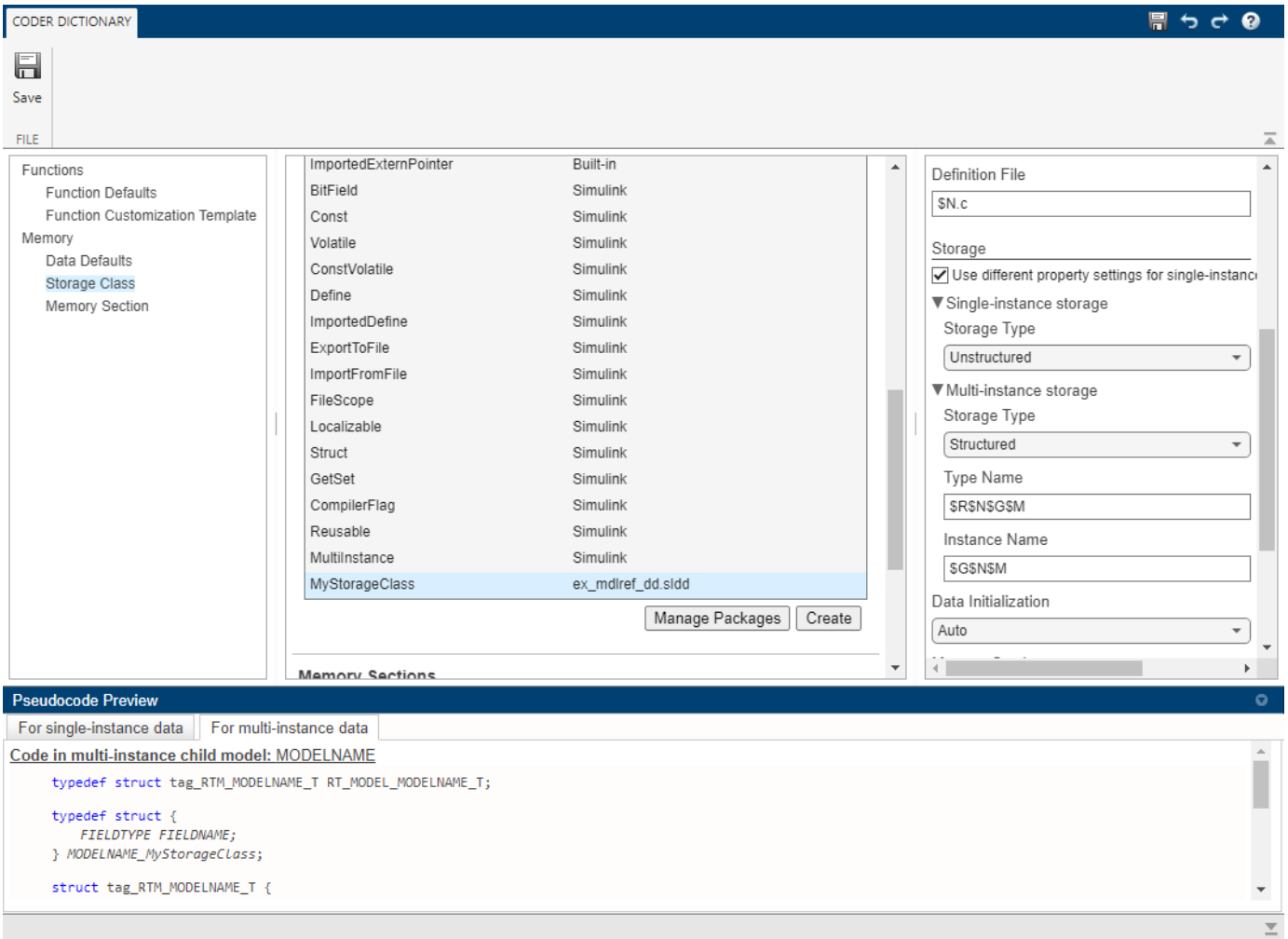
Flexible Storage Class for Different Model Hierarchy Contexts

This example shows how to use one storage class throughout a model hierarchy to generate code that is unstructured for single-instance data and structured for multi-instance data. When a model hierarchy contains single-instance and multi-instance data, use a flexible storage class to specify the settings for the two contexts instead of creating two separate storage classes.

This example uses the single-instance top model `ex_mdlreftop_dd`, which references the multi-instance model `ex_mdlrefbot_dd` three times. Both models share the data dictionary `ex_mdlref_dd.sldd`. When you define the storage class in the shared data dictionary, you can apply the class to data items in both models.

1. Open the example model `ex_mdlreftop_dd`.
2. Open the shared data dictionary. On the **Modeling** tab, click **Design** > **Data Dictionary**.
3. Open the shared Embedded Coder Dictionary. In the Model Explorer **Model Hierarchy** pane, expand the `ex_mdlref_dd` node and click **Embedded Coder Dictionary**. In the right pane, click **Open Embedded Coder Dictionary**.
4. On the left pane, click **Storage Class**.
5. The built-in storage class `MultiInstance` is configured to generate a structure for multi-instance data. Duplicate this storage class so that you can edit other properties. Select the `MultiInstance` storage class and click **Duplicate**.
6. In the **Storage Classes** section, click **Create**.
7. For the new storage class, in the property inspector pane, set these property values:

- **Name** — `MyStorageClass`
- **Use different property settings for single-instance and multi-instance data** — selected.
- **Single-instance storage** > **Storage Type** — `Unstructured`.
- **Multi-instance storage** > **Storage Type** — `Structured`.



When you apply `MyStorageClass` to a data item, the Embedded Coder Dictionary implements the single-instance settings or the multi-instance settings depending on the type of data and the context of the model within the model reference hierarchy.

1. Review the implementations for the different settings in the pseudocode preview.
2. Apply the storage class to internal data items by specifying it as a dictionary default. In the Embedded Coder Dictionary, click **Data Defaults**. For the **Signals, states, and internal data** row, set **Storage Class** to `MyStorageClass`. Click **OK**.Because `ex_mdlreftop_dd` and `ex_mdlrefbot_dd` share the dictionary `ex_mdlref_dd.sldd`, both models use `MyStorageClass` as the default storage class for internal data.
3. Open the Embedded Coder app for the model `ex_mdlreftop_dd`.
4. Generate code for the model.
5. Review the generated code for the referenced model. To open the referenced model in the editor, double-click the Model block, CounterA. The reference model code appears in the Code view. In `ex_mdlrefbot_dd.h`, the reference model code defines the storage class structure in which it stores the internal data of the referenced model.

```
/* Storage class 'MyStorageClass', for model 'ex_mdlrefbot_dd' */
```

```
typedef struct {

real_T PreviousOutput_DSTATE; /* '<Root>/Previous Output' */

} ex_mdlrefbot_dd_MyStorageClass;

/* Real-time Model Data Structure */

struct ex_mdlrefbot_dd_tag_RTM {

const char_T **errorStatus;

ex_mdlrefbot_dd_MyStorageClass *MyStorageClass_ex_mdlrefbot_dd;

};
```

Because the referenced model is multi-instance, the definition implements the multi-instance data settings of `MyStorageClass`. The reference model code stores the internal data in the structure `ex_mdlrefbot_dd_MyStorageClass`.

1. Navigate back to the top model and review the generated code. In `ex_mdlreftop_dd.c`, the top model code defines its internal data for each Model block by instantiating the storage class of the referenced model, `ex_mdlrefbot_dd_MyStorageClass`.

```
/* Storage class 'MyStorageClass' */

ex_mdlrefbot_dd_MdlrefDW ex_mdlreftop_dd_CounterA_InstanceData;

ex_mdlrefbot_dd_MdlrefDW ex_mdlreftop_dd_CounterB_InstanceData;

ex_mdlrefbot_dd_MdlrefDW ex_mdlreftop_dd_CounterC_InstanceData;

/* Storage class 'MyStorageClass' */

ex_mdlrefbot_dd_MyStorageClass MyStorageClass_CounterA;

/* Storage class 'MyStorageClass' */

ex_mdlrefbot_dd_MyStorageClass MyStorageClass_CounterB;

/* Storage class 'MyStorageClass' */

ex_mdlrefbot_dd_MyStorageClass MyStorageClass_CounterC;
```

Because the top model is single-instance, these definitions implement the single-instance data settings of `MyStorageClass`. The top model also packages its own internal data as standalone variables by using the single-instance data settings. The top model code does not contain a structure definition for internal data.

See Also

Embedded Coder Dictionary

Related Examples

- “Deploy Code Generation Definitions” on page 32-16
- “Organize Data into Structures in Generated Code” on page 36-147
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Control Placement of Data in Memory by Inserting Pragmas” on page 36-67

Separate C Code Interface Configuration Definition from Model-Specific Configurations

To generate C code for deployment to a specific target execution environment, in an Embedded Coder Dictionary, define a data or service code interface configuration definition. You can store the interface configuration definition in a shared Embedded Coder Dictionary, which is separate from your models. Separating the code interface configuration definition from your model-specific configurations in this way enables you to:

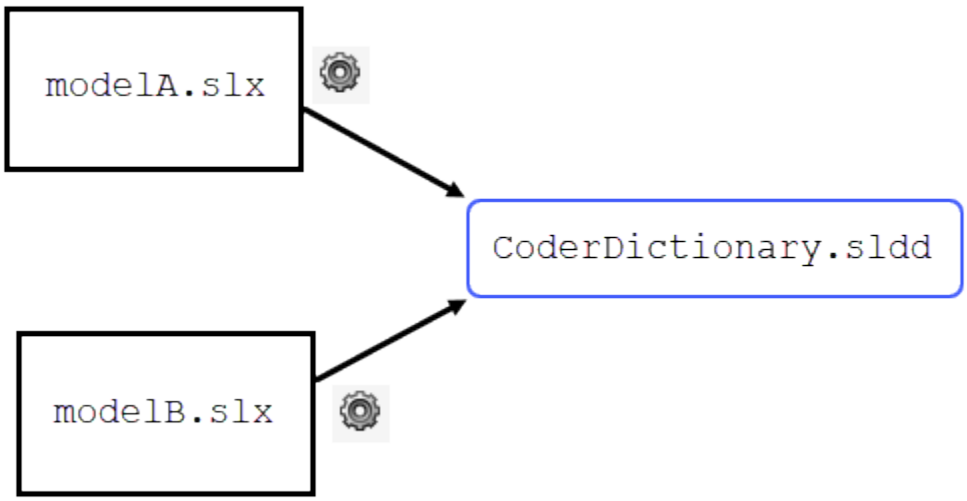
- Create, store, and maintain the interface specifications for your generated code in one place.
- Use one code interface configuration to generate code from multiple models for the same target execution environment.
- Select the code interface definition that the code generator uses by default for each category of interfaces.
- Configure the code interface of an individual model element by mapping it to a specific code interface definition.
- Use one model to generate code that is configured differently for different target execution environments.

For information on creating a shared coder dictionary, see “Configure Default Code Mapping in a Shared Dictionary” on page 32-16.

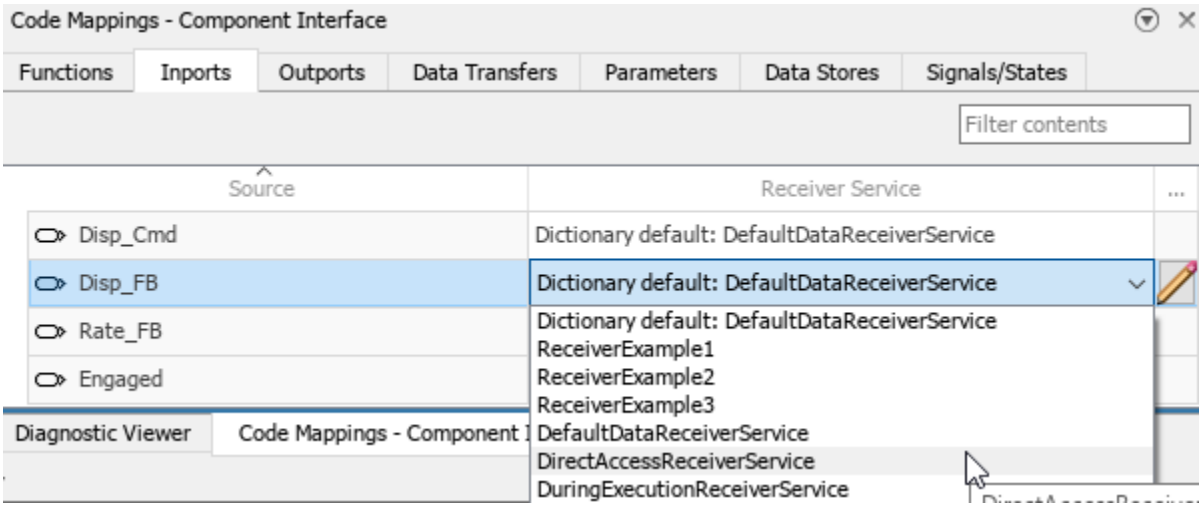
Set Up Models to Use Standard Code Interface Configuration

When you separate your code interface configuration definition from your model, you can set up multiple models to use the same code interface configuration. Sharing the definition among multiple models enables you to generate standardized code for the same target execution environment from the models. The typical setup of models with a code interface configuration is:

- The code interface configuration is saved in a shared coder dictionary `.sldd` file.
- The model configuration parameter **Shared coder dictionary** is used to attach the shared coder dictionary to each model.



- The code interface definitions from the shared code interface configuration definition appear in the Code Mapping editor for the models that use the shared dictionary. You can configure the code interface of each model individually by mapping:
  - Model functions to function customization templates
  - Model data elements to storage classes or service interfaces, depending on the type of interface configuration that you choose



- The shared coder dictionary specifies a dictionary default definition for each code interface category. If you do not select a different definition in the Code Mappings editor, the code generator uses the dictionary default definition for code generation.

By using this setup, you can maintain the code interface configuration definition for multiple models in one location. When you make changes to the code interface configuration definition, you can generate code that reflects the latest configuration without changing the models. You can easily apply standard code interfaces to the models to which the dictionary is attached by selecting the dictionary defaults, and you can apply non-default code interfaces to individual models by using the Code Mappings editor or the code mappings programming interface.

For more information about sharing your code interface configuration definition with models, see “Deploy Code Generation Definitions” on page 32-16.

Apply Code Interface Definitions to Model Elements with Flexibility

Storing code interface definitions in a shared coder dictionary gives modelers flexibility in applying the definitions to a model because the code definitions are separate from the elements in a model. Configure data elements and functions in your model by mapping them to code definitions. You can map data elements and functions to code definitions in multiple ways, depending on the type of code interface configuration definition that you use.

Data Interface Configuration

When you use a data interface configuration, the code generator creates a program intended to run on the target device. The generated code uses the data interfaces that you define. You can apply the code definitions from a data interface configuration to models in these ways:

- Apply a definition to a category of data elements or functions by default for the models to which the shared coder dictionary is attached by specifying the definition as the dictionary default for the category.
- Modelers can apply a definition to a category of data elements or functions by default in one model by specifying the definition as the model default for that category. The model default selection overrides the dictionary default for the shared coder dictionary that the model uses.
- Modelers can apply a definition to one data element or function in one model by specifying the definition using the mapping for individual model elements. The selection for an individual model element overrides the dictionary default and model default selections for that function or data element.

For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Service Interface Configuration

When you use a service interface configuration, the code generator creates an algorithm that you intend to deploy within a larger application. The target platform middleware calls the algorithm code and provides the services for the target device. The generated code calls the services according to your definitions in the service interface configuration. You can apply the code definitions from a service interface configuration to models in these ways:

- Apply a definition to a category of data elements or functions by default for the models to which the shared coder dictionary is attached by specifying the definition as the dictionary default for the category.
- Modelers can apply a definition to one data element or function in one model by specifying the definition using the mapping for individual model elements. The selection for an individual model element overrides the dictionary default and model default selections for that function or data element.

For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Generate Code for Different Platforms

By separating code definitions from your design data, you can generate code for different target platforms from the same model or model hierarchy. For example, you can share an Embedded Coder Dictionary with models for an embedded real-time (ERT) system target file. For each of the models, maintain a separate AUTOSAR dictionary with definitions for an AUTOSAR system target file. To configure the models for the ERT system target file:

1. Specify an ERT-based system target file for the models.
2. Create code interface definitions in the shared coder dictionary.
3. For each model, in the Code Mappings editor, map categories of model elements and individual model elements to definitions in the shared coder dictionary.

When you specify the ERT-based system target file, the models use this code mapping.

When you use an AUTOSAR code generation target, you define the interface in the AUTOSAR dictionaries and configure models by using the AUTOSAR code mappings. For more information, see “AUTOSAR Component Configuration” (AUTOSAR Blockset). When you specify the AUTOSAR-based system target file, the models use the AUTOSAR code mapping.

To generate code for either platform, specify the system target file. The code mapping for the model is updated to reflect the mapping and definitions that you specified for that platform. For more information, see “Manage Multiple Code Mappings for a Model” on page 33-92.

See Also

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Deploy Code Generation Definitions” on page 32-16
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

Data, Function, and File Definition

Configuring Data and Functions in the Generated Code

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Code Definition and Mapping Limitations and Considerations” on page 33-11
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Configure Root-Level Inport Blocks for C Code Generation” on page 33-30
- “Configure Root-Level Outport Blocks for C Code Generation” on page 33-41
- “Configure Signal Data for C Code Generation” on page 33-50
- “Configure Parameters for C Code Generation” on page 33-61
- “Configure Block States for C Code Generation” on page 33-74
- “Configure Data Stores for C Code Generation” on page 33-82
- “Manage Multiple Code Mappings for a Model” on page 33-92
- “Configure Model Data Interface for C Code Generation” on page 33-96
- “Migration of Model Data Configurations to Code Mappings” on page 33-103
- “Dimension Preservation of Multidimensional Arrays” on page 33-111
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Configure Multi-Instance Code Generation” on page 33-131
- “Configure Data Interfaces” on page 33-132
- “Configure Entry-Point Function Interfaces” on page 33-133
- “Configure Internal Data” on page 33-135
- “Configure Subsystem Function Packaging and Interface” on page 33-136

C Code Generation Configuration for Model Interface Elements

The code generator provides tooling for you to control the representation of the data and, if you have Embedded Coder, entry-point functions in the generated code. For example, you can configure data and functions for code generation to comply with code standards and guidelines, integrate generated code with external code, or address memory requirements.

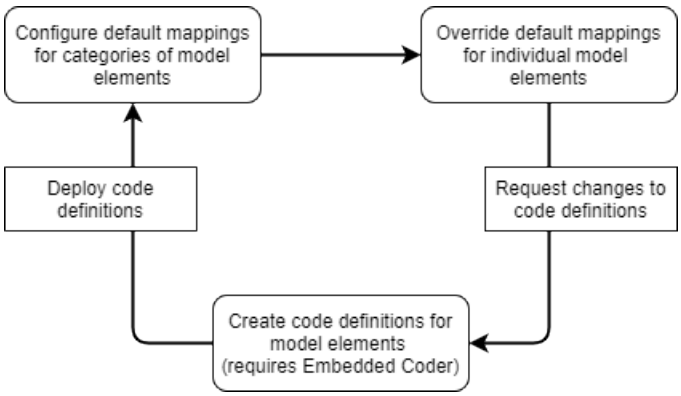
The code generator produces code for model data elements and functions:

- Data such as inports, signals, states, and parameters, which your application code can read from and write to.
- Entry-point functions such as `model_step`, which your application code calls to execute the model algorithm.

You can configure model data elements and functions for code generation by using the **Code Mappings Editor – C** or code mappings programming interface (`coder.mapping.api.CodeMapping`). Embedded Coder is required to configure functions. For data elements, use the Code Mappings editor or code mappings API instead of, or in addition to, creating and configuring Simulink data objects. The editor and API enable you to configure:

- Default code generation settings that apply throughout a model for a category of data elements or functions
- Configuration settings that apply to individual data elements or functions

This diagram shows the overall workflow:



Migration

When you open a model created in a release before R2020b, Simulink® migrates code definitions specified in the model to the model code mappings.

If necessary, as part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses. When you save the model, the code mappings are saved as part of the model.

For more information, such as effects of migration and what to consider before migrating, see “Migration of Model Data Configurations to Code Mappings” on page 33-103 .

Code Definitions

Code definitions include storage classes, function customization templates, and memory sections. A storage class defines properties, such as appearance and location, that the code generator uses when producing code for the associated data. A function customization template defines how the code generator produces code for functions. For a function category, you can define a rule for naming functions and a location in memory for function definitions (memory section).

Memory sections control the placement of data and function definitions in memory. In some cases, you can select a specific memory section that you want when configuring data elements and functions.

The code generator provides predefined code definitions for data elements. If you have Embedded Coder, you can use the Embedded Coder Dictionary to create and share code definitions, for example, to standardize code that you generate from multiple models or to create definitions for different application run-time environments. Creating custom definitions enables you to achieve code generation objectives that predefined definitions cannot satisfy. For data elements, you also have the option of creating code definitions created in packages by using Embedded Coder Custom Storage Class Designer.

Code definitions that you create by using the Embedded Coder dictionary appear in the **Code Mappings Editor – C** and are accessible by using the code mappings API. For some definitions that you create by using the Custom Storage Class Designer, you can make the definitions appear in the Code Mappings editor by configuring an Embedded Coder Dictionary to refer to them.

By default, code definitions that you create by using the Embedded Coder Dictionary for a model are stored in the model file. You can share code definitions between models and projects by storing the definitions in a Simulink data dictionary. With a data dictionary, to modify a shared definition, you change a definition in one place—the Embedded Coder Dictionary.

For more information, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2 and “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.

Data Default Configurations

Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of data elements, for example inports or model workspace parameters, across a model. Applying default configurations can save time and reduce the risk of introducing errors in code, especially for larger models and models from which you generate multi-instance code.

Default configurations apply throughout a model, excluding referenced models. Simulink saves a separate set of code mappings as part of each model in a model reference hierarchy.

If a model includes a significant number of elements of a given category (for example, more than 10), it is more efficient to configure the category by using a default setting, and then override that setting for special cases.

The default settings can help you to reduce manual data entry.

- As you add blocks and signals to a model, new data elements inherit the default settings.
- To change code generation settings for many data elements at once, make the changes in one place—the default settings.

- For efficiency, use naming rules as you make changes elsewhere in the model.

You configure data defaults by using the **Code Mappings Editor – C Data Defaults** tab or the `setDataDefault` function.

Model Data Categories

You configure data defaults by using the **Code Mappings Editor – C Data Defaults** tab or the `setDataDefault` function. In the **Data Defaults** tab or function call, you can select or specify these categories of data elements.

Model Element Category	Description
Inports	Root-level input ports of a model, such as Inport and In Bus Element blocks.
Outports	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.
Shared local data stores	Data Store Memory blocks that have the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model.
Global data stores	Data stores that are defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. To view and configure these data stores in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Model parameter arguments	Parameters in the model workspace that you configure as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the **Model Data Editor** > **Parameters** > **Argument** check box.
Model parameters	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
External parameters	Parameters that you define as objects in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. To view and configure these parameters in the Code Mappings editor, click the `Refresh` link to the right of the category name. Clicking this link updates the model diagram.
Constants	Constant-value block output and parameters that could not be inlined. These values are stored in variables for one of the following reasons. • The value is an array larger than the loop unrolling threshold. • The value address is needed in the code.

If you set the default storage for a category of data elements to `Default`, you have the option of specifying a memory section.

Considerations for Configuring Default Code Generation Settings for Data

Before configuring default code generation settings for data elements, take into account the considerations listed in this table.

Consideration	More Information
What categories are relevant to your model?	The data element categories in the preceding table
Does the model use several instances of data that is in a category? If the answer is yes, applying default mappings is beneficial. Otherwise, consider configuring code generation for each data element individually.	• "Configure Root-Level Inport Blocks for C Code Generation" on page 33-30 • "Configure Root-Level Outport Blocks for C Code Generation" on page 33-41 • "Configure Signal Data for C Code Generation" on page 33-50 • "Configure Parameters for C Code Generation" on page 33-61 • "Configure Block States for C Code Generation" on page 33-74 • "Configure Data Stores for C Code Generation" on page 33-82
Which storage class aligns with your code generation requirements for each category?	"Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
Do you want model data to be structured? For example, if you configured the model for multi-instance code generation, structures can improve code efficiency and readability.	"Organize Data into Structures in Generated Code" on page 36-147
Is the code generator required to store data in specific areas of memory? For example, consider whether to store initialization data in slow memory and algorithmic or computational data in fast memory.	• "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2 • "Control Data and Function Placement in Memory by Inserting Pragmas" on page 45-2
Do you want to prevent optimizations from eliminating specific data from the code?	"Elimination of Categories of Data by Optimizations" on page 33-6
Do you need to define new storage classes that you can select in the Code Mapping editor?	• "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2 • "Create Storage Classes by Using the Custom Storage Class Designer" on page 40-14

After you configure one or more categories of model elements, setting memory sections and **Shared utilities identifier format** model configuration parameters has no effect on the model configuration. When you open a coder app, Simulink migrates the model configuration parameter settings to the Code Mappings editor. As part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses, as described in "Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings" on page 32-31.

Elimination of Categories of Data by Optimizations

Code generation optimizations can eliminate data from the code, which means your application code cannot interact with the data. For general information about data elimination by optimizations, see "How Generated Code Stores Internal Signal, State, and Parameter Data" on page 36-49.

The optimizations can eliminate data in these categories only:

- **Model parameters**
- **Model parameter arguments**
- **External parameters**
- **Internal data**

After eliminating data with optimizations, the code generator applies the default code generation settings in the code mappings to the remaining data. For parameters and signal lines, optimizations can eliminate data for an entire category. If this elimination occurs, default settings that you specify for that category do not apply to data.

To prevent optimizations from eliminating individual data elements, explicitly apply a storage class to the individual element. The storage class controls the appearance of the data in the generated code. To force a data element to use the default storage class that you specify, explicitly apply the storage class `Model default`. See "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75.

Shared Dictionary Data Defaults

If you link a model to a Simulink data dictionary, which includes a coder dictionary that configures default code definitions for categories of data, you can use the Code Mappings editor to apply the dictionary defaults. In the Code Mappings editor, on the **Data Defaults** tab, select a category and set the storage class to `Dictionary Default`. If someone makes a change to the default settings in the shared coder dictionary, the code generator applies the updated default settings when producing code for your model. See "Configure Default Code Mapping in a Shared Dictionary" on page 32-16.

Function Default Configurations

Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of and functions, such as initialize/terminate and execution functions, across a model. Applying default configurations can save time and reduce the risk of introducing errors in code, especially for larger models and models from which you generate multi-instance code.

Default configurations apply throughout a model, excluding referenced models. Simulink saves a separate set of code mappings as part of each model in a model reference hierarchy.

If a model includes a significant number of functions of a given category (for example, more than 10), it is more efficient to configure the category by using a default setting, and then override that setting for special cases.

The default settings can help you to reduce manual data entry.

- As you add blocks and signals to a model, new functions inherit the default settings.
- To change code generation settings for many functions at once, make the changes in one place—the default settings.

- For efficiency, use naming rules as you make changes elsewhere in the model.

Model Function Categories

You configure function defaults by using the **Code Mappings Editor – C Function Defaults** tab or the `setDataDefault` function. In the **Function Defaults** tab or function call, you can select or specify these categories of functions.

Model Function Category	Description
Initialize/Terminate	Entry-point functions for initialization and termination
Execution	Entry-point functions for initiating execution and resets
Shared utility	Shared utility functions

The code generator uses default naming rules to name entry-point functions. To integrate generated code with existing external code or to comply with naming standards or guidelines, you can adjust the default naming rule. Adjusting the default naming rule can save time, especially for multirate models for which the code generator produces a unique `step` function for each rate.

You might need to define a function customization template in the Embedded Coder Dictionary. Unless someone has defined function customization templates for a model, the Code Mappings editor presents `Default` as the only template option.

Considerations for Configuring Default Code Generation Settings for Functions

Before configuring default code generation settings for functions, take into account the considerations listed in this table.

Consideration	More Information
What categories are relevant to your model?	The function categories in the preceding table
Does the model use several instances of functions that are in a category? If the answer is yes, applying default mappings is beneficial. Otherwise, consider configuring code generation for each function individually.	“Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
Which function customization template aligns with your code generation requirements for each category?	“Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
Do categories require the code generator to store function code in specific areas of memory? For example, consider whether to store initialization data in slow memory and algorithmic or computational data in fast memory.	• “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2 • “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2

Consideration	More Information
Do you need to define new function customization templates that you can select in the Code Mapping editor?	• "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2 • "Create Storage Classes by Using the Custom Storage Class Designer" on page 40-14
Do you have function naming requirements? If the answer is yes, what are they? To which categories do the requirements apply?	• "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114 • "Identifier Format Control" on page 55-17

After you configure one or more categories of model elements, setting memory sections and **Shared utilities identifier format** model configuration parameters has no effect on the model configuration. When you open a coder app, Simulink migrates the model configuration parameter settings to the Code Mappings editor. As part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses, as described in "Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings" on page 32-31.

Shared Dictionary Function Defaults

If you link a model to a Simulink data dictionary, which includes a coder dictionary that configures default code definitions for categories of functions, you can use the Code Mappings editor to apply the dictionary defaults. In the Code Mappings editor, on the **Function Defaults** tab, select a category and set the function customization template to `Dictionary Default`. If someone makes a change to the default settings in the shared coder dictionary, the code generator applies the updated default settings when producing code for your model. See "Configure Default Code Mapping in a Shared Dictionary" on page 32-16.

Configurations for Individual Data Elements and Functions

After you configure model-wide default settings, you can override the defaults for individual data elements and functions. An individual data element or function configuration applies to a specific data element or function in a model. Configure data elements and functions individually when:

- The model includes a few elements of a given category that have unique source, naming, or placement requirements.
- You configured default settings and you want to override the default setting for an individual element.

For individual data elements, initially the storage class is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration.

- To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
- To override the default configuration, specify the storage class that meets the code generation requirements for that data element.

If a data element in a model is unnamed (for example, a signal has no label or a state has no name), you must specify a code identifier for the data element when configuring it for C code generation. Set the code identifier by specifying a value for the storage class property **Identifier**.

For individual functions, you can customize the function name. For execution functions, such as the step functions and Simulink functions for a model, you can configure function arguments.

Configure Data

1. Open the Embedded Coder app.
2. Configure default mappings for categories of data elements. Select **Code Interface** > **Default Code Mappings**. In the Code Mappings editor, on the **Data Defaults** tab,select a model element category and set the storage class. In the Property Inspector, set storage class properties to align with model requirements.

   If you need to define a storage class, use the Embedded Coder Dictionary. Select **Code Interface** > **Embedded Coder Dictionary**.
3. Decide whether to override default configuration settings for individual data elements. If you choose not to override settings, go to step 8.
4. Identify MATLAB variables that you want to configure individually and store the configurations in the model file. In the Model Explorer, convert the variables to `Simulink.Parameter` objects. The resulting data objects appear in the Code Mappings editor as model parameters.
5. Identify signals representing algorithm data that you want to monitor while the generated code executes and configure those signals individually. For each signal that you identify, do one of the following:

   - For signal data that is specific to a model (not shared), add the signals to the Code Mappings editor.
   - For shared signal data, create and configure `Simulink.Signal` objects.
6. Configure individual data elements. In the Code Mappings editor, click a category tab, select a data element, and set the storage class. In the Property Inspector, set storage class property values. To change the setting of model configuration parameter **Default parameter behavior**, click the link indicating the behavior when the storage class is set to `Auto`. In the Model Configuration Parameters dialog box, change the parameter setting to `Tunable` or `Inline`.

   For information on configuring specific types of data elements, see:
7. To view and configure external data objects that are stored in the base workspace or a data dictionary, in the Code Mappings editor, on the **Data Defaults** tab, click the `Refresh` link to the right of the category name.
8. Generate and review code.

For more details and examples, see:

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- **Code Mappings Editor – C**
- “Configure Root-Level Inport Blocks for C Code Generation” on page 33-30
- “Configure Root-Level Outport Blocks for C Code Generation” on page 33-41
- “Configure Signal Data for C Code Generation” on page 33-50
- “Configure Parameters for C Code Generation” on page 33-61

- “Configure Block States for C Code Generation” on page 33-74
- “Configure Data Stores for C Code Generation” on page 33-82
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

Configure Functions

1. Open the Embedded Coder app.
2. Configure default mappings for categories of functions. Select **Code Interface** > **Default Code Mappings**. In the Code Mappings editor, on the **Function Defaults** tab, select a function category and set the function customization template.

   You might need to define a function customization template. Templates are not available by default. To define a function customization template, use the Embedded Coder Dictionary. Select **Code Interface** > **Embedded Coder Dictionary**.
3. Decide whether to override default configuration settings for individual functions. If you choose not to override settings, go to step 5.
4. Configure individual entry-point functions. In the Code Mappings editor, click **Functions** tab, select a function, and set the function customization template. Alternatively, you can specify a function name. For execution functions, such as a model step function, you can customize the function arguments. Clicking a function preview link opens a dialog box that you can use to customize the entire function interface.
5. Generate and review code.

For more details and examples, see:

- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

See Also

Code Mappings Editor – C | **Embedded Coder Dictionary** | `coder.mapping.api.CodeMapping`

More About

- “Configure Root-Level Inport Blocks for C Code Generation” on page 33-30
- “Configure Root-Level Outport Blocks for C Code Generation” on page 33-41
- “Configure Signal Data for C Code Generation” on page 33-50
- “Configure Parameters for C Code Generation” on page 33-61
- “Configure Block States for C Code Generation” on page 33-74
- “Configure Data Stores for C Code Generation” on page 33-82
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

Code Definition and Mapping Limitations and Considerations

Limitations

- If a data element in a model is unnamed (for example, a signal has no label or a state has no name), you must specify a code identifier for the data element when configuring it for C code generation. Set the code identifier by specifying a value for the storage class property **Identifier**.
- You cannot use code mappings to configure MATLAB variables that are used as block parameters. Convert the variables to `Simulink.Parameter` objects in the model workspace. The resulting data objects appear in the Code Mappings editor as model parameters.
- Use code mappings to configure only one discrete state per block in a model.
- To map these data categories to a storage class defined in an Embedded Coder Dictionary, you must set the storage class property **Data Initialization** to `AutoDynamic` or `None`:
  - **Inports**
  - **Outports**
  - **Signals, states, and internal data**
  - **Shared local data stores**
- You cannot map data categories **External parameters** and **Global data stores**. For referenced models you cannot map data categories **Inports** and **Outports** to structured storage classes that are defined in an Embedded Coder Dictionary.
- You cannot map categories of data to:
  - Predefined structure-based storage classes, `Struct` and `BitField`
  - Package storage classes of type `FlatStructure` or `Other` that you create by using the Custom Storage Class Designer
- If you configure a model to use a code replacement library that includes a data alignment specification, you cannot configure default code mappings that involve storage classes defined in the Embedded Coder Dictionary for that model. For more information, see “Data Alignment for Code Replacement” on page 72-105.
- If a model contains a tunable, nonfinite parameter, for example, with a value of `inf`:
  - The code generator does not apply a specified default parameter mapping. The parameter appears as a field in the default parameter data structure `rtP` (see “Real-Time Model Data Structure” on page 36-32).
  - You cannot map an individual element to a storage class defined in an Embedded Coder Dictionary. If you do, the code generator returns an error. Choose one of these workarounds:
    - For the individual nonfinite parameter, set the storage class to a value other than `Auto` or `Model default`, as described in “Configure Parameters for C Code Generation” on page 33-61.
    - If only one block uses the variable or object, instead of using the variable or object, specify the nonfinite value directly in the model, in the block parameter.

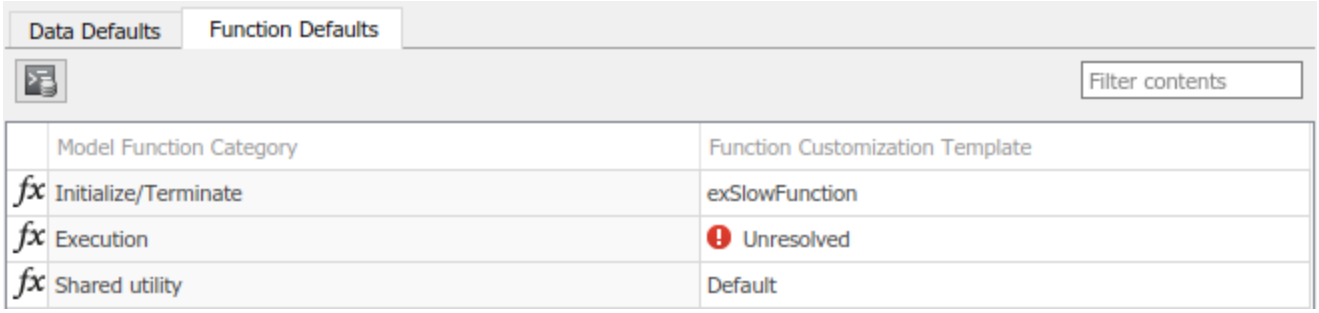
Default Code Mapping Constraints

- Map parameter categories to storage classes that are defined with the **Data Initialization** property set to `Auto`, `Static`, or `None`.

- If a model includes a reusable subsystem (a subsystem that has the block parameter **Function packaging** set to `Reusable function`) that contains a shared local data store, map the category **Shared local data stores** to a storage class that is defined in the Embedded Coder Dictionary for that model. For more information, see “Generate Reentrant Code from Subsystems” on page 8-28.
- For top and referenced models configured for multi-instance use, map data categories to storage classes that have the **Storage Type** property set to `Structured`. See “Generate Reentrant Code from Top Models” on page 8-13 and “Generate Reentrant Code from Simulink Function Blocks” on page 8-18.
- Default code mappings for categories **External parameters** and **Global data stores** must be the same for models throughout the model reference hierarchy.
- When a model is configured for compact file packaging, you cannot map a data category to a storage class defined with the **Data Scope** property set to `Exported` and the **Header File** property set to a naming rule other than `$N.h`.
- When evaluating storage class settings for variant control variables, the code generator ignores default code mappings. For more information, see “Represent Subsystem and Variant Models in Generated Code” on page 27-22.

Unresolved Code Definitions

If someone removes the definition for a storage class, function customization template, or memory section from the Embedded Coder Dictionary associated with a model, the Code Mappings editor identifies the definition as `Unresolved`. For example, this figure shows that the function customization template for execution entry-point functions was removed from the dictionary associated with the model.



To fix the unresolved code definition, do one of the following:

- Select a different definition in the **Storage Classes**, **Function Customization Templates**, or **Memory Sections** column.
- If you have Embedded Coder, replace or add a definition to the Embedded Coder Dictionary. Then, update the mapping.
- Consult with someone, such as the system architect for your project, about adding a definition to the Embedded Coder Dictionary. Then, update the mapping.

Model Editing Operations and Individual Data Element Code Mappings

While editing a model in the context of a coder app, you should be aware of the relationship between Simulink model editing tools, such as copy and undo, and code mappings. In the context of a coder

app, when you perform copy-and-paste or cut-and-paste editing operations within a model, the coder app preserves the code mappings for individual data elements. For example, within a model, if you copy and paste the source block of a signal that is configured with a specific storage class, the pasted version of the signal port retains the same storage class.

While you can use undo and redo operations for editing actions on model data elements, they are unavailable while configuring code mappings. For example, if you delete a state block, and then click **Undo**, the block reappears in the model canvas and the code configuration for the state reappears in the code mappings. In the Code Mappings editor, if you change the storage class setting for a state from `Auto` to `Model default`, and then click **Undo**, the storage class does not revert to `Auto`. In this case, the change pertains to the code mapping, not to the model data element.

Storage Classes and Reentrant, Multi-Instance Models and Components

The code that you generate from a model or component (referenced model or subsystem) is multi-instance code if it allows your application to maintain multiple independent instances of the component during execution. For example, you can generate reentrant, multi-instance code from an entire model by setting the model configuration parameter **Code interface packaging** to `Reusable function`. For general information about multi-instance models and components, see “Model Composition and Reuse”.

Under some circumstances, applying storage classes can:

- Prevent you from generating multi-instance code.
- Cause some data elements to appear in the generated code as singletons, which means that each instance of the model or component directly accesses the same shared, global data, creating dependencies between the instances.

To avoid errors and unexpected generation of single-instance code and singleton data, observe the guidelines and limitations below.

Directly Applied Storage Classes

When you apply a storage class directly to a data element (see “C Code Generation Configuration for Model Interface Elements” on page 33-2):

- To generate multi-instance code, you must apply the storage class only to parameter objects, global data stores, shared local data stores, and root-level I/O.
- The storage class yields only singleton data.

Storage Classes Applied by Default

When you apply a default storage class (see “C Code Generation Configuration for Model Interface Elements” on page 33-2):

- For these data categories, you can generate only singleton data:
  - **Model parameters**
  - **External parameters**
  - **Global data stores**

- **Shared local data stores**
- For **Internal data**, you can generate only instance-specific data, which means each instance of the model or component operates on a separate copy of the data. For this category, you must use the example storage class `SignalStruct`, which appears after preparing a model for code generation with the Quick Start tool. Alternatively, you can create and use your own structured storage class by using an Embedded Coder Dictionary.

For more information, see “What Is Reentrant Code?” and “Choose Storage Class for Controlling Data Representation in Generated Code”.

Copy Code Mappings When Converting Subsystems to Referenced Models

You can copy code mappings when you convert a subsystem to a referenced model.

- In the **Model Reference Conversion Advisor**, select **Copy code mappings**.
- In the MATLAB command window, use the `Simulink.SubSystem.convertToModelReference` function with the comma-separated pair consisting of `'CopyCodeMappings'` and `true`.

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Configure Default C Code Generation for Categories of Data Elements and Functions

Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of data elements and functions across a model. Applying default configurations can save time and reduce the risk of introducing errors in code, especially for larger models and models from which you generate multi-instance code.

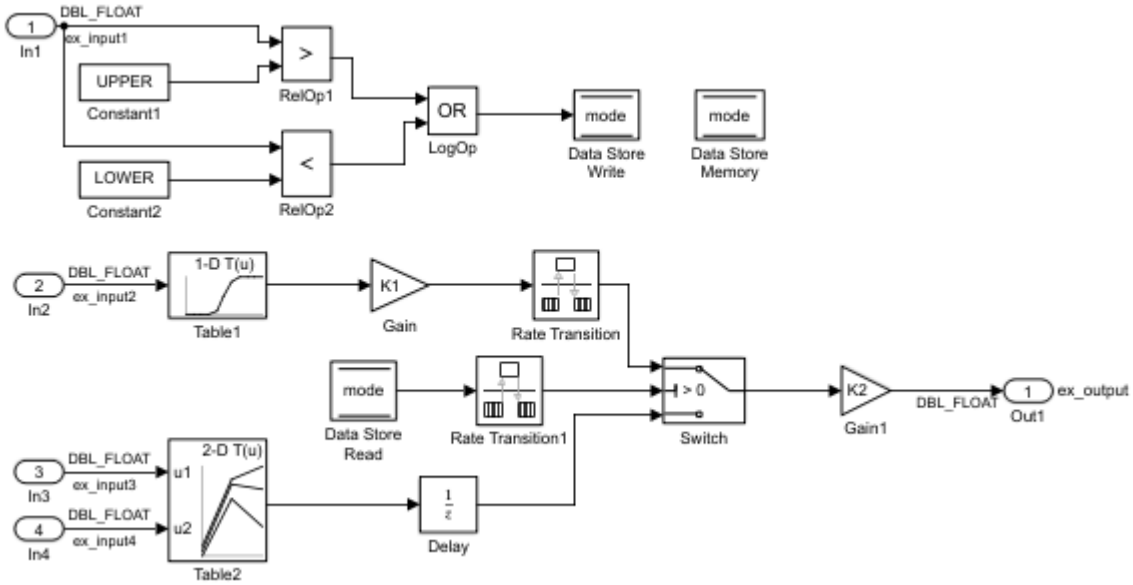
You must have set up the example environment, as described in “Set Up Example Environment” on page 33-15.

Set Up Example Environment

1. Copy these external source and header files from `matlabroot/toolbox/rtw/rtwdemos` to a writable working folder. These files define and declare the external data type and data that the generated code uses.

File	Description
`exDblFloat.h`	Defines the project data type alias for double, `DBL_FLOAT`. Simulink uses the `PreLoadFcn` callback specified for the model to parse this header file and create a corresponding `Simulink.AliasType` object.
`exInDataMem.c`	Includes `exInDataMem.h` and defines variable `ex_input1`.
`exInDataMem.h`	Includes `exDbFloat.h` and declares variable `ex_input1`.
`exInDataLut.c`	`exInDataLut.c` includes`exInDataLut.h` and defines variables `ex_input2`, `ex_input3`, and `ex_input4`.
`exInDataLut.h`	Includes `exDbFloat.h` and declares variables `ex_input2`, `ex_input3`, and `ex_input4`.
`exCodeDefs.sldd`	Data dictionary
`rtwdemo_configdefaults.slx`	Example model.

2 Open your copy of example model `rtwdemo_configdefaults`.



3 In the apps gallery, open the **Embedded Coder** app.

Configure Default Code Generation for Data

This example shows how to use the Code Mappings editor to specify code generation requirements for model data. The model uses multiple execution rates and is configured for single-instance usage.

This example assumes these code generation requirements for data:

- Use project type definition `DBL_FLOAT` defined in header file `exDblFloat.h`.
- Get data element `ex_input1` from header file `exInDataMem.h`. The data is used for computing a value stored in memory, and then used in an if-else condition of a Switch block.
- Get data elements `ex_input2`, `ex_input3`, and `ex_input4` from header file `exInDataLut.h`. The data is used in lookup tables `Table1` and `Table2`.
- Data imported into the model from header files `exInDataMem.h` and `exInDataLut.h` is of type `DBL_FLOAT`, a project standard.
- Parameter `K1` must be tunable to enable calibration.
- Data store `mode` defines data that is shared within the model. The Logical Operator block writes to the data store and a Rate Transition block reads from the data store.
- Data element `X` represents the delay for the Unit Delay block.
- Store data that is internal to the model, such as delay `X`, in a section of memory labeled `internalDataMem`.
- Export output data declarations to header file `exSysOut.h` and definitions to `exSysOut.c`.
- Name variables representing inports and outports in the generated code as specified in this table so they match variable names in the external header and definition files.

Although the model uses single instances of some data, the example configures default settings to demonstrate different types of mappings. As you add blocks to the model, new data elements acquire the default code generation mappings.

Configure Default Settings for Root Inports

Specify an external header file that declares input data. Three of the four root inports read input from variables declared in header file `exInDataLut.h`. Set that header file as the default.

1. Open model rtwdemo_configdefaults. Save a copy to a writable location.
2. Open the **Embedded Coder** app.
3. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**. This opens the Code Mappings editor **Data Defaults** tab.
4. In the**Data Defaults** tab, expand **Inports and Outports**. Then, select **Inports**.
5. Set the storage class to `ImportFromFile`.
6. Click the icon and set **Header File** to `exInDataLut.h`.

Configure Default Settings for Model Parameters

Configure default settings for model workspace parameters that you want to be tunable.

1. In the Code Mappings editor, click the **Data Defaults** tab.
2. In the**Data Defaults** tab, expand **Parameters**.
3. In the row for **Model parameters**, click the text `'Auto' will be inlined`. The Model Configuration Parameters dialog box opens to the **Code Generation** > **Optimization** pane. Set **Default parameter behavior** to `Tunable`. Apply the change and close the dialog box. In the Code Mappings editor, the text to the right of **Model parameters** now reads `'Auto' will be tunable`.
4. To include global variable definitions and declarations in the generated code for model parameters, set the storage class for the **Model parameters** category to `ExportedGlobal`.

Define Memory Section for Internal Data

Define a memory section for storing unit delay `X`, data that is internal to the model.

1. Open the Embedded Coder Dictionary by selecting **Code Interface** > **Embedded Coder Dictionary**.
2. In the Embedded Coder Dictionary, on the left pane, click **Memory Section**.
3. Click **Create**.
4. In the new row of the table, name the new memory section `internalDataMem`. Also set:
   - **Pre Statement** to `#pragma start INTERNALDATA`
   - **Post Statement** to `#pragma end INTERNALDATA`
   - **Statements Surround** to `Group of variables`
5. Close the dictionary.

For more information, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Configure Default Settings for Internal Data

Set up the default code generation configuration for internal data to include memory section `internalDataMem`.

1. In the Code Mappings editor, click the **Data Defaults** tab.
2. In the **Data Defaults** tab, expand **Signals**.
3. Select category **Signals, states, and internal data**.
4. Click the icon and set **Memory Section** to `internalDataMem`.

Configure Default Settings for Root Outport

Specify the default external header and definition files for variables to which the generated code writes output.

1. In the Code Mappings editor, click the **Data Defaults** tab.
2. In the**Data Defaults** tab, expand **Inports and Outports**.
3. Select category **Outports**.
4. Set the storage class to `ExportToFile`.
5. Click the icon and set **Header File** to `exSysOut.h` and **Definition File** to `exSysOut.c`.

Apply Default Configurations to Individual Data Elements

Unless you explicitly map individual data elements to alternative storage class settings, the Code Mappings editor assumes the storage class for elements is `Auto`. When the storage class for a data element is `Auto`, the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration.

- To avoid optimizations and force the code generator to use the default configuration, set the storage class for the individual element to `Model default`.
- To override the default configuration, specify the storage class that meets the code generation requirements for that data element.

For this example, configure the code generator to apply the default storage class setting to these data elements:

- Inports `In2`, `In3`, and `In4`
- Outport `Out1`
- Model parameter `K1`
- State `X`

1. In the Code Mappings editor, click the **Inports** tab.
2. Select the rows for inports `In2`, `In3`, and `In4`. Then, for one of the selected inports, set the storage class to `Model default: ImportFromFile`. The storage class for the three selected inports changes to `Model default: ImportFromFile`.
3. Click the **Outports** tab. For outport `Out1`, set the storage class to `Model default: ExportToFile`.
4. Click the **Parameters** tab. Expand **Model Parameters**. Then, for parameter `K1`, set the storage class to `Model default: ExportedGlobal`.
5. Click the **Signals/States** tab. Expand **States**. For state `X`, set the storage class to `Model default: Default`.

6 Save the model.

Generate and Verify Code

Generate code and verify the generated code.

- In the file `rtwdemo_configdefaults.h`, search for `#include` statements that include the header files that declare external input data.

  ```
  #include "exInDataMem.h"
  #include "exInDataLut.h"
  ```

- Open file `rtwdemo_configdefaults.c`. The code initializes the gain variable `K1` and uses the variable in the model step function `exFast_step1`.

  ```
  DBL_FLOAT K1 = 2.0;
  .
  .
  .
  void exFast_step1(void)
  {
    DBL_FLOAT rtb_Gain;

    rtb_Gain = K1 * look1_binlc(ex_input2, (&(rtTable1_bp01Data[0])),
      (&(rtTable1_tableData[0])), 10U);

    rtDWork.RateTransition_Buffer0 = rtb_Gain;
  }
    .
    .
    .
  }
  ```

- In file `rtwdemo_configdefaults.c`, find `#pragma` control lines that define memory sections for `INTERNALDATA` (the local data store, unit delay, and constants).

  ```
  #pragma start INTERNALDATA

  D_Work rtDWork;

  #pragma end INTERNALDATA
  .
  .
  .
  #pragma start INTERNALDATA

  static RT_MODEL rtM_;

  #pragma end INTERNALDATA
  .
  .
  .
  pragma start INTERNALDATA

  extern D_Work rtDWork;

  #pragma end INTERNALDATA
  ```

- Open file `exSysOut.c`. The file includes an exported data definition for `ex_output`.

  ```
  #include "rtwdemo_configdefaults.h"

  DBL_FLOAT ex_output;
  ```

- Open shared file `exSysOut.h`. The file declares `ex_output`. To gain access to `ex_output`, external code can include this header file.

  ```
  extern DBL_FLOAT ex_output;
  ```

Configure Default Code Generation for Functions

This example shows how to use the Code Mappings editor to specify code generation requirements for model functions. The model uses multiple execution rates and is configured for single-instance usage. The code generator produces initialize, execution, and terminate entry-point functions. Because the model uses multiple rates, the code generator produces an execution function for each rate.

This example assumes these code generation requirements:

- Store generated initialize and terminate functions in memory section `functionSlowMem` and execution functions in memory section `functionFastMem`.
- Use the naming rule `exSlow_$N` to name `initialize` and `terminate` functions. Use the naming rule `exFast_$N` to name execution functions.

Define Memory Sections

Define the two memory sections: `functionSlowMem` for initialize and terminate functions and `functionFastMem` for execution functions.

1. Open the **Embedded Coder** app.
2. Open the Embedded Coder Dictionary by selecting **Code Interface** > **Embedded Coder Dictionary**.
3. In the Embedded Coder Dictionary, on the left pane, click **Memory Section**.
4. Click **Create**.
5. In the new row of the table, name the new memory section `functionFastMem`. Then, set:
   - **Pre Statement** to `#pragma start FASTMEM`
   - **Post Statement** to `#pragma end FASTMEM`
6. Click **Add** again. Name the memory section `functionSlowMem`. Then, set:
   - **Pre Statement** to `#pragma start SLOWMEM`
   - **Post Statement** to `#pragma end SLOWMEM`

Define Function Customization Templates

To configure categories of functions, define function customization templates. Unless you define templates in the Embedded Coder Dictionary that are associated with a model, the only available template is `Default`. Based on the requirements, in the dictionary, define two function customization templates: one to specify the naming rule and memory section for initialize and terminate functions and one to specify the naming rule and memory section for execution functions.

1. In the Embedded Coder Dictionary, click **Function Customization Template**.
2. Click **Create**.
3. In the new row of the table, name the new template `exFastFunction`. Then, set:
   - **Function Name** to `exFast_$N`. This naming rule applies the prefix `exFast_` to the name that identifies the default code generator name of the function (for example, `initialize` or `step`).
   - **Memory Section** to `functionFastMem`. This mapping associates the memory section that you defined in “Define Memory Sections” on page 33-20 with the new template.

4. Click **Create** again. Name the template `exSlowFunction`. Then, set:
   - **Function Name** to `exSlow_$N`.
   - **Memory Section** to `functionSlowMem`.
5. Close the dictionary.

Configure Default Settings for Functions

1. In the **C Code** tab, click **Code Interface** > **Default Code Mappings**.
2. In the Code Mappings editor, click the **Function Defaults** tab.
3. Configure the initialize and terminate entry-point functions. For category **Initialize/Terminate**, select template `exSlowFunction`.
4. Configure the execution entry-point functions. For category **Execution**, select template `exFastFunction`.
5. Save the model.

Generate and Verify Code

1. Generate code.
2. In the **Code** view:
3. - Open file `rtwdemo_configdefaults.c`. Click in the **Search** field. A menu lists the generated entry-point functions:
     - `exFast_step0` (called periodically, every 0.5 seconds)
     - `exFast_step1` (called periodically, every 1 second)
     - `exFast_step2` (called periodically, every 1.5 seconds)
     - `exSlow_initialize`
   - To gain access to the entry-point function code in `rtwdemo_configdefaults.c`, click the function name. Verify the `pragma` control statements that surround the function code. For example:

```
.
.
.
#pragma start FASTMEM

void exFast_step2(void)                /* Sample time: [1.5s, 0.0s] */
{
   boolean_T rtb_DataStoreRead;

   rtb_DataStoreRead = ((ex_input1 > 10.0) || (ex_input1 < -10.0));

   rtDWork.RateTransition1_Buffer0 = rtb_DataStoreRead;
}

#pragma end FASTMEM

#pragma start SLOWMEM

void exSlow_initialize(void)
{
   /* (no initialization code required) */
}

#pragma end SLOWMEM
.
.
.
```

- Open file `rtwdemo_configdefaults.h`. Use **Search** to find `#pragma` control lines that define memory sections for `FASTMEM` and `SLOWMEM`. Verify the `pragma` control statements surround the declarations. For example:

```
.
.
.
#pragma start SLOWMEM

extern void exSlow_initialize(void);

#pragma end SLOWMEM
#pragma start FASTMEM

extern void exFast_step0(void);

#pragma end FASTMEM
.
.
.
```

Configure Model Elements for Code Generation with Definitions Stored in Shared Data Dictionary

You have the option of configuring default data and function code generation with definitions that are set up in a Simulink data dictionary. A data dictionary enables sharing of code definitions between models. This example shows how to change a model from using code definitions in a model-specific Embedded Coder Dictionary to using definitions in an Embedded Coder Dictionary that is in a shared data dictionary. If you have completed the examples in “Configure Default Code Generation for Data” on page 33-16 and “Configure Default Code Generation for Functions” on page 33-20 you have added these code definitions to the Embedded Coder Dictionary associated with model `rtwdemo_configdefaults`.

- Memory section `internalDataMem` with **Pre Statement** set to `#pragma start INTERNALDATA` and **Post Statement** set to `#pragma end INTERNALDATA`.
- Memory section `functionFastMem` with **Pre Statement** set to `#pragma start FASTMEM` and **Post Statement** set to `#pragma end FASTMEM`.
- Memory section `functionSlowMem` with **Pre Statement** set to `#pragma start SLOWMEM` and **Post Statement** set to `#pragma end SLOWMEM`.
- Function customization template `exFastFunction` with function naming rule `exFast_$N` and memory section `functionFastMem`
- Function customization template `exSlowFunction` with function naming rule `exSlow_$N` and memory section `functionSlowMem`

Update model `rtwdemo_configdefaults` to use the same code definitions in data dictionary exCodeDefs.sldd instead of definitions in the local model Embedded Coder Dictionary.

You must have set up the example environment, as described in “Set Up Example Environment” on page 33-15.

Attach Shared Data Dictionary to Model

1. In the Simulink Editor, select **Modeling** > **Link to Data Dictionary**.
2. In the Model Properties dialog box, in the **External Data** tab, browse to the location of your copy of data dictionary file `exCodeDefs` and select that file.

3. Click **Migrate data**.
4. In the Link Model to Data Dictionary dialog box, click **Apply**.
5. In the Migrate Data dialog box, click **Migrate**. When the data migration is complete, click **OK**.

Review Contents of Attached Data Dictionary

1. In the lower-left corner of the model canvas, click the **Model data** icon.
2. From the list of model data sources, click **External Data**.
3. In the Model Explorer, in the **Model Hierarchy** pane, expand the **exCodeDefs** node.
4. Right-click **Embedded Coder Dictionary**.
5. Click the **Open Embedded Coder Dictionary** button that appears.
6. In the Embedded Coder Dictionary, review the definitions in the **Function Customization Template** and **Memory Section** tabs.
7. Close the Embedded Coder Dictionary.

Remove Model Sourced Code Definitions from Model Embedded Coder Dictionary

1. Open the **Embedded Coder** app.
2. In the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.
3. Remove the code definitions created locally in the model.
   - In the **Function Customization Templates** table, select the rows for `exSlowFunction` and `exFastFunction` that have **Source** set to `rtwdemo_configdefaults`. Click the **Delete** icon.
   - In the **Memory Sections** table, select the rows for `functionFastMem`, `functionSlowMem`, and `internalDataMem` that have **Source** set to `rtwdemo_configdefaults`. Click the **Delete** icon.
4. Close the dictionary.
5. Save the model.

Configure Default Categories for Code Generation

1. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
2. In the Code Mappings editor, click the **Data Defaults** tab.
3. Expand **Signals**. Select category **Signals, states, and internal data**.
4. Click the icon. Set **Memory Section** to `internalDataMem`.
5. Click the **Function Defaults** tab.
   - For the **Initialize/Terminate** category, select function customization template `exSlowFunction`.
   - For the **Execution** category, select template `exFastFunction`.
6. Save the model.
7. Generate and review code.

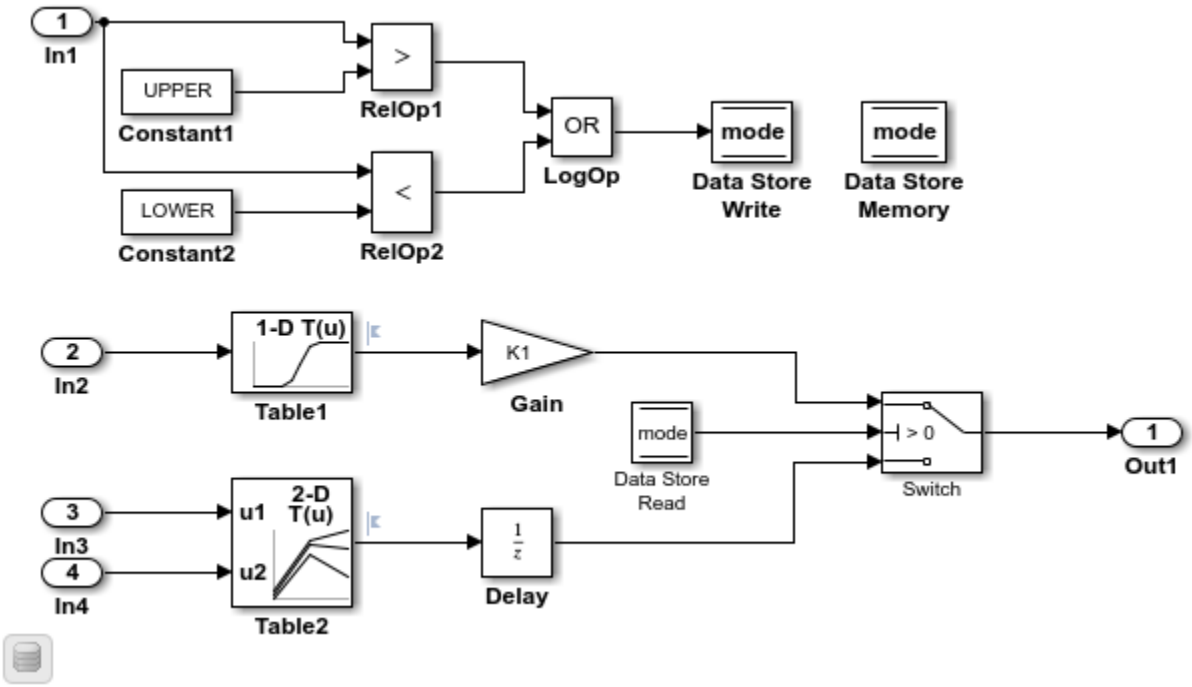
For more information about setting up an Embedded Coder Dictionary, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

Configure Default Data and Function Code Generation Programmatically

This example shows how to configure default data and function code generation for example model `rtwdemo_configdefaults`. The example uses the default mapping programming interface to specify code generation requirements for model data and functions. Use that interface to automate the configuration, or if you prefer to configure models programmatically. For information about configuring the default data and function code generation by using the Code Mappings editor, see "Configure Default C Code Generation for Categories of Data Elements and Functions" on page 33-15.

Open the Model

The model `rtwdemo_configdefaults` uses multiple execution rates and is configured for single-instance usage.




```
open_system('rtwdemo_configdefaults')
```

Code Generation Requirements

For this example, these are the code generation requirements:

- Import project type definition for data of type `double`, DBL_FLOAT, from header file `exDblFloat.h`.
- Import signal `ex_input1` for computing a value stored in memory and used in an if-else condition in the Switch block. Import the signal data from header file `exInDataMem.h`.
- Import signals `ex_input2`, `ex_input3`, and `ex_input4` for lookup tables `Table1` and `Table2`. Import the signal data from header file `exInDataLut.h`.

- Data imported into the model from header files `exInDataMem.h` and `exInDataLut.h` is of type `DBL_FLOAT`, a project standard.
- Parameters `UPPER`, `LOWER`, `K1`, and `K2` are parameter objects stored in the model workspace. Represent the parameters as global variables in the generated code.
- Data element `X` represents the delay for the Unit Delay block.
- Store data that is internal to the model in memory section `internalDataMem`.
- Store generated initialize and terminate functions in memory section `functionSlowMem` and execution functions in memory section `functionFastMem`.
- Use the identifier naming rule `exSlow_$N` to name `initialize` and `terminate` entry-point functions. Use the naming rule `exFast_$N` to name execution functions for .5 second and 1.5 second periodic functions. Set the name of the 1 second periodic function to `exFast_1sec`.
- Export output data declarations to header file `exSysOut.h` and definitions to `exSysOut.c`.

For this example, someone, such as a system architect, has previously created these code definitions in an Embedded Coder Dictionary that is part of Simulink data dictionary `exCodeDefs.sldd`:

- Memory sections `internalDataMem`, `functionFastMem`, and `functionSlowMem`.
- Function customization templates `exFastFunction` and `exSlowFunction`.

Get Data and Function Code Mappings for Model

Get the code mappings for example model `rtwdemo_configdefaults` by specifying the name of the model in a call to function `coder.mapping.api.get`. The function returns an object that represents the code mappings for the model. You specify that object as the first argument in subsequent calls to other functions in the API.

If code mappings do not exist for a model, create a code mappings object by calling `coder.mapping.utils.create`.

```
cm = coder.mapping.api.get('rtwdemo_configdefaults');
```

Set Relevant Category, Property, and Value Combinations

Set relevant category, property, and value combinations with calls to `setDataDefaults` and `setFunctionDefaults`. The first two arguments that you specify for these functions are the code mappings object returned by `coder.mapping.api.get` and the name of a data or function category. In addition, you specify name-value pair arguments that specify the default configuration information that you want to set, such as the storage class and storage class properties.

In this example, you set the default:

- Storage class for Inport blocks, Outport blocks, and model parameters
- Memory section for internal data (for example, signald and block states)
- Header and definition files for Inport and Outport blocks
- Function customization template for initialize, terminate, and execution functions

```
% Configure data defaults
setDataDefault(cm,'Inports','StorageClass','ImportFromFile','HeaderFile','exInDataLut.h');
setDataDefault(cm,'Outports','StorageClass','ExportToFile','HeaderFile','exSysOut.h',...
    'DefinitionFile','exSysOut.c');
setDataDefault(cm,'ModelParameters','StorageClass','ExportedGlobal');
setDataDefault(cm,'InternalData','MemorySection','internalDataMem');
```

```
% Configure function defaults
setFunctionDefault(cm,'InitializeTerminate','FunctionCustomizationTemplate','exSlowFunction');
setFunctionDefault(cm,'Execution','FunctionCustomizationTemplate','exFastFunction');
```

Verify Default Mappings

Verify default mappings with calls to `getDataDefaults` and `getFunctionDefaults`. The first two arguments that you specify for these functions are the code mappings object returned by `coder.mapping.api.get` and the name of a data or function category. In addition, you specify the name of the configuration information that you want the function to return, such as the name of the storage class or a storage class property.

In this example, you verify these default settings:

- Storage class for Inport blocks, Outport blocks, and model parameters
- Memory section for internal data (for example, signal data and block states)
- Header and definition files for Inport and Outport blocks
- Function customization template for initialize, terminate, and execution functions

```
% Verify default data configurations
defscInports = getDataDefault(cm,'Inports','StorageClass')

defscInports =
'ImportFromFile'

defhfileInports = getDataDefault(cm,'Inports','HeaderFile')

defhfileInports =
'exInDataLut.h'

defscOutport = getDataDefault(cm,'Outports','StorageClass')

defscOutport =
'ExportToFile'

defhfileOutport = getDataDefault(cm,'Outports','HeaderFile')

defhfileOutport =
'exSysOut.h'

defdffileOutport = getDataDefault(cm,'Outports','DefinitionFile')

defdffileOutport =
'exSysOut.c'

defscParams = getDataDefault(cm,'ModelParameters','StorageClass')

defscParams =
'ExportedGlobal'

defmemInternal = getDataDefault(cm,'InternalData','MemorySection')

defmemInternal =
'internalDataMem'


% Verify default function configurations
deftempInitTerm = getFunctionDefault(cm,'InitializeTerminate','FunctionCustomizationTemplate')
```

```
deftempInitTerm =
'exSlowFunction'
```

```
deftempExecution = getFunctionDefault(cm,'Execution','FunctionCustomizationTemplate')
```

```
deftempExecution =
'exFastFunction'
```

Configure Individual Data Elements to Use Default Configuration Settings

The storage class for each model data element is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration.

- To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
- To override the default configuration, specify the storage class that meets the code generation requirements for that inport.

For this example, you configure the code generator to apply the default storage class setting to the Inport blocks, the Output block, model parameters, and state `X` for the Unit Delay block. Use the `find` function to get the names of the data elements in the model of the different categories. Then, use the values returned by `find` in calls to `setInport`, `setOutport`, `setModelParameter`, and `setState` to set the storage class to `Model default`. Use calls to `getInport`, `getOutport`, `getModelParameter`, and `getState` to verify, the storage class settings.

In each of the function calls, you specify the code mappings object returned by `coder.mapping.api.get`. In addition:

- In the call to the `find` function, you specify a value indicating code mapping information that you want the function to return.
- The calls to `setInport`, `setOutport`, `setModelParameter`, and `setState` identify data elements that you want to configure and a name-value pair argument that specifies the configuration information that you want to set, such as the storage class or a storage class property.
- The calls to `getInport`, `getOutport`, `getModelParameter`, and `getState` identify a data element for which you want to return configuration information and the type of information that you want the function to return.

```
input = find(cm,'Inports');
setInport(cm,input,'StorageClass','Model default');

output = find(cm,'Outports');
setOutport(cm,output,'StorageClass','Model default');

params = find(cm,'ModelParameters');
setModelParameter(cm,params,'StorageClass','Model default');

states = find(cm,'States');
setState(cm,states,'StorageClass','Model default');

scIn1 = getInport(cm,'In1','StorageClass')
```

```
scIn1 =
'Model default'
```

```
scIn2 = getInport(cm,'In2','StorageClass')
```

```
scIn2 =
'Model default'
```

```
scIn3 = getInport(cm,'In3','StorageClass')
```

```
scIn3 =
'Model default'
```

```
scIn4 = getInport(cm,'In4','StorageClass')
```

```
scIn4 =
'Model default'
```

```
scK1 = getModelParameter(cm,'K1','StorageClass')
```

```
scK1 =
'Model default'
```

```
scTable2 = getModelParameter(cm,'Table2','StorageClass')
```

```
scTable2 =
'Model default'
```

```
scK2 = getModelParameter(cm,'K2','StorageClass')
```

```
scK2 =
'Model default'
```

```
scX = getState(cm,'rtwdemo_configdefaults/Delay','StorageClass')
```

```
scX =
'Model default'
```

Override Default Header File Setting for Inport Block In1

Previously, you set the default header file for inports to `exInDataLut.h`. The requirements specify that you import data for Inport block `In1` from header file `exInDataMem.h`.

For `Inport` block `In1`, override the default storage class to `ImportFromFile` and set the header file to `exInDataMem.h` with a call to `setInport`. The function call specifies the code mappings object returned by `coder.mapping.api.get`, the name of the Inport block to configure, and name-value pair arguments specifying that the function set `StorageClass` to `ImportFromFile` and `HeaderFile` to `exInDataMem.h`.

```
setInport(cm,'In1','StorageClass','ImportFromFile','HeaderFile','exInDataMem.h');
```

Verify Header File Setting

Verify the updated header file setting for Inport block `In1` with a call to `getInport`. The function call specifies the code mappings object returned by `coder.mapping.api.get`, the name of the Inport block of interest, and the configuration information to return.

```
hfileIn1 = getInport(cm,'In1','StorageClass')
```

```
hfileIn1 =
'ImportFromFile'
```

Override Default Function Name Setting for Step Function

Previously, you set the default function customization template for execution functions to `functionFastMem`, which applies the naming rule `exFast_$N` to the three step functions generated for the model. The requirements specify that the name for function `Periodic:D2`, which has a sample time of 1 second, be named `exFast_1sec`.

Set the function name with a call to `setFunction`.The function call specifies the code mappings object returned by `coder.mapping.api.get`, the source of the function, `Periodic:D2`, and the name-value pair argument specifying that the function set the function name to `exFast_1sec`.

```
setFunction(cm,'Periodic:D2','FunctionName','exFast_1sec')
```

Verify Function Name Setting

Verify the updated function name setting for execution function `Periodic:D2` with a call to `getFunction`.The function call specifies the code mappings object returned by `coder.mapping.api.get`, the function of interest, and property `FunctionName`.

```
functionName = getFunction(cm,'Periodic:D2','FunctionName')
```

```
functionName =
'exFast_1sec'
```

See Also

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Choose Data Configuration Approach” on page 36-70
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models” on page 32-24

Configure Root-Level Inport Blocks for C Code Generation

To use code that you generate from a model, you call generated entry-point functions. The calling environment and generated functions exchange input and output data, for example, as global variables or function arguments. Root-level Inport blocks (inports) constitute the input data of the interface. To integrate and deploy the generated code into applications, you can customize how the code generator produces the interface code, including how input data is declared and handled. Customizations can:

- Minimize the modifications that you must make to existing code.
- Generate stable interfaces that do not change or minimally change when you make changes to your model.
- Generate code that exchanges data more efficiently (for example, by using pointers and pass-by-reference arguments for nonscalar data).

Enable and Trigger blocks (function-call inports) also load data into a system from outside the system. From a code generation perspective, these modeling elements are exported functions. For an exported function, the code generator produces an entry-point function, which you can configure. See “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

For code generation, examples show how to customize the inport interface for model `rtwdemo_configinterface` . You can configure code mappings by using the **Code Mappings Editor – C** or code mappings API (`coder.mapping.api.CodeMapping`).

Choose Customization Options for Inport Interface

By default, root-level inports in a model appear in generated code as fields of a global data structure named *`model`*`_ExtU`. Based on your code interface requirements, decide whether to customize generation of root-level inport data. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of inports in generated code for optimization purposes. If you configure customizations, decide:

- Whether to set up a default configuration

  If a model includes a significant number (for example, more than 10) of root-level inports, it is more efficient to configure the inports with a default setting, and then override that setting for special cases. If the model includes a few root-level inports that have unique source, naming, or placement requirements, consider configuring the inports individually.
- How to declare and handle root-level inport data in the generated interface

  - As separate global variables
  - To read input data from global variables defined in external code
  - For referenced model inports, as global variables (`void-void`)
  - As calls to access functions. Requires Embedded Coder
  - As entry-point function arguments. Requires Embedded Coder

  For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations include whether to:

- Name inports in the generated code by using Inport block labels that appear in the model or by using unique code identifiers.
- Include the `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
- Place data read into a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of interface requirements that are relevant to inports with corresponding storage classes and storage class properties, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

For example model `rtwdemo_configinterface`, inport interface requirements are:

- Use project type definition `DBL_FLOAT` defined in header file `exDblFloat.h`.
- Get data for inport `In1` from header file `exInDataMem.h`. The data is used for computing a value that is stored in memory, and then used in an if-else condition of code generated for the Switch block.
- Get data elements for inports `In2`, `In3`, and `In4` from header file `exInDataLut.h`. The data is used in lookup tables `Table1` and `Table2`.
- Variables representing the inports in the generated code must be named `input1`, `input2`, `input3`, and `input4`.

For this example, configure inports in `rtwdemo_configinterface` to meet these code generation requirements.

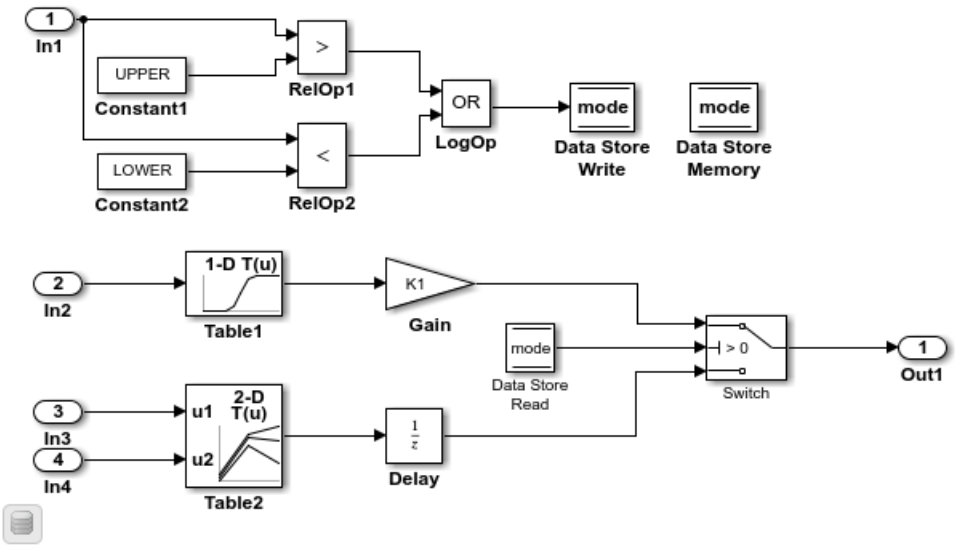
Configure Default Code Generation Settings for Root-Level Inports

A default code generation setting for root-level inports can reduce the effort of preparing a model for code generation, especially if a model has a significant number of inports. Choose configuration settings once, and the code generator applies those settings to the inports across the model. Simulink stores the default configuration as part of the model.

Consider configuring default code generation settings for model root-level inports if your model uses multiple root-level inports that do not have unique requirements or uses a shared Embedded Coder Dictionary.

This example shows how to use the **Code Mappings Editor – C** to configure default settings for root-level inports for model `rtwdemo_configinterface`. Specify an external header file that declares input data. Three of the four root inports read input from variables declared in header file `exInDataLut.h`. Set that header file as the default input source for inports.

1. Open example model `rtwdemo_configinterface`. Save a copy of the model to a writable location.

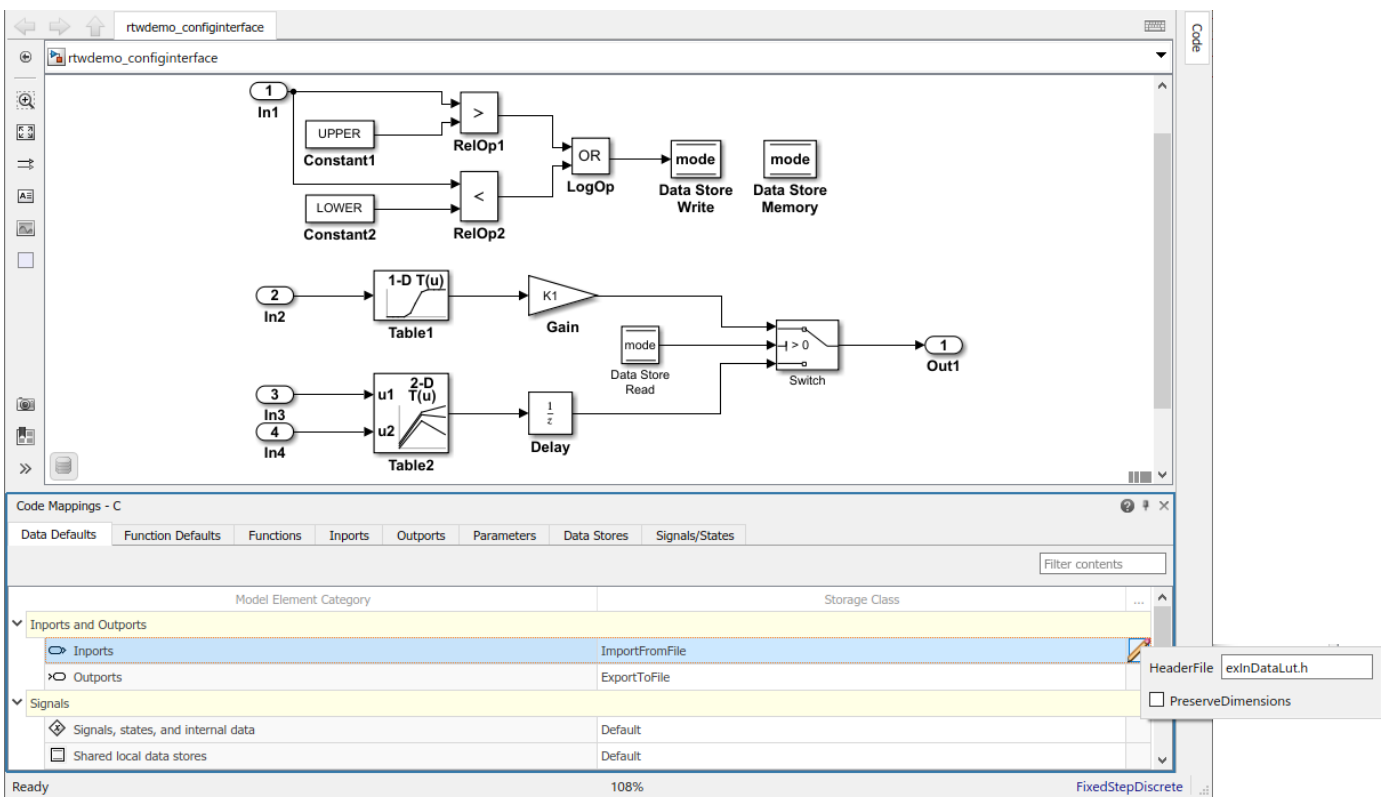


2 Open the **Embedded Coder** app.

3 In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.

4 In the Code Mappings editor, under **Inports and Outports**, select category **Inports**. Set the default storage class to `ImportFromFile`.

5 Click the icon, set **Header File** to `exInDataLut.h`.



6 Save the model.

Configure Code Generation Settings for Individual Root-Level Inports

You can configure individual root-level inports for code generation. For example, if a model has two root-level inports that have unique code generation requirements, configure the inports individually. Or, if you configure default settings for inports, configure individual inports to use the default settings or unique settings.

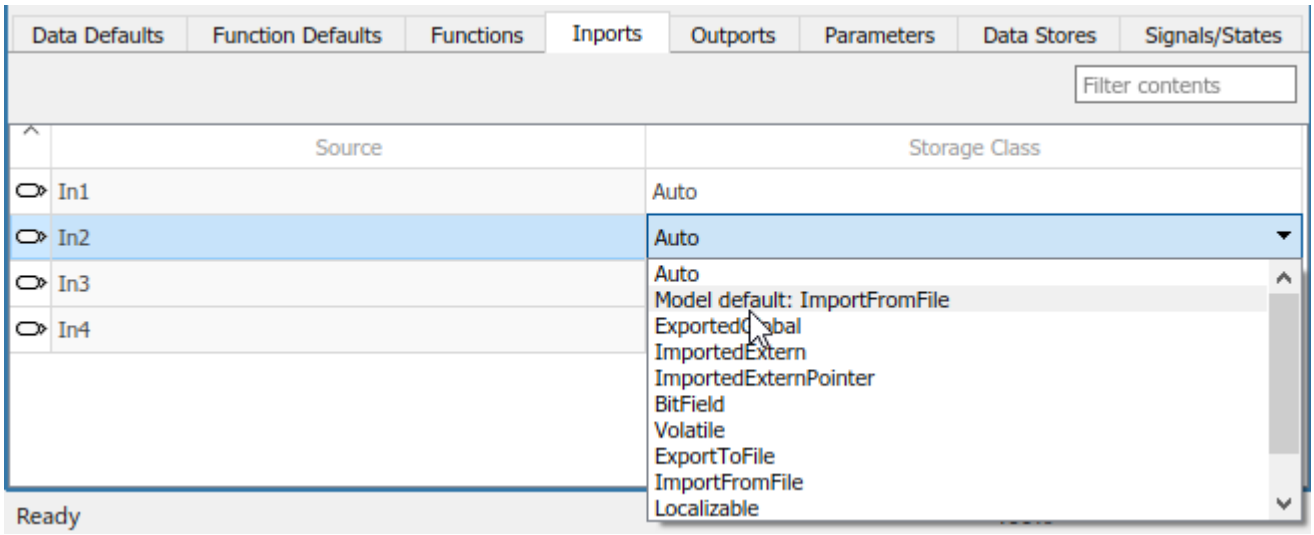
If your model meets at least one of these criteria, consider configuring code generation settings for root-level inports individually:

- Uses multiple inports that have unique requirements.
- Uses few inports.
- Has a default configuration for inports and you need to override the configuration for some specific inports.

This example shows how to use the Code Mappings editor to override the default code generation configuration for one of the inports in model `rtwdemo_configinterface`. In the preceding example, you set the default storage class for inports to `ImportFromFile`. The model requires that inport `In1` read input data from a global variable instead of a global variable pointer.

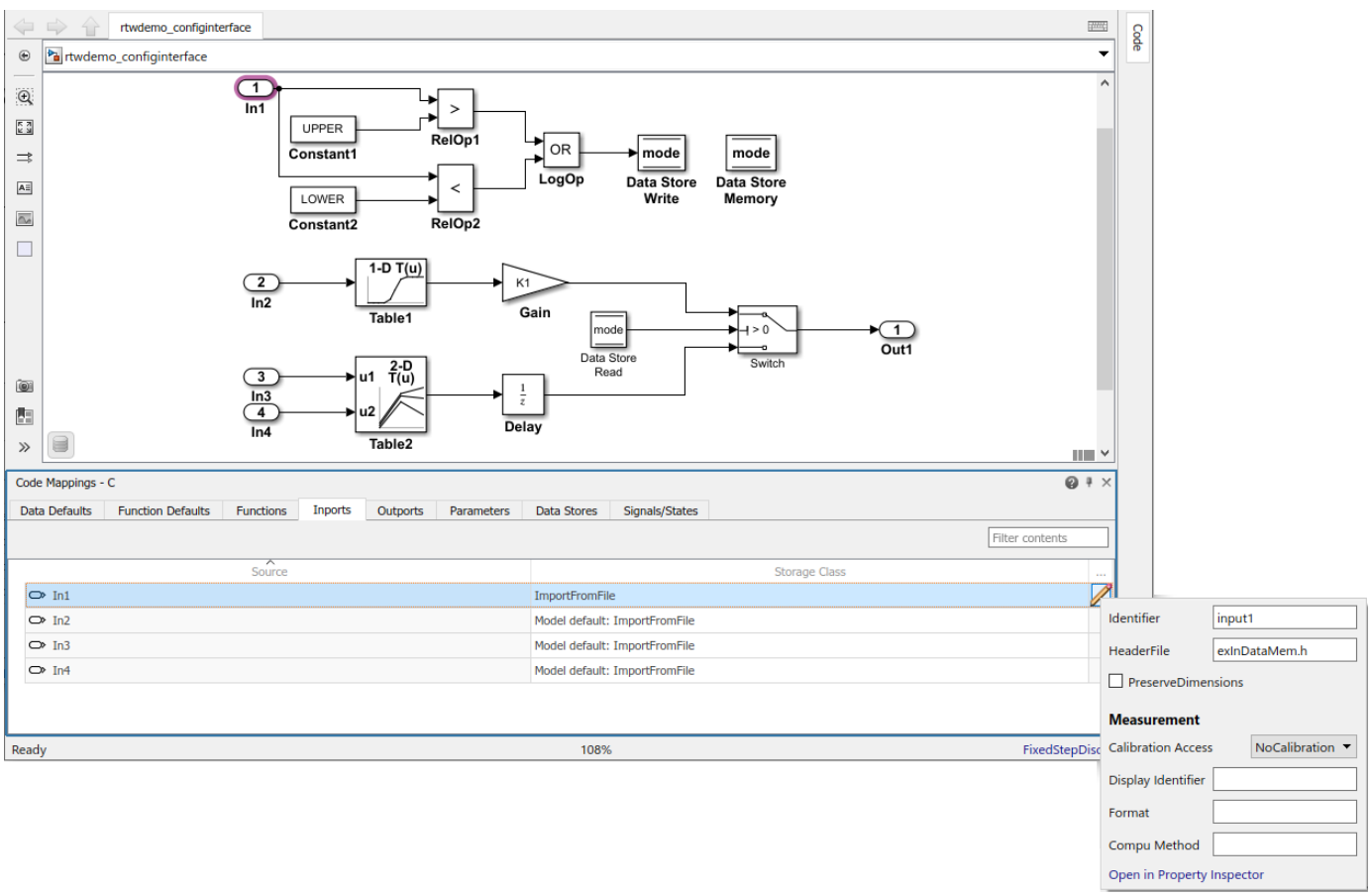
The example also shows how to configure identifiers that the code generator uses for naming the four inports in the generated code. You can specify code generation identifiers, for example for integration, without having to modify the model design.

1. If you have not already done so, complete the steps in “Configure Default Code Generation Settings for Root-Level Inports” on page 33-31.
2. In the Code Mappings editor, click the **Inports** tab. The editor lists the names of Inport blocks and bus elements that are in the model. If a port resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name. The storage class for each inport is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, the model default configuration specifies storage class `ImportedFromFile`.
   - To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
   - To override the default configuration, specify the storage class that meets the code generation requirements for that inport.
3. Configure the code generator to apply the default storage class setting to inports `In2`, `In3`, and `In4`. Select the rows for the three inports. Then, for one of the selected inports, set the storage class to `Model default: ImportFromFile`.



The storage class for the three selected inports changes to `Model default: ImportFromFile`.

4 For inport `In1`, override the default storage class setting. Set the storage class to `ImportFromFile`. Click the icon and set **Header File** to `exInDataMem.h`.



5. Configure code identifiers for the inports so that the interface argument names in the generated code match the interface names in the external header and definition files. In the Code Mappings editor, select the row for each inport. Click the icon and set the **Identifier** property to these values.

Inport	Code Identifier
In1	input1
In2	input2
In3	input3
In4	input4

6. Save the model.
7. Generate and view the code. For example, in `rtwdemo_configinterface.c` find where variables `input1`, `input2`, `input3`, and `input4` are used in the step entry-point function.

```
void rtwdemo_configinterface_step(void)
{
  mode = ((input1 > rtwdemo_configinterface_UPPER) || (input1 <
           rtwdemo_configinterface_LOWER));

  dout_Table1 = look1_binlc(input2, ((const MYTYPE *)&(Table1.BP[0])), ((
    const MYTYPE *)&(Table1.Table[0])), 10U);
.
.
.
  dout_Table2 = look2_binlc(input3, input4, ((const MYTYPE *)&(mp_Table2.BP1[0])),
    ((const MYTYPE *)&(mp_Table2.BP2[0])), ((const MYTYPE *)&(mp_Table2.Table[0])),
    ((const uint32_T *)&(rtwdemo_configi_Table2_maxIndex[0])), 3U);

  dstate_X = dout_Table2;
}
```

Configure Code Generation Settings for Root-Level Inports Programmatically

To automate configuration of root-level inports for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate data configuration.

This example shows how to use the programming interface to configure default settings for root-level inports for model `rtwdemo_configinterface`. Specify an external header file that declares input data. Three of the four root-level inports read input from variables declared in header file `exInDataLut.h`. You set that file as the default header file. Then, you override the default code generation configuration for `In1`, which requires that input data be read from header file `exInDataMem.h`.

In addition, you configure code identifiers for the inports so that global variable names in the generated code match the variable names defined in an external file.

1. Open the example model.

```
open_system('rtwdemo_configinterface')
```

2. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the data and function code generation configuration for model `rtwdemo_configinterface`.

```
cm = coder.mapping.api.get('rtwdemo_configinterface');
```

3. Configure default settings for inports by calling function `setDataDefault`. For the arguments, specify these values:

   - The object returned by `coder.mapping.api.get`
   - `Inports` for the default category
   - Property name `StorageClass` with property value `ImportFromFile`
   - Property name `HeaderFile` with property value `exInDataLut.h`, the name of the external file that contains the input data

   ```
   setDataDefault(cm,'Inports','StorageClass','ImportFromFile','HeaderFile','exInDataLut.h')
   ```

4. Verify your default configuration for inports. Issue two calls to `getDataDefault` that specify the object returned by `coder.mapping.api.get` and category `Inports`. In the first call, specify the third argument as property `StorageClass`. In the second call, for the third argument, specify `HeaderFile`.

   ```
   getDataDefault(cm,'Inports','StorageClass')

   ans =

       'ImportFromFile'

   getDataDefault(cm,'Inports','HeaderFile')

   ans =

       'exInDataLut.h'
   ```

5. Override the default inport configuration for inport `In1`. The requirements for that inport differ from requirements of the other three inports.

   By default, Simulink sets the storage class for individual inports to `Auto`. When the storage class is `Auto`, the code generator:

   - Determines whether to eliminate the data from the generated code for optimization purposes.
   - If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

   To control the storage class and header file configuration for an inport, configure settings for the inport with function `setInport`. Configure code generated for inport `In1` to read input data from global variables defined in external header file `exInDataMem.h`, as specified in the requirements. Issue two calls to function `setInport` that specify the object returned by `coder.mapping.api.get` and Inport block named `In1`. In the first call, also specify property name `StorageClass` and property value `ImportFromFile`. In the second call, specify property name `HeaderFile` and property value `exInDataMem.h`.

   ```
   setInport(cm,'In1','StorageClass','ImportFromFile')
   setInport(cm,'In1','HeaderFile','exInDataMem.h')
   ```

6. Verify your configuration for inport `In1` by calling function `getInport`.

   ```
   getInport(cm,'In1','StorageClass')

   ans =

       'ImportFromFile'
   ```

```
getInport(cm,'In1','HeaderFile')

ans =

    'exInDataMem.h'
```

7 Configure the code generator to apply your default inport settings to inports `In2`, `In3`, and `In4`. For each inport, call function `setInport`. Specify the object returned by `coder.mapping.api.get`, the Inport block name, property name `StorageClass`, and property value `Model default`.

```
setInport(cm,'In2','StorageClass','Model default')
setInport(cm,'In3','StorageClass','Model default')
setInport(cm,'In4','StorageClass','Model default')
```

8 Verify your configuration changes for inports `In2`, `In3`, and `In4` with calls to function `getInport`.

```
getInport(cm,'In2','StorageClass')

ans =

    'Model default'

getInport(cm,'In3','StorageClass')

ans =

    'Model default'


getInport(cm,'In4','StorageClass')

ans =

    'Model default'
```

9 Configure code identifiers for the inports so that the global variables that represent inports in the generated code match the variable names in the external definition file. For each inport, call function `setInport`. Specify the object returned by `coder.mapping.api.get`, the Inport block name, property name `Identifier`, and one of these property values.

Inport	Code Identifier
In1	input1
In2	input2
In3	input3
In4	input4

```
setInport(cm,'In1','Identifier','input1');
setInport(cm,'In2','Identifier','input2');
setInport(cm,'In3','Identifier','input3');
setInport(cm,'In4','Identifier','input4');
```

10 Verify your configuration changes for the inports by using calls to function `getInport`.

```
getInport(cm,'In1','Identifier')
```

```
ans =

    'input1'

getInport(cm,'In2','Identifier')

ans =

    'input2'

getInport(cm,'In3','Identifier')

ans =

    'input3'


getInport(cm,'In4','Identifier')

ans =

    'input4'
```

11 Save the model.

12 Generate and view the code. For example, in `rtwdemo_configinterface.c` find where variables `input1`, `input2`, `input3`, and `input4` are used in the step entry-point function.

```
void rtwdemo_configinterface_step(void)
{
  mode = ((input1 > rtwdemo_configinterface_UPPER) || (input1 <
           rtwdemo_configinterface_LOWER));

  dout_Table1 = look1_binlc(input2, ((const MYTYPE *)&(Table1.BP[0])), ((
    const MYTYPE *)&(Table1.Table[0])), 10U);
.
.
.
  dout_Table2 = look2_binlc(input3, input4, ((const MYTYPE *)&(mp_Table2.BP1[0])),
    ((const MYTYPE *)&(mp_Table2.BP2[0])), ((const MYTYPE *)&(mp_Table2.Table[0])),
    ((const uint32_T *)&(rtwdemo_configi_Table2_maxIndex[0])), 3U);

  dstate_X = dout_Table2;
}
```

Choose Storage Class and Storage Class Properties for Root-Level Inports

Depending on your code generation requirements, choose from these storage classes to configure code generation for root-level inports.

Requirements	Storage Class
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)

Requirements	Storage Class
Prevent optimizations from eliminating storage for a data element and use the default mapping for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mappings only)
Generate a global variable definition and declaration that have the `volatile` type qualifier.	`Volatile` (See “Const, Volatile, and ConstVolatile” on page 36-79)
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
Read input data for the inport from a specified external header file.	“ImportFromFile” on page 36-82
Generate variables that are local to functions.	“Localizable” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)
Generate a global variable that enables buffer reuse.	“Reusable” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual inport, use the **Identifier** storage class property to configure a name for the variable representing the inport in the generated code. If you leave the **Identifier** property blank, the code generator uses the name of the signal attached to the Inport block. If the signal label is empty, the code generator uses the name of the Inport block.

With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the inport and external code	`ExportToFile` and `Volatile`
`GetFunction`	Inport appears in the generated code as a call to a specified `get` function	`GetSet`

Property	Description	Storage Classes
`HeaderFile`	Source header file that contains declarations for global data that is read by the inport and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default inport configuration only)	Memory section that contains data read by the inport	`Default`
`Owner`	Code generator places the definition for inports in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See "Control Placement of Global Data Definitions and Declarations in Generated Files" on page 37-2.	`ExportToFile` and `Volatile`
`PreserveDimensions`	Code generator preserves dimensions of inport data that is represented in generated code as a multidimensional array. You must set model configuration parameter **Array layout** to `Row-major`. See "Preserve Dimensions of Multidimensional Arrays in Generated Code" on page 47-2.	`ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`
`SetFunction`	Inport appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for a structure in the generated code for inport.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- "C Code Generation Configuration for Model Interface Elements" on page 33-2
- "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
- "Configure Default C Code Generation for Categories of Data Elements and Functions" on page 33-15
- "Control Data and Function Interface in Generated Code" on page 36-44
- "Configure Model Data Elements for ASAP2 File Generation" on page 66-2

Configure Root-Level Outport Blocks for C Code Generation

To use code that you generate from a model, you call generated entry-point functions. The calling environment and generated functions exchange input and output data, for example, as global variables or function arguments. Root-level Outport blocks (outports) constitute the output data of the interface. To integrate and deploy the generated code into larger applications, you can customize how the code generator produces the interface code, including how output data is declared and handled. Customizations can:

- Minimize the modifications that you make to existing code.
- Generate stable interfaces that do not change or minimally change when you make changes to your model.
- Generate code that exchanges data more efficiently (for example, by using pointers and pass-by-reference arguments for nonscalar data).

For code generation, examples show how to customize the outport interface for the model `rtwdemo_configinterface`. You can configure code mappings by using the **Code Mappings Editor – C** or code mappings API (`coder.mapping.api.CodeMapping`).

Choose Customization Options for Outport Interface

By default, root-level outports in a model appear in generated code as fields of a global data structure named *`model`*`_ExtY`. Based on your code interface requirements, decide whether to customize generation of outport data. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of outports in generated code for optimization purposes. If you configure customizations, decide:

- Whether to set up a default configuration

  If a model includes a significant number (for example, more than 10) of root-level outports, it is more efficient to configure the outports with a default setting, and override that setting for special cases. If the model includes a few outports that have unique source, naming, or placement requirements, consider configuring the outports individually.
- How to declare and handle outport data in the generated interface

  - As separate global variables
  - To write output data to global variables defined in external code
  - For referenced model outports, as global variables (`void-void`)
  - As calls to access functions. Requires Embedded Coder
  - As entry-point function arguments. Requires Embedded Coder

  For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations include whether to:

- Name outports in the generated code by using Outport block labels that appear in the model or by using unique code identifiers.
- Include the `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.

- Write data from a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of lists interface requirements that are relevant to outports with corresponding storage classes and storage class properties, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

Outport interface requirements for example model `rtwdemo_configinterface` are:

- Use project type definition `MYTYPE` defined in `rtwdemo_configinterface_data.m`.
- Declare a variable for outport `Out1` in generated external header file `exOutSys.h` and define the variable in source definition file `exOutSys.c`. The data written to `Out1` is the output of code generated for the Switch block.
- The variable representing the outport in the generated code must be named `output1`. That name is the variable name that the external code interface uses to access the output data.

For this example, configure the outport in `rtwdemo_configinterface` to meet these code generation requirements.

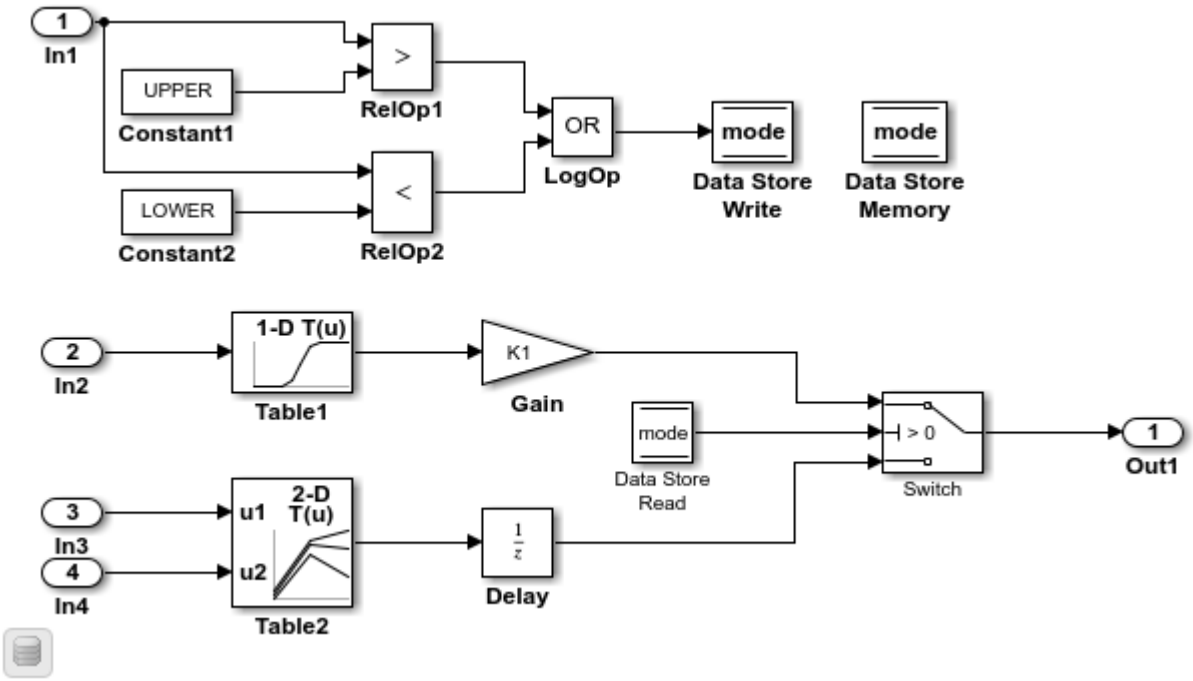
Configure Default Code Generation Settings for Root-Level Outports

A default code generation setting for root-level outports can reduce the effort of preparing a model for code generation, especially if a model has a significant number of outports. Choose configuration settings once, and the code generator applies those settings to outports across the model. Simulink stores the default configuration as part of the model.

Consider configuring default code generation settings for model outports if your model uses multiple root-level outports that do not have unique requirements or uses a shared Embedded Coder Dictionary.

The example model `rtwdemo_configinterface` includes one root-level outport. This example shows how to use the **Code Mappings Editor – C** to configure default settings for root-level outports. Specify that the code generator declare and define variables to represent root outports in generated external header and definition files. Name the generated default external header file `exOutSys.h` and the generated definition file `exOutSys.c`.

1. Open example model `rtwdemo_configinterface`. Save a copy of the model to a writable location.

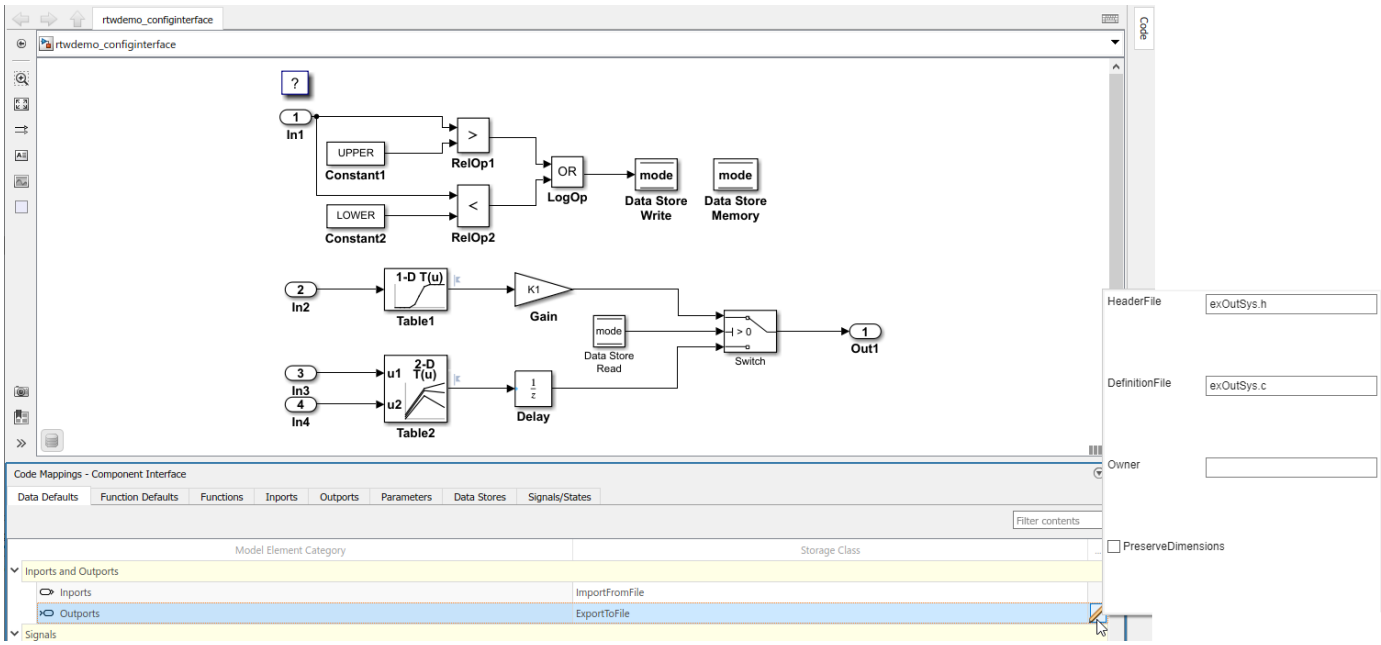


2 Open the **Embedded Coder** app.

3 In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.

4 In the Code Mappings editor, under **Inports and Outports**, select category **Outports**. Set the default storage class to `ExportToFile`.

5 Click the icon and set **Header File** to `exOutSys.h` and **Definition File** to `exOutSys.c`.



6 Save the model.

Configure Code Generation Settings for Individual Root-Level Outports

You can configure individual root-level outports for code generation. For example, if a model has two root-level outports that have unique code generation requirements, configure the outports individually. Or, if you configure default settings for outports, configure individual outports to use the default settings or unique settings.

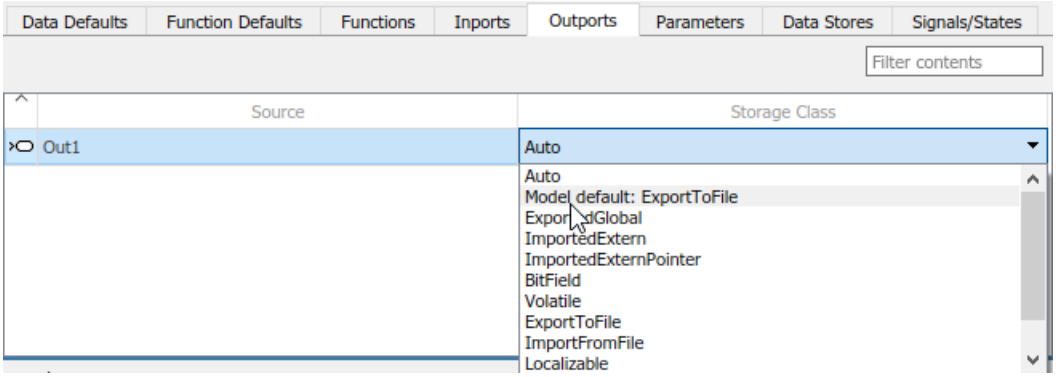
If your model meets at least one of these criteria, consider configuring code generation settings for outports individually:

- Uses multiple outports that have unique requirements.
- Uses few outports.
- Has a default configuration for outports and you need to override the configuration for some specific outports.

This example shows how to use the Code Mappings editor to apply the default code generation configuration for outports in model `rtwdemo_configinterface`. In the preceding example, you set the default storage class for outports to `ExportToFile`. A requirement specifies that outport `Out1` write output data to a global variable declared and defined in generated header and definition files accessible to external code.

The example also shows how to configure an identifier that the code generator uses for naming the outport in the generated code. You can specify code generation identifiers, for example for integration, without modifying the model design.

1. If you have not already done so, complete the steps in “Configure Default Code Generation Settings for Root-Level Outports” on page 33-42.
2. In the Code Mappings editor, click the **Outports** tab. The editor lists the names of Outport blocks and bus elements that are in the model. If a port resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name. The storage class for the outport in the example model is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, the model default configuration specifies storage class `ExportToFile`.
   - To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
   - To override the default configuration, specify the storage class that meets the code generation requirements for that outport.
3. Configure the code generator to apply the default storage class setting to outport `Out1`. Select the row for the outport. Set the storage class to `Model default: ExportToFile`.



The storage class for the selected outport changes to `Model default: ExportToFile`.

4. Configure the code identifier for the outport so that the interface argument name in the generated code matches the interface name used by the external code. In the Code Mappings editor, select the row for the outport. Click the icon and set the **Identifier** property to `output`.
5. Save the model.
6. Generate and view the code. For example, in the file `rtwdemo_configinterface.c`, find where variable `output` is used in the step entry-point function.

```
if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
```

Configure Code Generation Settings for Root-Level Outports Programmatically

To automate configuration of root-level outports for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate data configuration.

This example shows how to use the programming interface to configure default settings for root-level outports for model `rtwdemo_configinterface`.

Configure code an identifier for the outport so that global variable name in the generated code matches the variable name in the external code interface.

1. Open the example model.

```
open_system('rtwdemo_configinterface')
```

2. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the code generation configuration for data elements and functions in model `rtwdemo_configinterface`.

```
cm = coder.mapping.api.get('rtwdemo_configinterface');
```

3. Configure default settings for outports by calling function `setDataDefault`. For the arguments, specify these values.

- The object returned by `coder.mapping.api.get`
- `Outports` for the default category
- Property name `StorageClass` that has property value `ExportToFile`
- Property name `HeaderFile` that has property value `exOutSys.h` and property name `DdefinitionFile` that has property value `exOutSys.c`, the names for the generated files that declare and define the output data

```
setDataDefault(cm,'Outports','StorageClass','ExportToFile','HeaderFile','exOutSys.h',...
'DefinitionFile','exOutSys.c')
```

4 Verify your default configuration for outports. Issue three calls to `getDataDefault` that specify the object returned by `coder.mapping.api.get` and category `Outports`.

- In the first call, specify the third argument as property `StorageClass`.
- In the second call, for the third argument, specify `HeaderFile`.
- In the third call, for the third argument, specify `DefinitionFile`.

```
getDataDefault(cm,'Outports','StorageClass')

ans =

    'ExportToFile'

getDataDefault(cm,'Outports','HeaderFile')

ans =

    'exOutSys.h'

getDataDefault(cm,'Outports','DefinitionFile')

ans =

    'exOutSys.c'
```

5 Apply the default outport configuration for outport `Out1`.

By default, Simulink sets the storage class for individual outports to `Auto`. When the storage class is `Auto`, the code generator:

- Determines whether to eliminate the data from the generated code for optimization purposes.
- If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

To control the configuration for an outport, call function `setOutport`.

Issue a call to `setOutport` that specifies:

- Object returned by `coder.mapping.api.get`
- Outport block name `Out1`
- Default storage class previously set for the outport by using property `StorageClass` and property value `Model default`
- Property `Identifier` and property value `output`

```
setOutport(cm,'Out1','StorageClass','Model default','Identifier','output');
```

6. Verify your configuration changes by calling function `getOutport`. Specify the object returned by `coder.mapping.api.get`, name of the outport block, and property `StorageClass` or `Identifier`.

```
getOutport(cm,'Out1','StorageClass')

ans =

    'Model default'

getOutport(cm,'Out1','Identifier')

ans =

    'output'
```

7. Save the model.
8. Generate and view the code. For example, in the file `rtwdemo_configinterface.c`, find where variable `output` is used in the step entry-point function.

```
if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
```

Choose Storage Class and Storage Class Properties for Root-Level Outports

Depending on your code generation requirements, choose from these storage classes to configure code generation for root-level outports.

Requirements	Storage Class
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)
Prevent optimizations from eliminating storage for a data element and use the default mapping for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mappings only)
Generate a global variable definition and declaration that has the `volatile` type qualifier.	`Volatile` (See “Const, Volatile, and ConstVolatile” on page 36-79)
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81

Requirements	Storage Class
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
Generate code that reads from and writes to a global variable defined by your external header file.	“ImportFromFile” on page 36-82
Generate variables that are local to functions.	“Localizable” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual outport, use the **Identifier** storage class property to configure a name for the variable representing the outport in the generated code. If you leave the **Identifier** property blank, the code generator uses the **Signal name** parameter of the Outport block. If this parameter is empty, the code generator uses the name of the Outport block.

With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the outport and external code	`ExportToFile` and `Volatile`
`GetFunction`	Outport appears in the generated code as a call to a specified `get` function	`GetSet`
`HeaderFile`	Name of generated source header file that contains declarations for global data, which is written by the outport and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default outport configuration only)	Memory section that contains data written by the outport	`Default`
`Owner`	Code generator places the definition for outports in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.	`ExportToFile` and `Volatile`

Property	Description	Storage Classes
`PreserveDimensions`	Code generator preserves dimensions of outport data that is represented in generated code as a multidimensional array. You must set model configuration parameter **Array layout** to `Row-major`. See "Preserve Dimensions of Multidimensional Arrays in Generated Code" on page 47-2.	`ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`
`SetFunction`	Outport appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for a structure in the generated code for outport.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- "C Code Generation Configuration for Model Interface Elements" on page 33-2
- "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
- "Configure Default C Code Generation for Categories of Data Elements and Functions" on page 33-15
- "Control Data and Function Interface in Generated Code" on page 36-44
- "Configure Model Data Elements for ASAP2 File Generation" on page 66-2

Configure Signal Data for C Code Generation

Signal data consists of variables that store intermediate results in global memory as generated algorithmic code computes values of block output signals from block input signals, excluding signals that connect to root Inport and Outport blocks. Signal data is not signal lines in a model diagram. For example, in these cases, a signal line has no data representation in generated code:

- Feeds a function-call port
- Exits a message block, such as the Send block
- Crosses subsystem boundaries with no computation blocks in between

When you configure a model for code generation, you choose the block input and output signals to make accessible in the generated code. For example, configure signal data to:

- Make the data accessible for interaction and monitoring while the generated code executes.
- Minimize the amount of data that is stored in memory.
- Control where the code generator places signal data in memory.
- Promote signal data to the model interface so that other components and systems can access that data.
- Improve readability and traceability of the generated code.

The code generator checks for signal line configuration consistency. For example, the code generator verifies that the configuration of a signal line feeding into a subsystem matches the configuration of the signal line that exits the corresponding subsystem inport.

For code generation, examples show how to configure block signals for the model `rtwdemo_configinterface`. You can configure code mappings by using the **Code Mappings Editor – C** or code mappings programming interface (`coder.mapping.api.CodeMapping`).

Choose Code Configuration Options for Signals

Based on your code generation requirements, decide which block signals to represent in the generated code and how to represent the signal data. By default, signals in a model appear in generated code as fields of a global data structure named *`model`*`_B`. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of signals in generated code for optimization purposes. If you configure customizations, decide:

- Which signals to make accessible in the generated code

  You must add signals that you want to make accessible in the generated code to the model code mappings.
- Whether to set up a default configuration

  If you need to gain access to a significant number (for example, more than 10) of signals , it is more efficient to configure the signals with default settings and then override those settings for special cases. If you need to gain access to a few signals that have unique source, naming, or placement requirements, consider configuring the signals individually.
- How to declare and handle signal data in the generated code

  - As separate global variables

- To read input data from global variables defined in external code
- As calls to access functions. Requires Embedded Coder

For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations include whether to:

- Name signals in the generated code by using signal labels that appear in the model or by using unique code identifiers.
- Include the `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
- Place signal data into a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of interface requirements that are relevant to signals with corresponding storage classes and storage class properties, see “Choose Storage Class and Storage Class Properties for Data Stores” on page 33-89.

Signal requirements for example model `rtwdemo_configinterface` are:

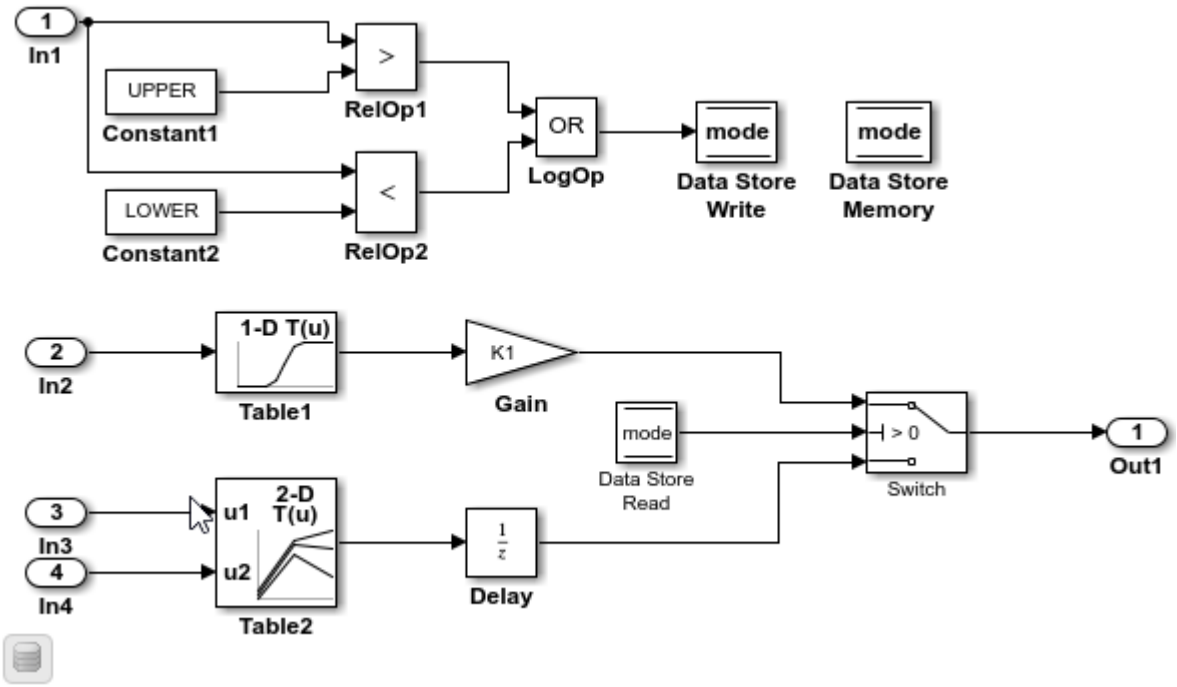
- Configure signals that feed into and exit from the Switch block for code generation. Retain that signal data for monitoring while the generated code executes.
- Represent signals as separate global variables.
- Apply prefix `dout_` to names of variables that represent signals.

Set the default representation of signals in the generated code as global variables that have the `static` type qualifier. Then, you configure the output signals of the two lookup table blocks to use the default storage class and unique code identifiers that include the required prefix `dout_`.

Add Signals to Model Code Mappings

Before you can configure a signal for code generation, add the signal to the model code mappings. Add the output signals for the two lookup table blocks to the model code mappings.

1. Open example model `rtwdemo_configinterface`. Save a copy of the model to a writable location.



2 Open the **Embedded Coder** app.

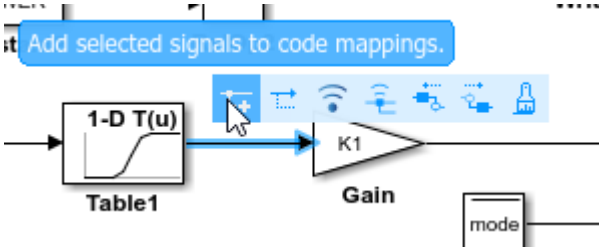
3 In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

4 In the Code Mappings editor, click the **Signals/States** tab. No signals are listed.

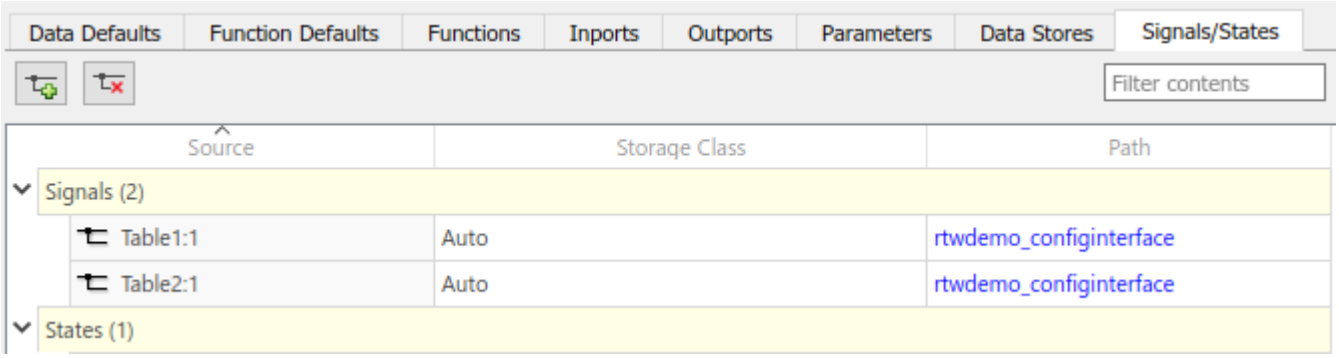
5 Add signals to the code mappings. For the output signal of lookup table blocks `Table1` and `Table2`:

   a In the model, select the signal.

   b Pause on the ellipsis that appears above or below the signal line to open the action bar. Click the **Add Signal** button.



In the Code Mappings editor, the **Signals** node expands and lists the two signals that you added.



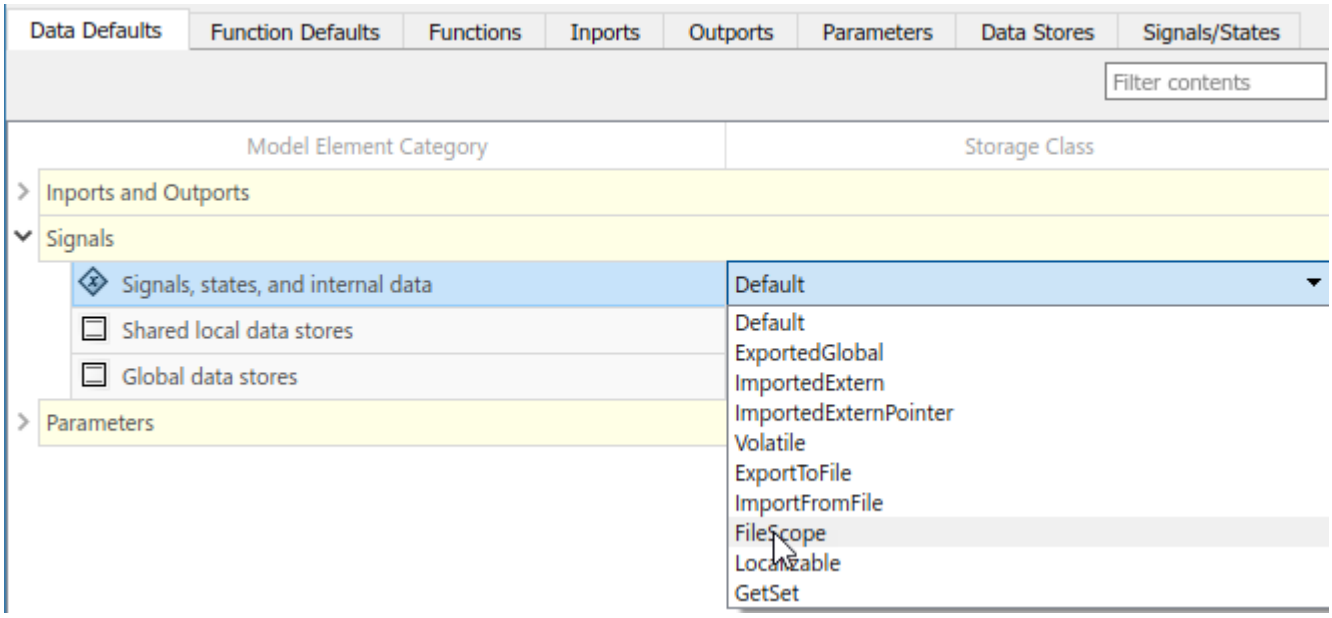
Configure Default Code Generation Settings for Signals

A default code generation setting for signals can reduce the effort of preparing a model for code generation, especially if a model has a significant number of signals that you want to monitor while the generated code executes. Choose configuration settings once, and the code generator applies those settings to signals across the model. Simulink stores the default configuration as part of the model.

Consider configuring default code generation settings for model signals if your model uses multiple signals that do not have unique requirements or uses a shared Embedded Coder Dictionary.

This example shows how to use the **Code Mappings Editor – C** to set the default storage class for signals in the model `rtwdemo_configinterface` to `FileScope`. With that storage class setting, the code generator represents signal data in the generated code as global variables that have the `static` type qualifier.

1. If you have not already done so, add signals to the model code mappings as described in “Add Signals to Model Code Mappings” on page 33-51.
2. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
3. In the Code Mappings editor, on the **Data Defaults** tab, under **Signals**, select category **Signals, states, and internal data**. Set the default storage class to `FileScope`.



4 Save the model.

Configure Code Generation Settings for Individual Signals

You can configure individual signals for code generation. For example, if a model has two signals that have unique code generation requirements, configure the signals individually. Or, if you configure default settings for signals, you can override those settings for specific signals.

If your model meets at least one of these criteria, consider configuring code generation settings for signals individually:

- Uses multiple signals that have unique source, naming, or placement requirements.
- Uses a few signals.
- Has a default configuration for signals and you need to override the configuration for some specific signals.

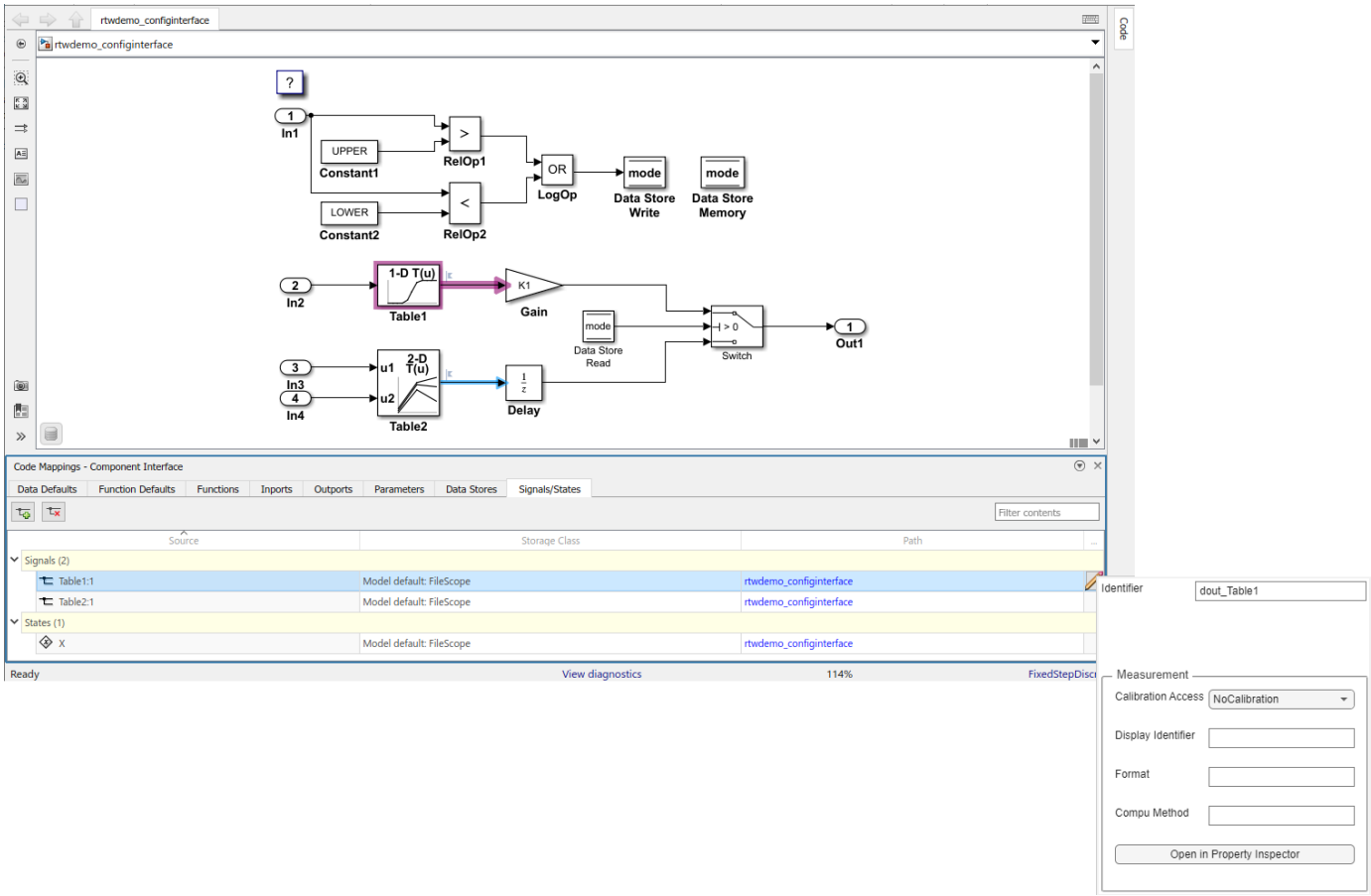
This example shows how to use the Code Mappings editor to apply your default storage class setting to the output signals of lookup table blocks `Table1` and `Table2` in model `rtwdemo_configinterface`. The example also shows how configure code identifiers for those output signals. You can specify code generation identifiers, for example for integration, without modifying the model design.

1 If you have not already done so, complete the steps in “Configure Default Code Generation Settings for Signals” on page 33-53.

2 In the Code Mappings editor, click the **Signals/States** tab. Expand **Signals**. The editor lists the names or block port identifiers of signals that you added to the code mappings. If a signal resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name or port identifier. The storage class for each signal is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, the model default configuration specifies storage class `FileScope`.

- To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
- To override the default configuration, specify the storage class that meets the code generation requirements for that signal.

3. In the Code Mappings editor, select the output signals for blocks `Table1` and `Table2`. Set the storage class to `Model default: FileScope`.
4. Configure the code identifier for the output signals for the two lookup table blocks with names that include the prefix `dout_`. In the Code Mappings editor, select signal `Table1:1`. Click the icon and set the **Identifier** property to `dout_Table1`. For signal `Table2:1`, set **Identifier** to `dout_Table2`.



5. Save the model.
6. Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definitions for the lookup table data.

```
static MYTYPE dout_Table1;
static MYTYPE dout_Table2;
```

Find where lookup table data is used in the step entry-point function.

```
.
.
.
  dout_Table1 = look1_binlc(input2, ((const MYTYPE *)&(mp_Table1.BP[0])), ((
```

```
    const MYTYPE *)&(mp_Table1.Table[0])), 10U);

  if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }

  dout_Table2 = look2_binlc(input3, input4, ((const MYTYPE *)&(mp_Table2.BP1[0])),
    ((const MYTYPE *)&(mp_Table2.BP2[0])), ((const MYTYPE *)&(mp_Table2.Table[0])),
    ((const uint32_T *)&(rtwdemo_configi_Table2_maxIndex[0])), 3U);

  dstate_X = dout_Table2;
}
```

Remove Signals from Model Code Mappings

1. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.
2. In the Code Mappings editor, click the **Signals/States** tab.
3. For each signal that you want to remove:
   a. In the model, select the signal.
   b. Pause on the ellipsis that appears above or below the signal line to open the action bar. Click the **Remove Signal** button.

Configure Code Generation Settings for Signals Programmatically

To automate configuration of signals for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate data configuration.

This example shows how to use the programming interface to configure signals for model `rtwdemo_configinterface`. Set the default representation of signals in the generated code as global variables that have the `static` type qualifier. Then, configure the output signals of the two lookup table blocks to use the default storage class and unique code identifiers that include the required prefix `dout_`.

1. Open the example model.

   ```
   open_system('rtwdemo_configinterface')
   ```

2. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the code generation configuration for data and functions for model `rtwdemo_configinterface`.

   ```
   cm = coder.mapping.api.get('rtwdemo_configinterface');
   ```

3. Configure default settings for signals by calling function `setDataDefault`. For the arguments, specify these values:
   - The object returned by `coder.mapping.api.get`
   - `InternalData` for the default category
   - Property name `StorageClass` with property value `FileScope`

   ```
   setDataDefault(cm,'InternalData','StorageClass','FileScope');
   ```

4. Verify your default configuration for signals. Issue a call to `getDataDefault` that specifies the object returned by `coder.mapping.api.get` and category `InternalData`. Specify the third argument as property `StorageClass`.

```
getDataDefault(cm,'InternalData','StorageClass')

ans =

    'FileScope'
```

5 Get handles to the output ports for lookup table blocks `Table1` and `Table2`. You use the port handles to add the signal data to the model code mappings.

```
lut1D_ports = get_param('rtwdemo_configinterface/Table1','PortHandles');
lut2D_ports = get_param('rtwdemo_configinterface/Table2','PortHandles');

lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;
```

6 Add the output signals of the lookup table blocks to the model code mappings.

```
addSignal(cm,[lut1D_outPort,lut2D_outPort]);
```

7 Apply the default configuration for signals to the output signal of the lookup table blocks.

By default, Simulink sets the storage class for individual signals to `Auto`. The code generator:

- Determines whether to eliminate the data from the generated code for optimization purposes.
- If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

To control the configuration for a signal, call function `setSignal`.

For each signal, issue a call to `setSignal` that specifies:

- Object returned by `coder.mapping.api.get`
- Port handle for the signal, `lut1D_outport` or `lut2D_outport`.
- Default storage class previously set for signals by using property `StorageClass` and property value `Model default`.
- Property `Identifier` and property value `dout_Table1` or `dout_Table2`

```
setSignal(cm,lut1D_outPort,'StorageClass','Model default','Identifier','dout_Table1');
setSignal(cm,lut2D_outPort,'StorageClass','Model default','Identifier','dout_Table2');
```

8 Verify your configuration settings by calling function `getsignal`. Specify the object returned by `coder.mapping.api.get`, the port handle for the signal (`lut1D_outport` or `lut2D_outport`), and property `StorageClass` or `Identifier`.

```
getSignal(cm,lut1D_outPort,'StorageClass')

ans =

    'Model default'

getSignal(cm,lut1D_outPort,'Identifier')

ans =

    'dout_Table1'

getSignal(cm,lut2D_outPort,'StorageClass')

ans =
```

```
    'Model default'

getSignal(cm,lut2D_outPort,'Identifier')

ans =

    'dout_Table2'
```

9 Save the model.

10 Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definitions for the lookup table data.

```
static MYTYPE dout_Table1;
static MYTYPE dout_Table2;
```

Find where lookup table data is used in the step entry-point function.

```
.
.
.
  dout_Table1 = look1_binlc(input2, ((const MYTYPE *)&(mp_Table1.BP[0])), ((
    const MYTYPE *)&(mp_Table1.Table[0])), 10U);

  if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }

  dout_Table2 = look2_binlc(input3, input4, ((const MYTYPE *)&(mp_Table2.BP1[0])),
    ((const MYTYPE *)&(mp_Table2.BP2[0])), ((const MYTYPE *)&(mp_Table2.Table[0])),
    ((const uint32_T *)&(rtwdemo_configi_Table2_maxIndex[0])), 3U);

  dstate_X = dout_Table2;
}
```

Choose Storage Class and Storage Class Properties for Signals

Depending on your code generation requirements, choose from these storage classes to configure code generation for block signals.

Requirements	Storage Class to Choose
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)
Prevent optimizations from eliminating storage for a data element and use the default storage class for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mapping only)
Generate a global variable definition and declaration that have the `volatile` type qualifier.	`Volatile` (See “Const, Volatile, and ConstVolatile” on page 36-79)
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78

Requirements	Storage Class to Choose
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate a global variable definition and declaration that have the `static` type qualifier.	“FileScope” on page 36-81
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
Generate code that reads from and writes to a global variable defined by your external header file.	“ImportFromFile” on page 36-82
Generate variables that are local to functions.	“Localizable” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)
Generate a global variable that enables buffer reuse.	“Reusable” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual signal, use the **Identifier** storage class property to configure a name for the variable representing the signal in the generated code. If you leave the **Identifier** property blank, the code generator uses the signal label. If the signal label is empty, the code generator uses the name of the block that outputs the signal.

With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Relevant Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the signal and external code	`ExportToFile` and `Volatile`
`GetFunction`	signal appears in the generated code as a call to a specified `get` function	`GetSet`
`HeaderFile`	Source header file that contains declarations for global data, which is read by the signal and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default signal configuration only)	Memory section that contains data read by the signal	`Default`

Property	Description	Relevant Storage Classes
`Owner`	Code generator places the definition for signals in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.	`ExportToFile` and `Volatile`
`PreserveDimensions`	Code generator preserves dimensions of signal data that is represented in generated code as a multidimensional array. You must set model configuration parameter **Array layout** to `Row-major`. See “Preserve Dimensions of Multidimensional Arrays in Generated Code” on page 47-2.	`ExportToFile`, `FileScope`, `GetSet`, `ImportFromFile`, `Localizable` and `Volatile`
`SetFunction`	Signal appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for structure in the generated code for signal.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Configure Model Data Elements for ASAP2 File Generation” on page 66-2

Configure Parameters for C Code Generation

When configuring a model for code generation, you can identify and configure parameters for tunability, for example for calibration. Types of parameters that you can configure are listed in this table.

Type of Parameter	Description
Model parameter argument	Block parameter in the model workspace that you configure as a model argument. Each instance of a model that includes the model parameter argument can have the argument set to a unique value.
Model parameter	Parameter defined within a model, such as a parameter in the model workspace.
External parameter	Parameter defined as an object in the base workspace or in a data dictionary.
Constant	Constant parameter.

To configure a parameter for code generation, you must associate the parameter with a data object. For example, before configuring a MATLAB variable for code generation, in the Model Explorer, convert the variable to a parameter object.

When you open a model in the **Embedded Coder** app, model parameters, model parameter arguments, and constants that are associated with data objects appear in the Code Mappings editor. In the editor, you can configure the parameters for code generation. If a model uses external parameters, you can add them to the Code Mappings editor by clicking the `Refresh` link to the right of the parameter name. That link initiates an update diagram and, in the editor view, adds external parameters that are used by the model.

Configure parameter data to:

- Make the data accessible for interaction while the generated code executes.
- Minimize the amount of data that is stored in memory.
- Control where the code generator places parameter data in memory.
- Promote parameter data to the model interface so that other components and systems can access that data.
- Improve readability and traceability of the generated code.

To control whether the parameters throughout a model are tunable, you can gain access to model configuration parameter **Default parameter behavior** by using the `'Auto' will be tunable/inline` link on the Code Mappings editor **Data Defaults** tab. For example, you can use that link to make parameters tunable in these cases:

- Refine parameter settings during rapid prototyping
- Calibrate parameters
- Optimize parameters for production code

For code generation, examples show how to configure model parameters and model parameter arguments for the model `rtwdemo_configinterface`. You can configure code mappings by using the **Code Mappings Editor – C** or code mappings programming interface (`coder.mapping.api.CodeMapping`).

Choose Customization Options for Parameters

By default, parameters in a model appear in generated code as fields of global data structures named:

- *model*_P for model parameters
- *model*_InstP for model parameter arguments
- *model*_ConstP for constant parameters

Based on your code interface requirements, decide whether to customize generation of parameter data. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of parameters in generated code for optimization purposes. If you configure customizations, decide:

- Whether to set up a default configuration

  If a model includes a significant number of parameters of a category that must be tunable (for example, more than 10), it is more efficient to configure the parameters of that category by using a default setting, and then override that setting for special cases. If the model includes a few parameters of a given category that have unique source, naming, or placement requirements, consider configuring the parameters individually.
- How to declare and handle model parameter data in the generated code

  - As separate global variables
  - To read model parameter data from global variables defined in external code
  - As calls to access functions. Requires Embedded Coder

  For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations for model parameters include whether to:

- Name parameters in the generated code by using parameter names in the model or by using unique code identifiers.
- Support preprocessor conditionals defined by compiler flags or options. Requires Embedded Coder. See “Use Variant Models to Generate Code That Uses C Preprocessor Conditionals” on page 27-70.
- Include the `static` type qualifier in global variable definitions and declarations, for example, to prevent name clashes. Requires Embedded Coder. See “Prevent Name Clashes by Configuring Data Item as static” on page 37-8.
- Include the `const`, `volatile`, or `const` and `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
- Generate a macro (`#define`) or code that uses a macro defined in an external header file. Requires Embedded Coder. See “Macro Definitions (#define)” on page 26-56.
- Generate a global data structure with a name that you specify. Requires Embedded Coder. See “Organize Data into Structures in Generated Code” on page 36-147.
- Place parameter data into a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of interface requirements that are relevant to parameters that have corresponding storage classes and storage class properties, see “Choose Storage Class and Storage Class Properties for Data Stores” on page 33-89.

Parameter requirements for example model `rtwdemo_configinterface` are:

- By default, retain model parameters in the generated code for tuning. Do not optimize the code by inlining the parameters.
- Represent model parameters as separate global variables defined and declared with the `const` and `volatile` type qualifiers.
- Apply prefix `mp_` to names of variables that represent model parameters.
- Enable constant parameters `UPPER` and `LOWER` to have different values for each instance of the model.
- For each instance of a model parameter argument, by default, allocate an area of memory represented as a uniquely named field in an exported structure.
- Model parameter arguments appear as individual model function arguments in the generated code. Enable the code generator to optimize model parameter arguments that are not used by blocks in the model.

For this example, configure model parameters and model parameter arguments in `rtwdemo_configinterface` to meet these code generation requirements.

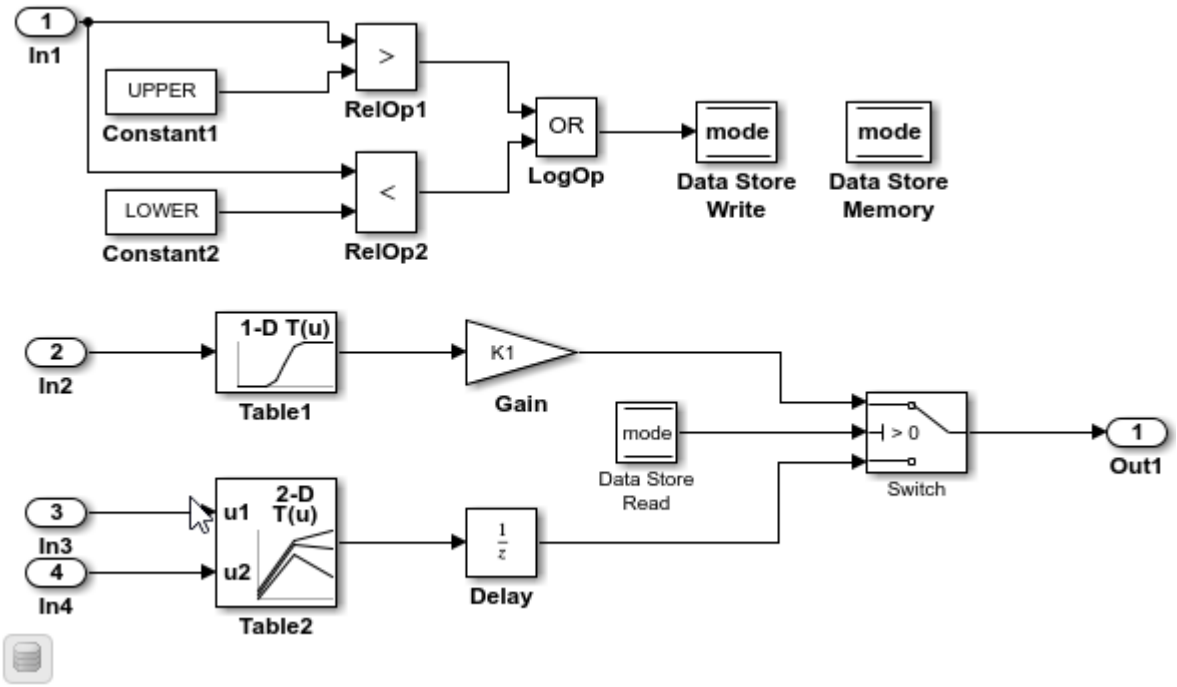
Configure Default Code Generation Settings for Parameters

Default code generation settings for parameters can reduce the effort of preparing a model for code generation, especially if a model has a significant number of parameters with which you need to interact while the generated code executes. Choose configuration settings once, and the code generator applies those settings to parameters across the model. Simulink stores the default configuration as part of the model.

Consider configuring default code generation settings for model parameters if your model uses multiple parameters for the same category that do not have unique requirements or use a shared Embedded Coder Dictionary.

This example shows how to use the **Code Mappings Editor – C** to configure default settings for model parameters and model parameter arguments for the model `rtwdemo_configinterface`. Configure model parameters so that they are tunable and are defined and declared in the generated code as separate global variables that have type qualifiers `const` and `volatile`. Configure model parameter arguments to use default settings, where the code generator allocates an area of memory represented as a uniquely named field in an exported structure for each argument instance.

1. Open model `rtwdemo_configinterface`. Save a copy of the model to a writable location.



2 Open the **Embedded Coder** app.

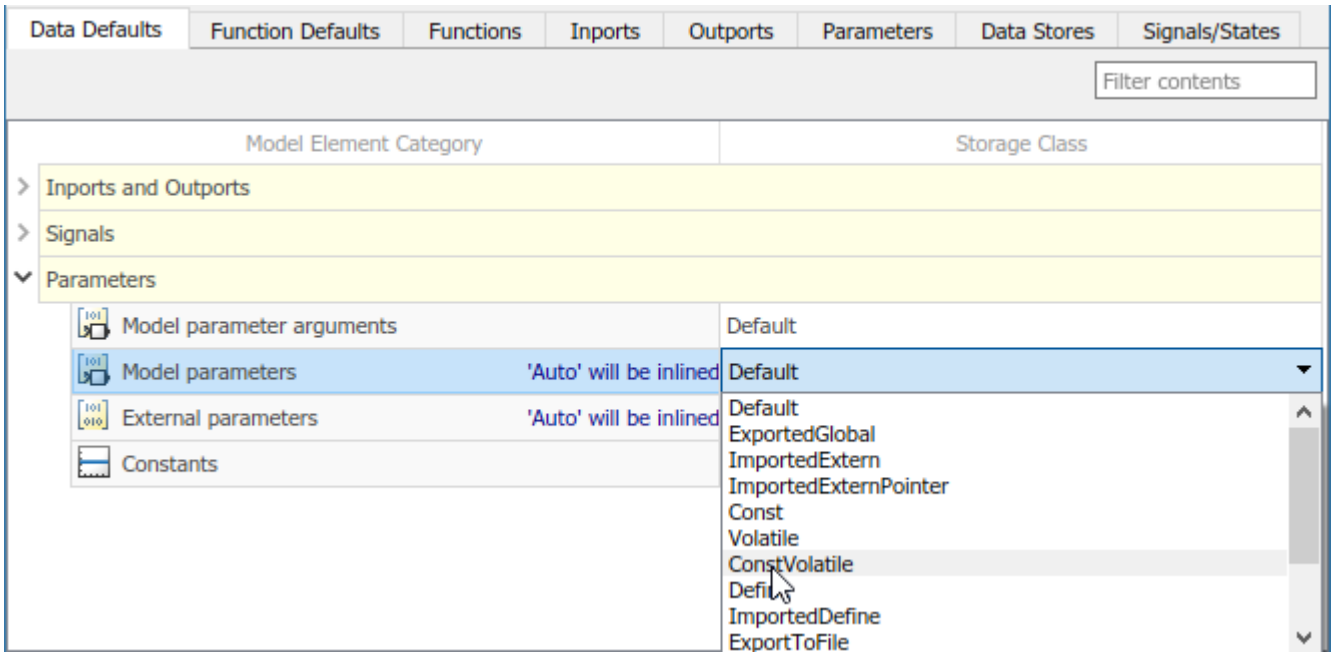
3 In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.

4 In the Code Mappings editor, select **Model parameter arguments**. Leave the storage class set to `Default`. The code generator allocates a separate area of memory for each argument instance.

5 In the Code Mappings editor, under **Parameters**, select category **Model parameters**. Link text `'Auto' will be inlined` indicates that the code generator is configured to inline model parameters by default. A requirement for this example is model parameters be tunable. Click `'Auto' will be inlined`.

6 In the Model Configuration Parameters dialog box, set model configuration parameter **Default parameter behavior** to `Tunable`. Save the change and close the dialog box. In the Code Mappings editor, the link text changes to `'Auto' will be tunable`.

7 In the Code Mappings, with the **Model parameters** category still selected, set the storage class to `ConstVolatile`.



8 Save the model.

Configure Code Generation for Individual Parameters

You can configure individual parameters for code generation. For example, if a model has two parameters of the same category that have unique code generation requirements, configure the parameters individually. Or, if you configure default settings for a category of parameters, you can override those settings for specific parameters.

If your model meets at least one of these criteria, consider configuring code generation settings for parameters individually:

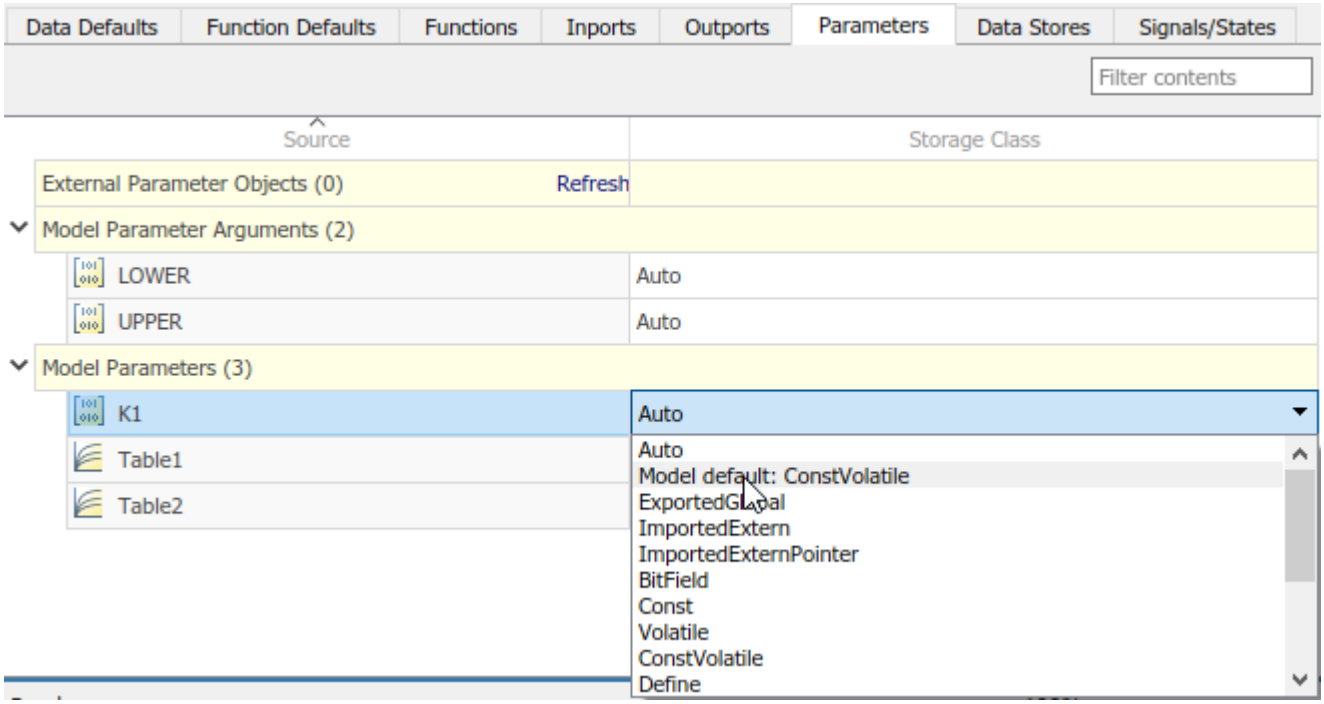
- Uses multiple parameters of the same category that have unique source, naming, or placement requirements.
- Uses a few parameters of the same category.
- Has a default configuration for a category of parameters and you need to override the configuration for some specific parameters.

This example shows how to use the Code Mappings editor to apply your default storage class setting for model parameters to parameters `K1`, `Table1`, and `Table2` in model `rtwdemo_configinterface`. You configure code identifiers for those parameters and for model parameter arguments `LOWER` and `UPPER`. You can specify code generation identifiers, for example for integration, without modifying the model design.

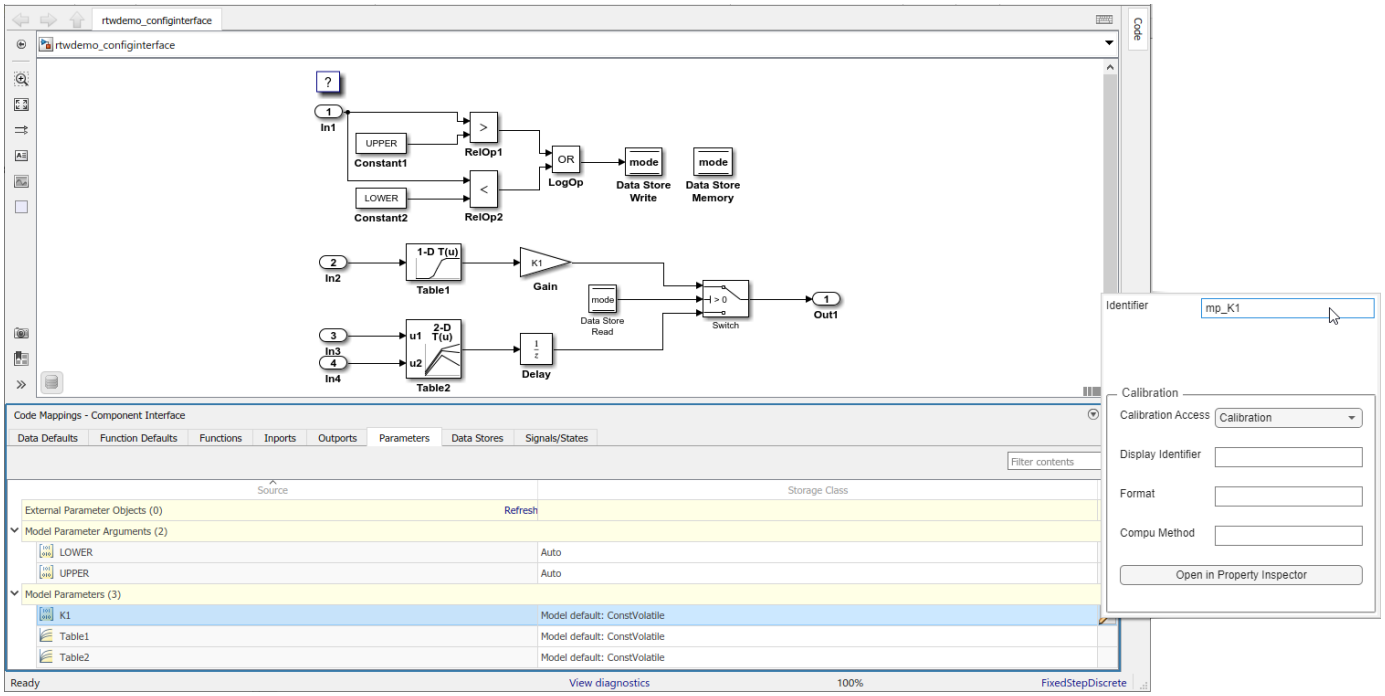
1 If you have not already done so, complete the steps in “Configure Default Code Generation Settings for Parameters” on page 33-63.

2 In the Code Mappings editor, click the **Parameters** tab. Expand **Model Parameter Arguments**. By default, the storage class for each model parameter is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, leave the storage class set to `Auto`.

3. Expand **Model Parameters**. By default, the storage class for each parameter is set to `Auto`. Use the model default configuration, which specifies storage class `ConstVolatile`.
   - To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
   - To override the default configuration, specify the storage class that meets the code generation requirements for that parameter.
4. In the Code Mappings editor, under **Model Parameters**, select parameters `K1`, `Table1`, and `Table2`. Set the storage class to `Model default: ConstVolatile`.



5. Configure the code identifiers for model parameters with names that include the prefix `mp_`. In the Code Mappings editor, select model parameter `K1`. Click the icon and set the storage class property **Identifier** to `mp_K1`. For parameters `Table1` and `Table2`, set **Identifier** to `mp_Table1` and `mp_Table2`.



6 Save the model.

7 Generate and view the code. For example, find the data definitions for model parameter `mp_K1` in `rtwdemo_configinterface.c`.

```
const volatile int8_T mp_K1 = 2;
```

Find where the parameter is used in the step entry-point function.

```
if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
```

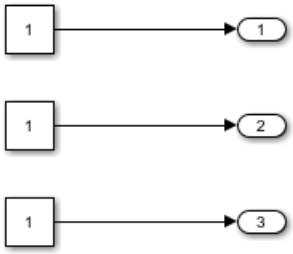
Organize Parameter Data into a Structure by Using Struct Storage Class

This example shows how to use the `Struct` storage class to organize block parameter values into a structure in the generated code. You apply the storage class directly to individual data items in a model.

To create a custom structure of parameter data in the generated code, consider creating a corresponding structure in Simulink. See “Organize Data into Structures in Generated Code” on page 36-147.

Create the Example Model

Create this example model with three Constant blocks and three Outport blocks. Name the model `ex_struct_param`.



Set Up Constant Parameters

1. Open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**.
2. In the Model Data Editor, select the **Parameters** tab.
3. In the model, select the upper Constant block.
4. In the Model Data Editor, use the **Value** column to set the constant value to `p1`. Next to `p1`, click the action button ⋮ and select **Create**.
5. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(1.0)` and click **Create**.

   Simulink adds a `Simulink.Parameter` object named `p1` to the model workspace.
6. In the property dialog box for `p1`, click **OK**.

   If you did not have to set up other parameters, you could click the **Code Generation** tab. Then, click the **Configure in Coder App** button, which opens the **Embedded Coder** app and Code Mappings editor.
7. Use the Model Data Editor to set the other constant values by using parameter objects named `p2` (value `2.0`) and `p3` (value `3.0`).

Configure Constant Parameters for Code Generation

1. Open the **Embedded Coder** app.
2. On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.
3. In the Code Mappings editor, click the **Parameters** tab.
4. Expand **Model Parameters**.
5. Set the storage class for parameters `p1`, `p2`, and `p3` to `Struct`.
6. Select parameter `p1`. Click the icon and set property **StructName** to `my_struct`. For parameters `p2` and `p3`, set **StructName**to `my_struct`.
7. Save the model.

Generate and Inspect Code

Generate and inspect the code.

The file `ex_struct_param.h` defines structure type, `my_struct_type`.

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

```
     real_T p3;
  } my_struct_type;
```

The file `ex_struct_param.c` defines global variable `my_struct`.

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

     /* p2 */
     2.0,

     /* p3 */
     3.0
};
```

Configure Code Generation Settings for Parameters Programmatically

To automate configuration of parameters for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate data configuration.

This example shows how to configure default settings for model parameters and model parameter arguments for model rtwdemo_configinterface. Configure model parameters so that they are tunable and are defined and declared in the generated code as separate global variables that have type qualifiers `const` and `volatile`. Configure model parameter arguments to use default settings, where the code generator allocates an area of memory represented as a uniquely named field in an exported structure for each argument instance. Apply your default storage class setting for model parameters to parameters `K1`, `Table1`, and `Table2`. Configure code identifiers for those parameters.

1. Open the example model.

   ```
   open_system('rtwdemo_configinterface');
   ```

2. Set model configuration parameter **Default parameter behavior** to `Tunable`.

   ```
   model='rtwdemo_configinterface';
   ```

   ```
   set_param(model,'DefaultParameterBehavior','Tunable');
   ```

3. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the code generation configuration for data elements and functions in model `rtwdemo_configdefaults`.

   ```
   cm = coder.mapping.api.get('rtwdemo_configinterface');
   ```

4. Configure default settings for model parameters by calling function `setDataDefault`. For the arguments, specify these values:

   - The object returned by `coder.mapping.api.get`
   - `ModelParameters` for the default category
   - Property name `StorageClass` with property value `ConstVolatile`

   ```
   setDataDefault(cm,'ModelParameters','StorageClass','ConstVolatile');
   ```

5. Configure default settings for model parameter arguments. In the call to `setDataDefault`, specify these values for arguments:

   - The object returned by `coder.mapping.api.get`.
   - `ModelParameterArguments` for the default category.
   - Property name `StorageClass` with property value `Default`.

```
setDataDefault(cm,'ModelParameterArguments','StorageClass','Default');
```

6 Verify your default configuration for model parameters and model parameter arguments. Issue calls to `getDataDefault` that specify the object returned by `coder.mapping.api.get`, category `ModelParameters` or `ModelParameterArguments`, and `StorageClass`.

```
getDataDefault(cm, 'ModelParameters', 'StorageClass')

ans =

    'ConstVolatile'

getDataDefault(cm, 'ModelParameterArguments', 'StorageClass')

ans =

    'Default'
```

7 Apply the default configuration for model parameters to parameters `K1`, `Table1`, and `Table2` .

By default, Simulink sets the storage class for individual parameters to `Auto`. When the storage class is `Auto`, the code generator:

- Determines whether to eliminate the data from the generated code for optimization purposes.
- If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

Configure the code generator to apply your default model parameter settings to parameters `K1`, `Table1`, and `Table2`. For each parameter, call function `setModelParameter`. Specify the object returned by `coder.mapping.api.get`, the parameter name, property name `StorageClass`, and property value `Model default`.

```
setModelParameter(cm,'K1','StorageClass','Model default');
setModelParameter(cm,'Table1','StorageClass','Model default');
setModelParameter(cm,'Table2','StorageClass','Model default');
```

8 Verify your configuration changes for parameters `K1`, `Table1`, and `Table2` by using calls to function `getModelParameter`.

```
getModelParameter(cm,'K1','StorageClass')

ans =

    'Model default'

getModelParameter(cm,'Table1','StorageClass')

ans =

    'Model default'


getModelParameter(cm,'Table2','StorageClass')

ans =

    'Model default'
```

9 Configure code identifiers for the model parameters. For each parameter, call function `setModelParameter`. Specify the object returned by `coder.mapping.api.get`, the parameter name, property name `Identifier`, and one of these property values.

Model Parameter	Code Identifier
K1	mp_K1
Table1	mp_Table1
Table2	mp_Table2

```
setModelParameter(cm,'K1','Identifier','mp_K1');
setModelParameter(cm,'Table1','Identifier','mp_Table1');
setModelParameter(cm,'Table2','Identifier','mp_Table2');
```

10 Verify your configuration changes for the model parameters by using calls to function `getModelParameter`.

```
getModelParameter(cm,'K1','Identifier')

ans =

    'mp_K1'

getModelParameter(cm,'Table1','Identifier')

ans =

    'mp_Table1'

getModelParameter(cm, 'Table2', 'Identifier')

ans =

    'mp_Table2'
```

11 Save the model.

12 Generate and view the code. For example, find the data definitions for model parameter `mp_K1` in `rtwdemo_configinterface.c`.

```
const volatile int8_T mp_K1 = 2;
```

Find where the parameter is used in the step entry-point function.

```
if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
```

Choose Storage Class and Storage Class Properties for Model Parameters

Depending on your code generation requirements, choose from these storage classes to configure code generation for model parameters. For constants, only `Default` applies. For model parameter arguments, only `Auto`, `Default`, and `Model default` apply.

Requirements	Storage Class
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)
Prevent optimizations from eliminating storage for a data element and use the default mapping for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mapping only)
Support preprocessor conditionals defined by using a compiler flag or option.	“CompilerFlag” on page 36-79
Generate a global variable definition and declaration that has the `const`, `volatile`, or `const` and `volatile` type qualifiers.	“Const, Volatile, and ConstVolatile” on page 36-79
Generate a macro (`#define` directive) or code that uses a macro defined in a header file in your external code.	“Define, ImportedDefine” on page 36-80
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate a global variable definition and declaration that has the `static` type qualifier.	“FileScope” on page 36-81 (Local and shared local data store mappings only)
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
Generate code that reads from and writes to a global variable defined by your external header file.	“ImportFromFile” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual model parameter, use the **Identifier** storage class property to configure a name for the variable representing the parameter in the generated code. With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the parameter and external code	`ExportToFile` and `Volatile`
`GetFunction`	Parameter appears in the generated code as a call to a specified `get` function	`GetSet`
`HeaderFile`	Source header file that contains declarations for global data, which is read by the parameter and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default parameter configuration only)	Memory section that contains data read by the parameter	`Default`
`Owner`	Code generator places the definition for parameters in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See "Control Placement of Global Data Definitions and Declarations in Generated Files" on page 37-2.	`ExportToFile` and `Volatile`
`PreserveDimensions`	Code generator preserves dimensions of parameter data that is represented in generated code as a multidimensional array. You must set model configuration parameter **Array layout** to `Row-major`. See "Preserve Dimensions of Multidimensional Arrays in Generated Code" on page 47-2.	`ExportToFile`, `FileScope`, `GetSet`, `ImportFromFile`, `Localizable` and `Volatile`
`SetFunction`	Parameter appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for a structure in the generated code for parameter.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- "C Code Generation Configuration for Model Interface Elements" on page 33-2
- "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75
- "Configure Model Data Elements for ASAP2 File Generation" on page 66-2
- "Configure Default C Code Generation for Categories of Data Elements and Functions" on page 33-15

Configure Block States for C Code Generation

Block states, such as the state of a Unit Delay block, retain a state value between execution cycles of an algorithm. When configuring a model for code generation, you can configure state data to:

- Minimize the amount of data that is stored in memory.
- Control where the code generator places state data in memory.
- Promote state data to the model interface so that other components and systems can access that data.
- Improve readability and traceability of the generated code.

For code generation, examples show how to configure a block state for model `rtwdemo_configinterface`. You can configure code mappings by using the **Code Mappings Editor – C** or code mappings programming interface (`coder.mapping.api.CodeMapping`).

Choose Code Configuration Options for States

Based on your code generation requirements, decide how to represent state data. By default, states in a model appear in generated code as fields of a global data structure named *`model`*`_DW`. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of states in generated code for optimization purposes. If you configure customizations, decide:

- Whether to set up a default configuration

  If you need to gain access to a significant number of states (for example, more than 10) , it is more efficient to configure the states with default settings and then override those settings for special cases. If you need to gain access to a few states that have unique source, naming, or placement requirements, consider configuring the states individually.
- How to declare and handle state data in the generated code

  - As separate global variables
  - To read input data from global variables defined in external code
  - As calls to access functions. Requires Embedded Coder

  For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations include whether to:

- Name states in the generated code by using state labels that appear in the model or by using unique code identifiers.
- Include the `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
- Place state data into a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of interface requirements that are relevant to states with corresponding storage classes and storage class properties, see “Choose Storage Class and Storage Class Properties for Data Stores” on page 33-89.

State requirements for example model `rtwdemo_configinterface` are:

- Retain the state data of the Unit Delay block for accessibility while the generated code executes.
- Represent the state as a separate global variable.
- Apply prefix `dtemp_` to the name of the variable that represents the state.

For this example, you set the default representation of the state in the generated code as a global variable with the `static` type qualifier. Then, you configure the state of the Unit Delay block to use the default storage class and a unique code identifier that includes the required prefix `dtemp_`. The code identifier capability enables you to specify code generation identifiers without having to modify the model design.

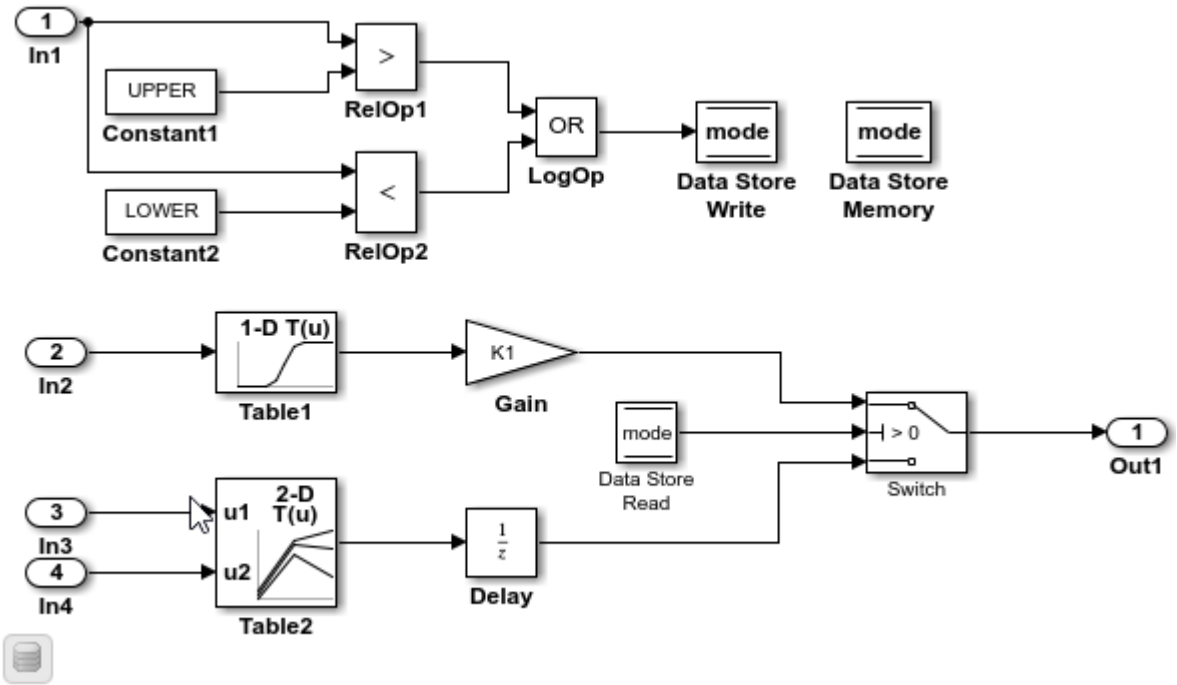
Configure Default Code Generation Settings for States

A default code generation setting for states can reduce the effort of preparing a model for code generation, especially if a model has a significant number of states that you want to gain access to while the generated code executes. Choose configuration settings once, and the code generator applies those settings to states across the model. Simulink stores the default configuration as part of the model.

Consider configuring default code generation settings for model states if your model uses multiple states that do not have unique requirements or uses a shared Embedded Coder Dictionary.

This example shows how to use the **Code Mappings Editor – C** to configure default settings for states. Use the Code Mappings editor to set the default storage class for states in model `rtwdemo_configinterface` to `FileScope`. With that storage class setting, the code generator represents state data in the generated code as global variables that have the `static` type qualifier.

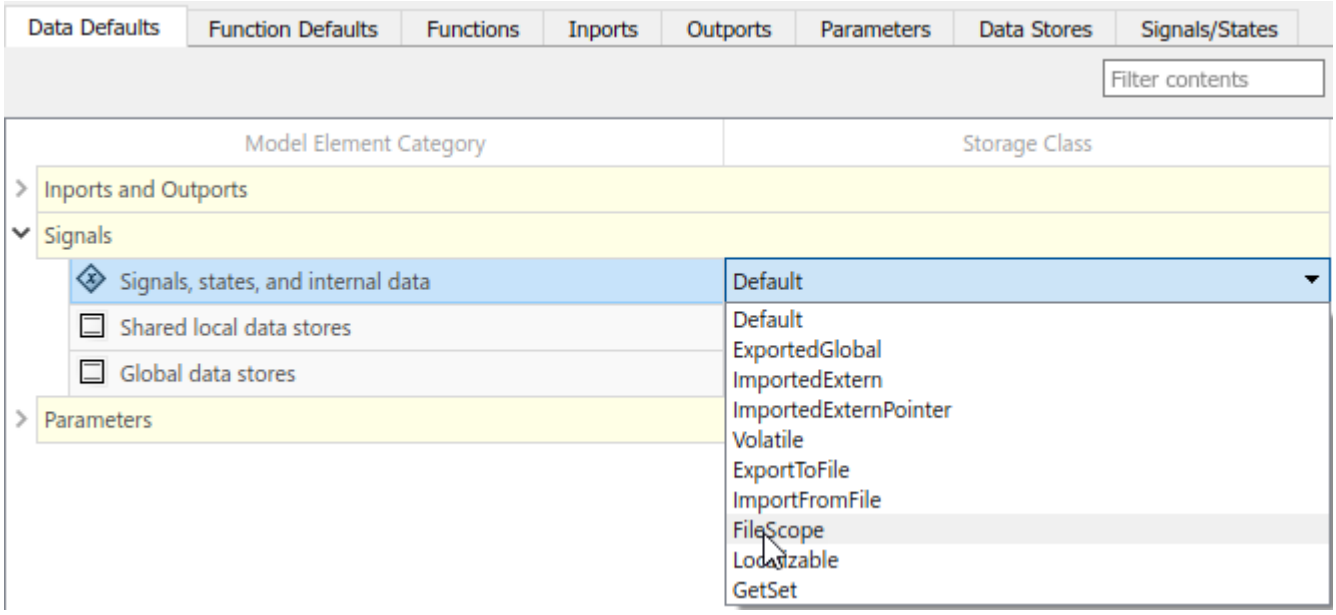
1 Open model `rtwdemo_configinterface`. Save a copy of the model to a writable location.



2 Open the **Embedded Coder** app.

3 In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.

4 In the Code Mappings editor, under **Signals**, select category **Signals, states, and internal data**. Set the default storage class to `FileScope`.



5 Save the model.

Configure Code Generation Settings for Individual States

You can configure individual states for code generation. For example, if a model has two states that have unique code generation requirements, configure the states individually. Or, if you configure default settings for states, you can override those settings for specific states.

If your model meets at least one of these criteria, consider configuring code generation settings for states individually:

- Uses multiple states that have unique source, naming, or placement requirements.
- Uses a few states.
- Has a default configuration for states and you need to override the configuration for some specific states.

This example shows how to use the Code Mappings editor to apply your default storage class setting to the Unit Delay block state X in model `rtwdemo_configinterface`.

The example also shows how to configure a code identifier for that state. You can specify code generation identifiers, for example for integration, without modifying the model design.

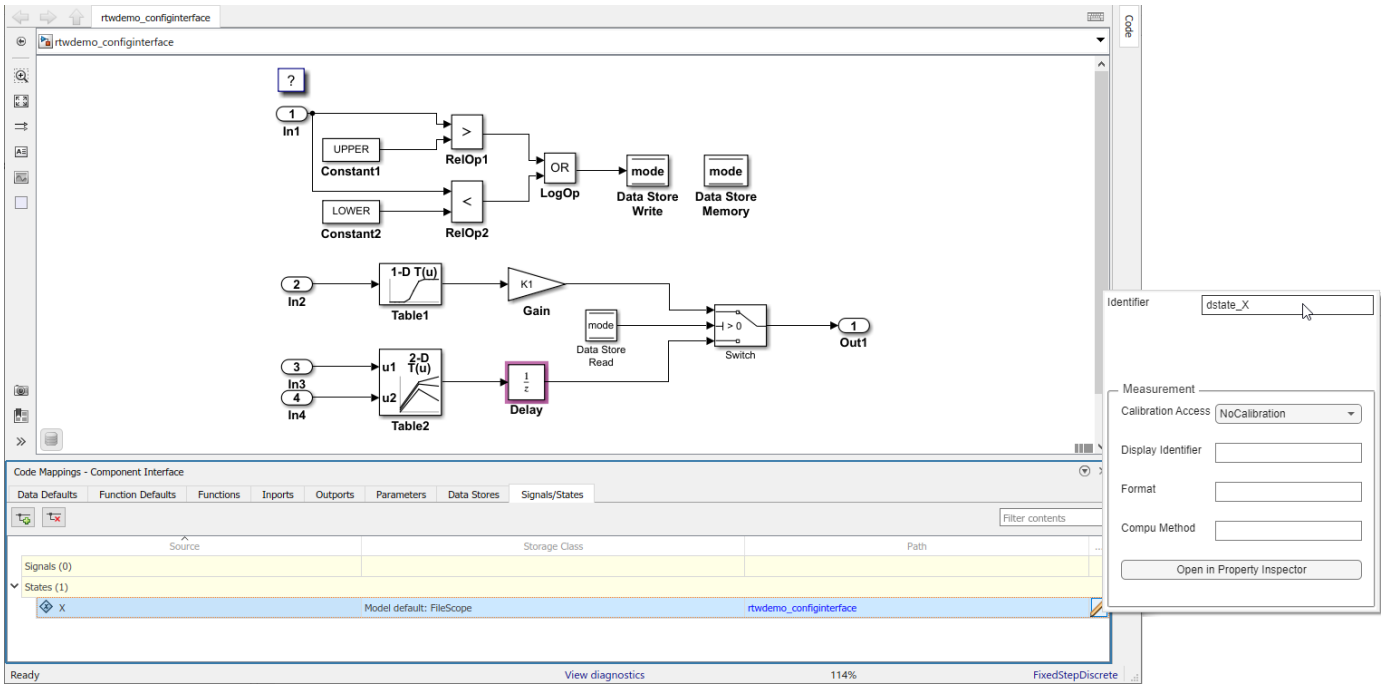
1 If you have not already done so, complete the steps in “Configure Default Code Generation Settings for States” on page 33-75.

2 In the Code Mappings editor, click the **Signals/States** tab. Expand **States**. The storage class for the state is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the

code generator applies the model default configuration. For this example, the model default configuration specifies storage class `FileScope`.

- To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.
- To override the default configuration, specify the storage class that meets the code generation requirements for that state.

3. In the Code Mappings editor, select state `X`. Set the storage class to `Model default: FileScope`.
4. Configure the code identifier for the state with a name that includes the prefix `dstate_`. In the Code Mappings editor, select state `X`. Click the icon and set the storage class property **Identifier** to `dstate_X`.



5. Save the model.
6. Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definitions for the state data.

```
static MYTYPE dstate_X;
```

Find where the state data is used in the step entry-point function.

```
.
.
.
  if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
.
.
.
   dstate_X = dout_Table2;
}
```

Configure Code Generation Settings for States Programmatically

To automate configuration of states for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or as part of creating an application test environment, use the programming interface to automate data configuration.

This example shows how to use the programming interface to configure states for model rtwdemo_configinterface. Set the default representation of states in the generated code as global variables that have the `static` type qualifier. Then, configure state `X` of the Unit Delay block to use the default storage class and unique code identifier that includes the required prefix `dstate_`.

1. Open the example model.

   ```
   open_system('rtwdemo_configinterface')
   ```

2. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the code generation configuration for data and functions for model `rtwdemo_configinterface`.

   ```
   cm = coder.mapping.api.get('rtwdemo_configinterface');
   ```

3. Configure default settings for states by calling function `setDataDefault`. For the arguments, specify these values:

   - The object returned by `coder.mapping.api.get`
   - `InternalData` for the default category
   - Property name `StorageClass` with property value `FileScope`

   ```
   setDataDefault(cm,'InternalData','StorageClass','FileScope');
   ```

4. Verify your default configuration for states. Issue a call to `getDataDefault` that specifies the object returned by `coder.mapping.api.get` and category `InternalData`. Specify the third argument as property `StorageClass`.

   ```
   getDataDefault(cm,'InternalData','StorageClass')

   ans =

       'FileScope'
   ```

5. Apply the default configuration for states to state `X`.

   By default, Simulink sets the storage class for individual states to `Auto`. The code generator:

   - Determines whether to eliminate the data from the generated code for optimization purposes.
   - If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

   To control the configuration for a state, call function `setState`.

   Issue a call to `setState` that specifies:

   - Object returned by `coder.mapping.api.get`
   - State name `X`
   - Default storage class previously set for states by using property `StorageClass` and property value `Model default`.
   - Property `Identifier` and property value `dstate_X`

```
setState(cm,'X','StorageClass','Model default','Identifier','dstate_X');
```

6 Verify your configuration changes by calling function `getState`. Specify the object returned by `coder.mapping.api.get`, name of the state, and property `StorageClass` or `Identifier`.

```
getState(cm,'X','StorageClass')

ans =

    'Model default'

getState(cm,'X','Identifier')

ans =

    'dstate_X'
```

7 Save the model.

8 Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definitions for state data.

```
static MYTYPE dstate_X;
```

Find where the state data is used in the step entry-point function.

```
.
.
.
  if (mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
  }
.
.
.
   dstate_X = dout_Table2;
}
```

Choose Storage Class and Storage Class Properties for States

Depending on your code generation requirements, choose from these storage classes to configure code generation for block states.

Requirements	Storage Class
Let the code generator handle how to represent the state in the generated code. For example, for optimization purposes, the code generator might eliminate or change the representation of data.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)
Prevent optimizations from eliminating storage for a data element and use the default mapping for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mapping only)

Requirements	Storage Class
Generate a global variable definition and declaration that has the `volatile` type qualifier.	`Volatile` (See “Const, Volatile, and ConstVolatile” on page 36-79)
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate a global variable definition and declaration that has the `static` type qualifier.	“FileScope” on page 36-81 (Local and shared local data store mappings only)
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
“ExportToFile” on page 36-80	“ImportFromFile” on page 36-82
Generate variables that are local to functions.	“Localizable” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)
Generate a global variable that enables buffer reuse.	“Reusable” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual state, use the **Identifier** storage class property to configure a name for the variable representing the state in the generated code. If you leave the **Identifier** property blank, the code generator uses the **State name** parameter of the block that has the state. If this parameter is empty, the code generator uses the name of the block.

With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the state and external code	`ExportToFile` and `Volatile`
`GetFunction`	State appears in the generated code as a call to a specified `get` function	`GetSet`
`HeaderFile`	Source header file that contains declarations for global data, which is read by the state and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default state configuration only)	Memory section that contains data read by the state	`Default`

Property	Description	Storage Classes
`Owner`	Code generator places the definition for states in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.	`ExportToFile` and `Volatile`
`PreserveDimensions`	Code generator preserves dimensions of state data that is represented in generated code as a multidimensional array. You must set model configuration parameter **Array layout** to `Row-major`. See “Preserve Dimensions of Multidimensional Arrays in Generated Code” on page 47-2.	`ExportToFile`, `FileScope`, `GetSet`, `ImportFromFile`, `Localizable` and `Volatile`
`SetFunction`	State appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for a structure in the generated code for state.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Configure Model Data Elements for ASAP2 File Generation” on page 66-2

Configure Data Stores for C Code Generation

Data stores enable subsystems and referenced models to share data without using I/O ports to pass the data from level to level (see “Data Store Basics” and “Data Stores in Generated Code” on page 36-94). Types of data stores include the types listed in this table.

Type of Data Store	Description
Local data store	Data store that is accessible from anywhere in a model hierarchy that is at or below the level at which you define the data store. You can define a local data store graphically in a model by including a Data Store Memory block or by creating a signal object (synthesized data store) in the model workspace.
Shared local data store	Data Store Memory block with the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model.
Global data store	Data store that is defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores.

When you open a model in the **Embedded Coder** app, local and shared local data stores appear in the Code Mappings editor, where you can configure the data stores for code generation. If a model uses global data stores, you can view them in the Code Mappings editor by clicking the `Refresh` link that appears to the right of the data store name. That link initiates an update diagram and adds global data stores that the model uses in the editor view.

Configure data stores to:

- Make data accessible for interaction while the generated code executes.
- Control where the code generator places data in memory.
- Improve readability and traceability of generated code.

For code generation, examples show how to configure a data store for model `rtwdemo_configinterface`. You can configure code mappings by using the **Code Mappings Editor – C** or code mappings API (`coder.mapping.api.CodeMapping`).

Choose Customization Options for Data Stores

By default, local data stores in a model appear in generated code as fields of a global data structure named *`model`*`_DW`. Shared local data stores appear as a field of global data structure *`model`*`_SharedDSM`. Based on your code interface requirements, decide whether to customize code generation of data stores. If you do not configure customizations, the code generator determines whether to eliminate or change the representation of data stores in generated code for optimization purposes. If you configure customizations, decide:

- Whether to set up a default configuration

  If a model includes a significant number (for example, more than 10) of data stores for a given category that must be accessible during program execution, it is more efficient to configure the data stores with a default setting, and then override that setting for special cases. If the model

includes a few data stores for a given category that have unique source, naming, or placement requirements, consider configuring the data stores individually.

- How to declare and handle model data stores in the generated code
  - As separate global variables
  - To read from data stores as global variables defined in external code
  - As calls to access functions. Requires Embedded Coder

  For more information about these options, see “Control Data and Function Interface in Generated Code” on page 36-44.

Other considerations for model parameters include whether to:

- Name data stores in the generated code by using data store names in the model or by using unique code identifiers.
- Include the `static` type qualifier in global variable definitions and declarations, for example, to prevent name clashes. Requires Embedded Coder. See “Prevent Name Clashes by Configuring Data Item as static” on page 37-8.
- Include the `volatile` type qualifier in global variable definitions and declarations. Requires Embedded Coder. See “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
- Generate a global data structure with a name that you specify. Requires Embedded Coder. See “Organize Data into Structures in Generated Code” on page 36-147.
- Place data stores in a specific area of memory. Requires Embedded Coder. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

For a list of interface requirements that are relevant to data stores with corresponding storage classes and storage class properties, see “Choose Storage Class and Storage Class Properties for Data Stores” on page 33-89.

For example model `rtwdemo_configinterface`, data store requirements are:

- Represent local data stores as separate global variables defined and declared with the `static` type qualifier.
- Represent shared local data stores as separate global variables defined and declared with the `volatile` type qualifier.
- Apply prefix `ds_` to names of variables that represent data stores.

For this example, you set the default representation for local data stores in the generated code as global variables that have the `static` type qualifier and shared local data stores as global variables that have the `volatile` type qualifier. Then, you configure the local data store in the model to use the default storage class and a unique code identifier that includes the required prefix `ds_`. The code identifier capability enables you to specify code generation identifiers without having to modify the model design.

Configure Default Code Generation Settings for Data Stores

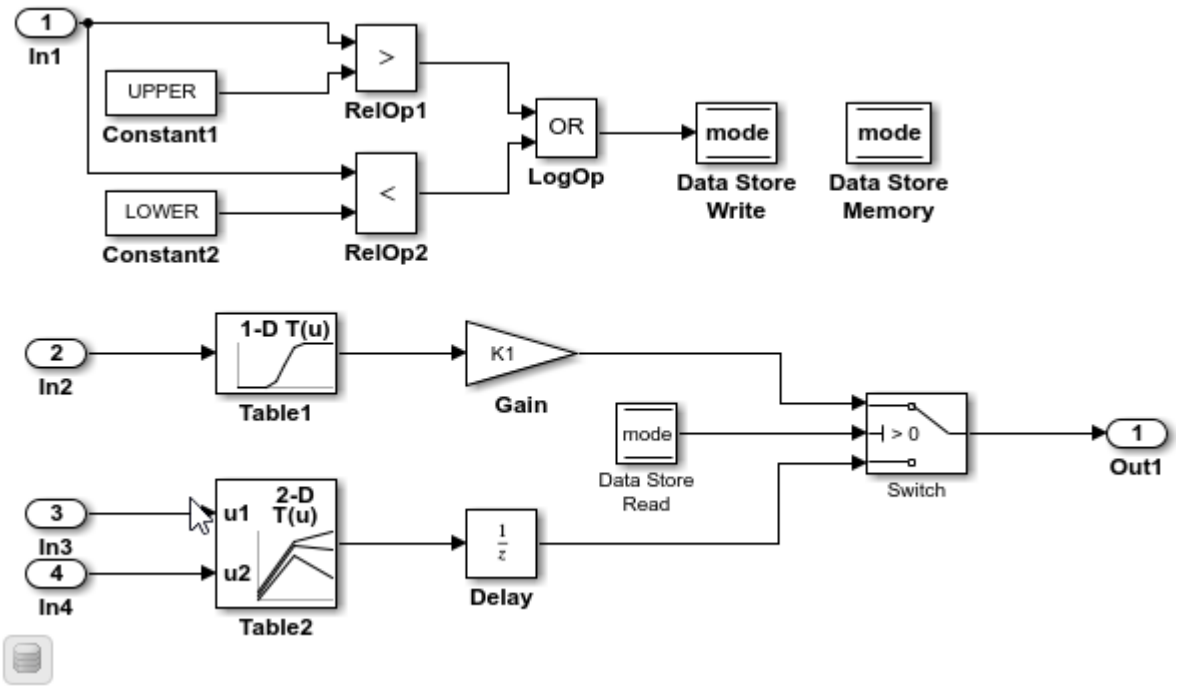
Default code generation settings for data stores can reduce the effort of preparing a model for code generation, especially if a model has a significant number of data stores that you want to gain access to while the generated code executes. Choose configuration settings once, and the code generator

applies those settings to data stores across the model. Simulink stores the default configuration as part of the model.

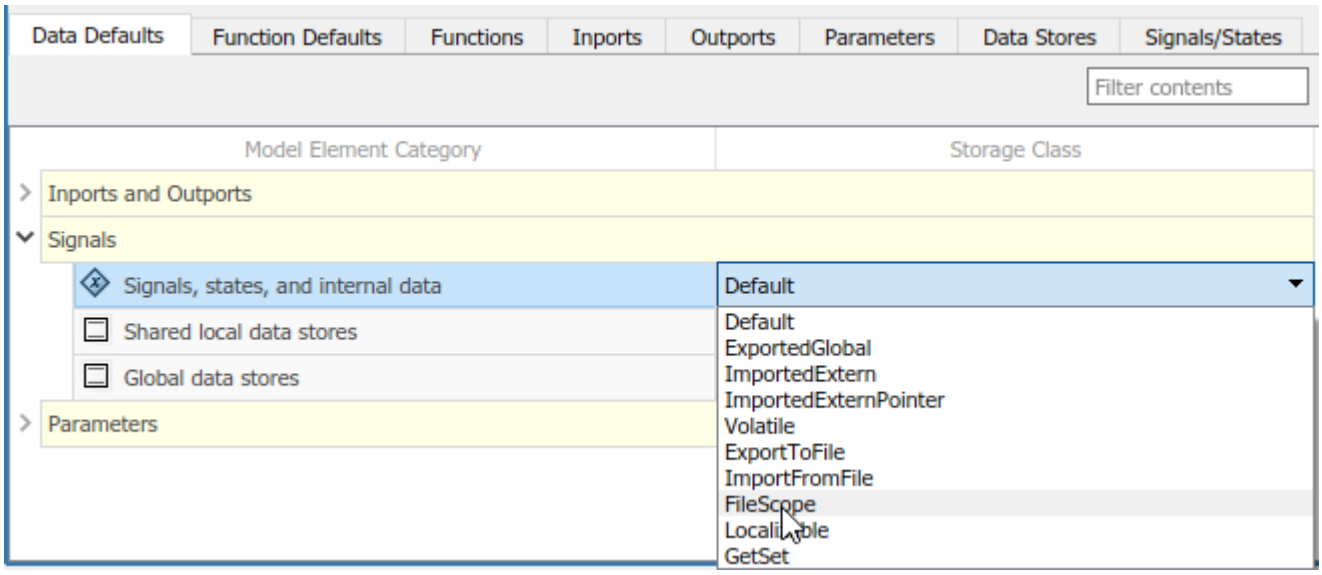
Consider configuring default code generation settings for model data stores if your model uses multiple data stores for a given category that do not have unique requirements or use a shared Embedded Coder Dictionary.

This example shows how to use the **Code Mappings Editor – C** to configure default settings for data stores. Use the Code Mappings editor to set the default storage class for local and shared local data stores in model `rtwdemo_configinterface` to `FileScope` and `Volatile`, respectively. With those storage class settings, the code generator represents the data stores in the generated code as global variables that have the `static` and `volatile` type qualifiers.

1. Open model `rtwdemo_configinterface`. Save a copy of the model to a writable location.



2. Open the **Embedded Coder** app.
3. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
4. Configure default code mappings for local data stores. In the Code Mappings editor, on the **Data Defaults** tab, expand the **Signals** node. Select category **Signals, states, and internal data**. Set the storage class to `FileScope`.



5 Configure default code mappings for shared local data stores. Select category **Shared local data stores**. Set the storage class to `Volatile`.

6 Save the model.

Configure Code Generation for Individual Data Stores

You can configure individual data stores for code generation. For example, if a model has two data stores of the same category that have unique code generation requirements, configure the data stores individually. Or, if you configure default settings for a category of data stores, you can override those settings for specific data stores.

If your model meets at least one of these criteria, consider configuring code generation settings for data stores individually:

- Uses multiple data stores of the same category that have unique source, naming, or placement requirements.
- Uses a few data stores of the same category.
- Has a default configuration for a category of data stores and you need to override the configuration for some specific states.

This example shows how to use the Code Mappings editor to apply your default storage class setting to the Data Store Memory block `mode` in model `rtwdemo_configinterface`. The example also shows how configure a code identifier for that data store.

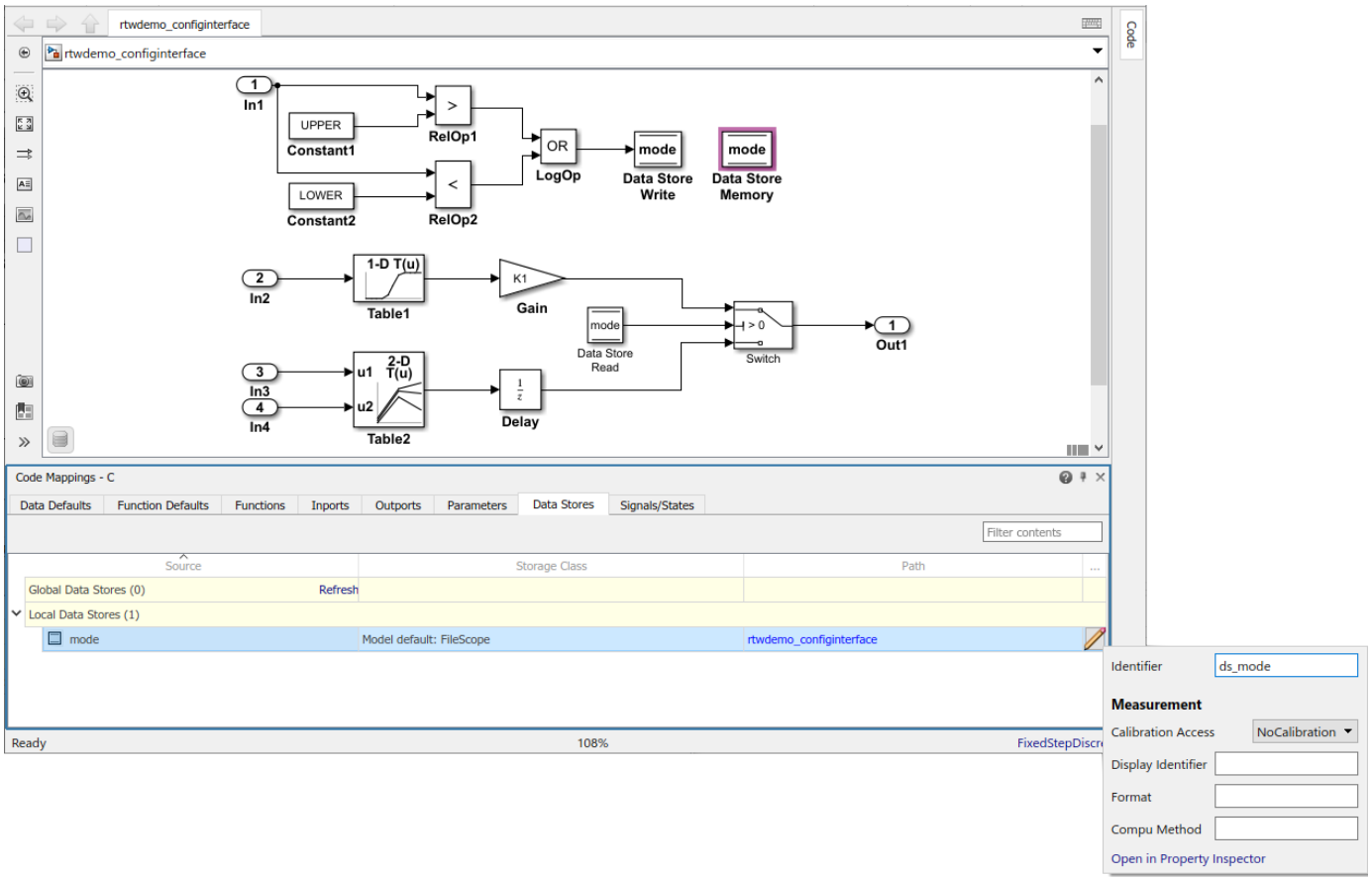
1 If you have not already done so, complete the steps in “Configure Default Code Generation Settings for Data Stores” on page 33-83.

2 In the Code Mappings editor, click the **Data Stores** tab. Expand **Local Data Stores**. The storage class for data store `mode` is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, the model default configuration specifies the storage class `Volatile`.

   - To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`.

- To override the default configuration, specify the storage class that meets the code generation requirements for that data store.

3. In the Code Mappings editor, select local data store `mode`. Set the storage class to `Model default: Volatile`.
4. Configure the code identifier for the data store with a name that includes the prefix `ds_`. In the Code Mappings editor, select shared local data store `mode`. Click the icon and set the storage class property **Identifier** to `ds_mode`.



5. Save the model.
6. Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definitions for the data store.

```
static boolean_T ds_mode;
```

Find where the state data is used in the step entry-point function.

```
.
.
.
  ds_mode = ((input1 > rtwdemo_configinterface_UPPER) || (input1 <
              rtwdemo_configinterface_LOWER));
.
.
.
  if (ds_mode) {
    output = (real_T)mp_K1 * dout_Table1;
```

```
  } else {
    output = dstate_X;
  }
.
.
.
```

Configure Code Generation Settings for Data Stores Programmatically

To automate configuration of data stores for code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate data configuration.

This example shows how to use the programming interface to configure data stores for model `rtwdemo_configinterface`. Set the default representation of local and shared local data stores in the generated code as global variables that have the `static` and `volatile` type qualifier, respectively. Configure the Data Store Memory block `mode` to use the default storage class and a unique code identifier that includes the required prefix `ds_`.

1. Open the example model.

   ```
   open_system('rtwdemo_configinterface')
   ```

2. Create object `cm` by calling function `coder.mapping.api.get`. The object stores the data and function code generation configuration for model `rtwdemo_configinterface`.

   ```
   cm = coder.mapping.api.get('rtwdemo_configinterface');
   ```

3. Configure default settings for local data stores by calling function `setDataDefault`. For the arguments, specify these values:

   - The object returned by `coder.mapping.api.get`
   - `InternalData` for the default category
   - Property name `StorageClass` with property value `FileScope`

   ```
   setDataDefault(cm,'InternalData','StorageClass','FileScope');
   ```

4. Configure default settings for shared local data stores by calling function `setDataDefault`. Specify these values for arguments:

   - The object returned by `coder.mapping.api.get`.
   - `SharedLocalDataStore` for the default category.
   - Property name `StorageClass` with property value `Volatile`.

   ```
   setDataDefault(cm,'SharedLocalDataStore','StorageClass','Volatile');
   ```

5. Verify your default configuration settings for local and shared local data stores. Issue a call to `getDataDefault` that specifies the object returned by `coder.mapping.api.get`, category `InternalData`, and the property `StorageClass`. In a second call to `getDataDefault`, replace the category with `SharedLocalDataStore`.

   ```
   getDataDefault(cm,'Internal','StorageClass')

   ans =

       'FileScope'

   getDataDefault(cm,'SharedLocalDataStore','StorageClass')
   ```

```
ans =

    'Volatile'
```

6 Apply the default configuration for local data store `mode`.

By default, Simulink sets the storage class for data stores to `Auto`. When the storage class is `Auto`, the code generator:

- Determines whether to eliminate the data from the generated code for optimization purposes.
- If retaining the data, determines how to efficiently represent the data in the generated code, taking into account default configuration settings.

To control the configuration for an individual data store, call function `setDataStore`. Issue a call to `setDataStore` that specifies:

- Object returned by `coder.mapping.api.get`
- Data store name `mode`
- Default storage class previously set for the data store by using property `StorageClass` and property value `Model default`.
- Property `Identifier` and property value `ds_mode`

```
setDataStore(cm,'mode','StorageClass','Model default','Identifier','ds_mode');
```

7 Verify your configuration changes by calling function `getDataStore`. Specify the object returned by `coder.mapping.api.get`, the name of the data store, and property `StorageClass` or `Identifier`.

```
getDataStore(cm,'mode','StorageClass')

ans =

    'Model default'

getDataStore(cm,'mode','Identifier')

ans =

    'ds_mode'
```

8 Save the model.

9 Generate and view the code. For example, in `rtwdemo_configinterface.c`, find the data definition for the data store.

```
static boolean_T ds_mode;
```

Find where the data store is used in the step entry-point function.

```
.
.
.
  ds_mode = ((input1 > rtwdemo_configinterface_UPPER) || (input1 <
              rtwdemo_configinterface_LOWER));
.
.
.
  if (ds_mode) {
    output = (real_T)mp_K1 * dout_Table1;
  } else {
    output = dstate_X;
```

```
    }
  .
  .
  .
```

Choose Storage Class and Storage Class Properties for Data Stores

Depending on your code generation requirements, choose from these storage classes to configure code generation for data stores.

Requirements	Storage Class
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual mappings only)
For data elements that cannot be optimized, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping only)
Prevent optimizations from eliminating storage for a data element and use the default storage class for the data element category.	“Model Default” on page 36-77 (Individual mappings only) , `Dictionary Default` (Individual mappings only)
Generate a structure that stores Boolean data in named bitfields.	“Bitfield” on page 36-79 (Individual mapping only)
Generate a global variable definition and declaration that has the `volatile` type qualifier.	`Volatile` (See “Const, Volatile, and ConstVolatile” on page 36-79)
Generate a global variable definition and declaration.	“ExportedGlobal” on page 36-78
Generate a global variable definition and declaration to a specified file.	“ExportToFile” on page 36-80
Generate a global variable definition and declaration that has the `static` type qualifier.	“FileScope” on page 36-81 (Local and shared local data store mappings only)
Generate code that interacts with data by calling your custom accessor functions.	“GetSet” on page 36-81
Generate code that reads from and writes to a global variable or global variable pointer defined by your external code.	“ImportedExtern, ImportedExternPointer” on page 36-78
Generate code that reads from and writes to a global variable defined by your external header file.	“ImportFromFile” on page 36-82
Generate variables that are local to functions.	“Localizable” on page 36-82
Generate a global structure that has a name, which you can specify.	“Struct” on page 36-83 (Individual mappings only)
Generate a global variable that enables buffer reuse.	“Reusable” on page 36-83 (Individual mappings only)

The list of available storage classes might include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special requirements that are not met by the listed storage classes and you have Embedded Coder software, you can define a storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

For an individual data store, use the **Identifier** storage class property to configure a name for the variable representing the data store in the generated code. With Embedded Coder, depending on the storage class that you choose, you can also configure these properties.

Property	Description	Storage Classes
`DefinitionFile`	Source definition file that contains definitions for global data, which is read by the data store and external code	`ExportToFile` and `Volatile`
`GetFunction`	Data store appears in the generated code as a call to a specified `get` function	`GetSet`
`HeaderFile`	Source header file that contains declarations for global data, which is read by the data store and external code	`ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`
`Memory Section` (default data store configuration only)	Memory section that contains data read by the data store	`Default`
`Owner`	Code generator places the definition for data stores in the code generated for one of multiple models in a model hierarchy that share definitions. You must select the model configuration parameter **Use owner from data object for data definition placement**. See “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.	`ExportToFile` and `Volatile`
`PreserveDimensions`	Code generator preserves dimensions of data stores that are represented in generated code as a multidimensional array. You must set the model configuration parameter **Array layout** to `Row-major`. See “Preserve Dimensions of Multidimensional Arrays in Generated Code” on page 47-2.	`ExportToFile`, `FileScope`, `GetSet`, `ImportFromFile`, `Localizable` and `Volatile`
`SetFunction`	Data store appears in the generated code as a call to a specified `set` function.	`GetSet`
`StructName`	Name for a structure in the generated code for the data store.	`BitField` and `Struct`

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

- “Configure Model Data Elements for ASAP2 File Generation” on page 66-2

Manage Multiple Code Mappings for a Model

Code mappings separate code generation configurations from model designs. This separation enables you to design models that are platform-neutral and ready to deploy to different target platform environments. If you have access to the necessary products, you can associate a model with code mappings for several different platforms.

Platform	System Target File	Programming Language	Required Products
C rapid prototyping	GRT-based	C	Simulink Coder
C production	ERT-based	C	Embedded Coder
C++ production	ERT-based	C++	Embedded Coder
AUTOSAR classic	AUTOSAR	C	AUTOSAR Blockset, Embedded Coder (for generating code)
AUTOSAR adaptive	AUTOSAR Adaptive	C++	AUTOSAR Blockset, Embedded Coder (for generating code)

You can generate code for the different platforms from a model or model hierarchy. Associate the models in a model hierarchy with the same platform.

Simulink activates an existing code mapping when you do one of the following:

- In the Simulink Editor, open a coder app for a model and specify an output type that supports code mappings.
- Use the code mappings API to create a code mapping.

The Code Mappings editor and code mappings programming interface display and operate on the active code mapping.

Configure Alternative Code Mappings by Using Configuration Sets

You can configure alternative code mappings for a model by creating multiple model configuration sets and changing which configuration set is active. Simulink creates or opens an existing code mapping for a model based on settings of these model configuration parameters:

- **System target file**
- **Language**
- **Code interface packaging**

Create a configuration set for each platform for which you want to generate code for a model. Code mappings are supported for these combinations of parameter settings.

Platform	System target file	Language	Code interface packaging
C/C++ rapid prototyping	`grt.tlc` or derivative of `grt.tlc`	`C/C++`	`Nonreusable function` or `Reusable function`
C/C++ production	`ert.tlc` or derivative of `ert.tlc`	`C/C++`	`Nonreusable function` or `Reusable function`
C++ production	`ert.tlc` or derivative of `ert.tlc`	`C++`	`C++ class`
AUTOSAR classic	`autosar.tlc`	`C`	`Nonreusable function` or `Reusable function`
AUTOSAR adaptive	`autosar_adaptive.tlc`	`C++`	`C++ class`

For C++ platforms, reusable and nonreusable functions use C code mappings.

Activate the configuration set of interest. Then, open the corresponding code mappings in the Simulink Editor, as described in “Activate Code Mappings in the Simulink Editor” on page 33-93 or the code mappings API in “Create and Activate Code Mappings Programmatically” on page 33-94.

Activate Code Mappings in the Simulink Editor

When you open a model in the Simulink Editor, Simulink activates an existing code mapping for the model based on settings of these model configuration parameters:

- **System target file**
- **Language**
- **Code interface packaging**

When you open one of the following coder apps, Simulink checks whether the system target file setting of the active configuration set aligns with your coder app selection, which can be one of these:

- **Simulink Coder** (GRT-based system target file)
- **Embedded Coder** (ERT-based system target file)
- **AUTOSAR Component Designer** (AUTOSAR system target file)

When Simulink detects a misalignment, the editor displays a message that identifies the current system target file setting and questions whether you want to change it and open the selected coder app. You can cancel the operation and open the app that aligns with the current system target file setting or continue with changing the system target file and open the app that you selected.

If the current system target file aligns with your coder app selection, Simulink opens the coder app and adds a code tab (for example, **C Code** or **AUTOSAR**) to the editor toolstrip.

After a coder app opens, on the toolstrip, for **Output**, select the C or C++ code setting or, for AUTOSAR, select one of the AUTOSAR platforms. Then, use the **Code Interface** menu to open the Code Mappings editor.

Create and Activate Code Mappings Programmatically

You can use the code mappings functions `coder.mapping.utils.create` and `coder.mapping.api.get` to create and activate code mappings for GRT- and ERT-based model platforms.

1. Set up multiple model configuration set objects (`Simulink.ConfigSet`) for the model. For example, you might set up GRT and ERT C code configurations or two ERT code configurations, one for generating reusable functions and one for generating nonreusable function code. Name the configuration sets and attach them to your model.

   ```
   grtConfigObj = Simulink.ConfigSet;
   set_param(grtConfigObj,'Name','grtConfig');
   attachConfigSet('myModel',grtConfigObj);

   ertConfigObj = Simulink.ConfigSet;
   set_param(ertConfigObj,'Name','ertConfig');
   attachConfigSet('myModel',ertConfigObj);
   ```

2. Activate and create a code mapping for one of the configuration sets by calling `setActiveConfigSet` and `coder.mapping.utils.create`.

   ```
   setActiveConfigSet('myModel','grtConfig');
   grt_cm = coder.mapping.utils.create('myModel');
   ```

3. Activate and create a code mapping for the second configuration set.

   ```
   setActiveConfigSet('myModel','ertConfig')
   ert_cm = coder.mapping.utils.create('myModel');
   ```

4. Activate the model code mappings by using calls to `coder.mapping.api.get` that specify the code mapping type, `SimulinkCoderC` or `EmbeddedCoderC`.

   ```
   grt_cm = coder.mapping.api.get('myModel','SimulinkCoderC');
   .
   .
   .
   # Configure model data elements
   .
   .
   ..
   ert_cm = coder.mapping.api.get('myModel','EmbeddedCoderC');
   ```

Code Mappings and Embedded Coder Dictionary

If you have Embedded Coder, you can use the Embedded Coder Dictionary to generate standardized code from multiple models. Because the definitions in the Embedded Coder Dictionary for a model are separate from your model elements, you can share the definitions by storing the Embedded Coder Dictionary in a Simulink data dictionary outside of your model. See “Separate C Code Interface Configuration Definition from Model-Specific Configurations” on page 32-38.

See Also

Code Mappings Editor – C | **Code Mappings – C++ Editor** | `Simulink.ConfigSet` | `setActiveConfigSet` | `coder.mapping.api.CodeMapping`

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- Code interface packaging
- “Manage Configuration Sets for a Model”

Configure Model Data Interface for C Code Generation

To integrate the generated code with external code or to generate code that complies with coding standards and guidelines, you can configure how data elements in a model, such as block parameters and signals lines, appear in the generated code. You can configure default code generation settings for categories of data elements and configure individual elements by using the Code Mappings editor or code mappings API. For example, you can configure settings that:

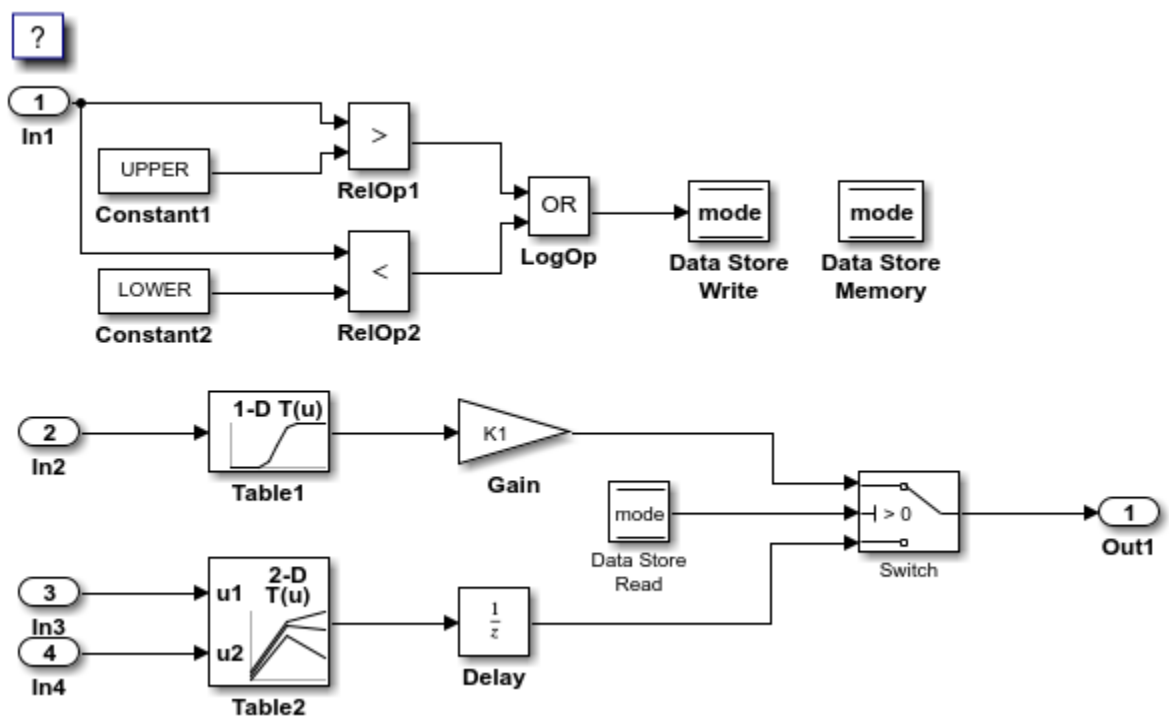
- Control placement of declarations and definitions in generated and external (exported and imported) files.
- Package multiple data items into structures.
- Apply storage type qualifiers `const` and `volatile`.

This example shows how to configure data elements for code generation for example model `rtwdemo_configinterface`.

Explore Example Model

Open example model `rtwdemo_configinterface.`

```
model = 'rtwdemo_configinterface';
open_system(model)
```



The example shows how to configure these model data elements for code generation:

- `Inport` blocks `In1`, `In2`, `In3`, and `In4`
- Outport block `Out1`

- Model parameter arguments `LOWER` and `UPPER`
- Model parameters `K1`, `Table1`, and `Table2`
- Local data store `mode`
- Signal lines sourced from lookup table blocks `Table1` and `Table2`
- State `X` (delay)

Configure Default Settings for `Inport` Blocks

1. Open the **Embedded Coder** app.
2. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
3. In the Code Mappings editor, under **Inports and Outports**, select category **Inports**. Set the default storage class to `ImportFromFile`.
4. In the Property Inspector, set **Header File** to `exInDataLut.h`.

Configure Default Settings for Outport Block

1. On the **Data Defaults** tab, under **Inports and Outports**, select category **Outports**. Set the default storage class to `ExportToFile`.
2. In the Property Inspector, set **Header File** to `exOutSys.h` and **Definition File** to `exOutSys.c`.

Configure Default Settings for Parameters

1. On the **Data Defaults** tab, under **Parameters**, select category **Model parameter arguments**. Leave the storage class set to `Default`. With that setting, the code generator allocates a separate area of memory for each argument instance.
2. Select category **Model parameters**. Link text `'Auto' will be inlined` indicates that the code generator is configured to inline model parameters by default. For this example, configure the model parameters to be tunable. Click `'Auto' will be inlined`. The Model Configuration Parameters dialog box opens.
3. Set model configuration parameter **Default parameter behavior** to `Tunable`. Save the change and close the dialog box. In the Code Mappings editor, the link text changes to `'Auto' will be tunable`.
4. In the Code Mappings editor, with the **Model parameters** category selected, set the storage class to `ConstVolatile`.

Configure Default Settings for Signals, States, and Data Stores

1. On the **Data Defaults** tab, under **Signals**, select category **Signals,states, and internal data**. Set the default storage class to FileScope.
2. Configure default code mappings for shared local data stores. Select category **Shared local data stores**. Set the storage class to `Volatile`.
3. Save the model.

Configure Code Generation Settings for Individual Inport Blocks

1. In the Code Mappings editor, click the **Inports** tab. The editor lists the names of root-level `Inport` and `In Bus Element` blocks that are in the model. If a port resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name. The storage class for each inport is set to `Auto`, which means that the code generator might eliminate relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model

default configuration. For this example, the model default configuration specifies storage class `ImportedFromFile`. To avoid avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default`. To override the default configuration, specify the storage class that meets the code generation requirements for that inport.

2. Configure the code generator to apply the default storage class setting to inports `In2`, `In3`, and `In4`. Select the rows for the three inports. Then, for one of the selected inports, set the storage class to `Model default: ImportFromFile`. The storage class for the three selected inports changes to Model default: ImportFromFile.
3. For inport `In1`, override the default storage class setting. Set the storage class to `ImportFromFile`. In the Property Inspector, set **Header File** to `exInDataMem.h`.
4. Configure code identifiers for the inports so that the variables in the generated code match the interface names in the external header and definition files. In the Code Mappings editor, select the row for each inport. In the Property Inspector, set the **Identifier** property to these values: `input1`, `input2`, `input3`, and `input4`.

Configure Code Generation Settings for Individual Outport Blocks

1. In the Code Mappings editor, click the **Outports** tab. The editor lists the names of root-level `Outport` and `Out Bus Element` blocks that are in the model. If a port resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name. The storage class for the outport in the example model is set to `Auto`, which means that the code generator might eliminate relevant code for optimization purposes.
2. To avoid optimizations and force the code generator to use the default configuration, set the storage class to `Model default: ExportToFile`. The storage class for the selected outport changes to `Model default: ExportToFile`.
3. Configure the code identifier for the outport so that the variable in the generated code matches the interface name used by the external code. In the Code Mappings editor, select the row for the outport. In the Property Inspector, set the **Identifier** property to `output`.

Configure Code Generation Settings for Individual Parameters

1. In the Code Mappings editor, click the **Parameters** tab. Expand **Model Parameter Arguments**. By default, the storage class for each model parameter argument is set to `Auto`, which means that the code generator might eliminate relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration. For this example, leave the storage class set to `Auto`.
2. Expand **Model Parameters**. By default, the storage class for each model parameter is set to `Auto`. For this example, you want to avoid optimizations and for the code generator to use the default storage class setting, ConstVolatile.
3. In the Code Mappings editor, under **Model Parameters**, select parameters `K1`, `Table1`, and `Table2`. Set the storage class to `Model default: ConstVolatile`.
4. Configure the code identifiers for model parameters with names that include the prefix `mp_`, as the requirements specify. In the Code Mappings editor, select model parameter `K1`. In the Property Inspector, expand the **Code** node. Set the storage class property **Identifier** to `mp_K1`. For parameters `Table1` and `Table2`, set **Identifier** to `mp_Table1` and `mp_Table2`.

Configure Code Generation Settings for Individual Data Stores

1. In the Code Mappings editor, click the **Data Stores** tab. Expand **Local Data Stores**. The storage class for data store mode is set to Auto, which means that the code generator might eliminate

relevant code for optimization purposes. For this example, you want to avoid optimizations and for the code generator to use the default storage class setting, `Volatile`.

2. In the Code Mappings editor, select local data store `mode`. Set the storage class to `Model default: Volatile`.
3. Configure the code identifier for the data store with a name that includes the prefix `ds_`. In the Code Mappings editor, select shared local data store `mode`. In the Property Inspector, expand the **Code** node. Set the storage class property **Identifier** to `ds_mode`.

Configure Code Generation Settings for Individual States

1. In the Code Mappings editor, click the **Signals/States** tab. Expand **States**. The storage class for the state is set to `Auto`, which means that the code generator might eliminate relevant code for optimization purposes. For this example, you want to avoid optimizations and for the code generator to use the default storage class setting, `FileScope`.
2. In the Code Mappings editor, select state `X`. Set the storage class to `Model default: FileScope`.
3. Configure the code identifier for the state with a name that includes the prefix `dstate_`, as the requirements specify. In the Code Mappings editor, select state `X`. In the Property Inspector, expand the **Code** node. Then, set the storage class property **Identifier** to `dstate_X`.

Configure Code Generation Settings for Individual Signals

Add signals that you want to configure to the model code mappings. For this example, add to the model code mappings the output signals for the two lookup table blocks. Then, configure those signals.

1. In the Code Mappings editor, click the **Signals/States** tab. No signals are listed.
2. Add signals to the code mappings. For the output signals of lookup table blocks `Table1` and `Table2`, in the model, select a signal, pause on the ellipsis that appears above or below the signal line to open the action bar, and click the **Add Signal** button. In the Code Mappings editor, the **Signals** node expands and lists the names or block identifiers of the two signals that you added. If a signal resolves to a signal object, a resolve-to-signal-object icon appears to the right of the element name or port identifier. The storage class for each signal is set to `Auto`, which means that the code generator might eliminate relevant code for optimization purposes. For this example, you want to avoid optimizations and for the code generator to use the default storage class setting, `FileScope`.
3. In the Code Mappings editor, select the output signals for blocks `Table1` and `Table2`. Set the storage class to `Model default: FileScope`.
4. Configure the code identifier for the output signals for the two lookup table blocks with names that include the prefix `dout`. In the Code Mappings editor, select signal `Table1:1`. In the Property Inspector, expand the **Code** node. Set the storage class property **Identifier** to `dout_Table1`. For signal `Table2:1`, set **Identifier** to `dout_Table2`.
5. Save the model.

Configure Data for Code Generation Programmatically

Alternatively, use the code mappings API to configure model data elements programmatically.

1. Get the object that represents the code mappings for the model by calling `coder.mapping.api.get`. You specify the returned object as the first argument in subsequent calls to code mapping functions.

```
cm = coder.mapping.api.get(model);
```

2. Configure default code generation settings for categories of data elements with calls to `setDataDefaults`. In each call, include the object returned by the call to `coder.mapping.api.get`, the name of a data element category, and name-value pair arguments that specify the storage class and storage class properties. Use `setDataDefaults` to configure these default settings:

- `Inport` blocks with storage class `ImportFromFile` and header file `exInDataLut.h`
- Outport blocks with storage class ExportTo`File` and header file `exOutSys.h` and definition file `exOutSys.c`
- Model parameter arguments with the code generator default configuration
- Model parameters with storage class `ConstVolatile`
- Internal data (signals, states, and local data stores) with storage class `FileScope`
- Shared locoal data stores with storage class `Volatile`

```
setDataDefault(cm,'Inports','StorageClass','ImportFromFile','HeaderFile','exInDataLut.h');
setDataDefault(cm,'Outports','StorageClass','ExportToFile','HeaderFile','exOutSys.h',...
'DefinitionFile','exOutSys.c');
setDataDefault(cm,'ModelParameterArguments','StorageClass','Default');
setDataDefault(cm,'ModelParameters','StorageClass','ConstVolatile');
setDataDefault(cm,'InternalData','StorageClass','FileScope');
setDataDefault(cm,'SharedLocalDataStore','StorageClass','Volatile');
```

Set model configuration parameter `DefaultParameterBehavior` to `Tunable`, so that model parameters are accessible during program exeuction.

```
set_param(model,'DefaultParameterBehavior','Tunable');
```

3. Configure individual `Inport` blocks with calls to `setInport`. To avoid optimizations, apply the default configuration for `Inport` blocks to `In2`, `In3`, and `In4`. For `In1`, override the default configuration, by setting the storage class to `ImportFromFile` and the header file to `exInDataMem.h`. Also, configure code identifiers for the inports so that the global variables that represent inports in the generated code match the variable names in the external definition file. In each call to `setInport`, specify the object returned by `coder.mapping.api.get`, the `Inport` block name, and name-value pair arguments for storage class and property settings.

```
setInport(cm,'In1','StorageClass','ImportFromFile','HeaderFile','exInDataMem.h',...
    'Identifier','Input1');
setInport(cm,'In2','StorageClass','Model default','Identifier','Input2');
setInport(cm,'In3','StorageClass','Model default','Identifier','Input3');
setInport(cm,'In4','StorageClass','Model default','Identifier','Input4');
```

4. Configure individual data element settings for Outport block `Out1` with a call to `setOutport`.To avoid optimizations, apply the default configuration. Also, configure a code identifer for the outport. In the call to `setOutport`, specify the object returned by `coder.mapping.api.get`, the Outport block name, and name-value pair arguments for the storage class and code identifer property settings.

```
setOutport(cm,'Out1','StorageClass','Model default','Identifier','output');
```

5. Configure individual model parameters with calls to `setModelParameter`.To avoid optimizations, apply the default configuration. Also, configure code identifers for the parameters. In each call to

`setModelParameter`, specify the object returned by `coder.mapping.api.get`, the parameter name, and name-value pair arguments for the storage class and code identifer property settings.

```
setModelParameter(cm,'K1','StorageClass','Model default','Identifier','mp_K1');
setModelParameter(cm,'Table1','StorageClass','Model default','Identifier','mp_Table1');
setModelParameter(cm,'Table2','StorageClass','Model default','Identifier','mp_Table2');
```

6. Configure individual data element settings for local data store `mode` with a call to `setDataStore`.To avoid optimizations, apply the default configuration. Also, configure a code identifer for the data store. In the call to `setDataStore`, specify the object returned by `coder.mapping.api.get`, the data store name, and name-value pair arguments for the storage class and code identifer property settings.

```
setDataStore(cm,'mode','StorageClass','Model default','Identifier','ds_mode');
```

7. Configure individual data element settings for state X with a call to `setState`.To avoid optimizations, apply the default configuration. Also, configure a code identifer for the state. In the call to `setState`, specify the object returned by `coder.mapping.api.get`, the path for the block that uses the state, and name-value pair arguments for the storage class and code identifer property settings.

```
setState(cm,'rtwdemo_configinterface/Delay','StorageClass','Model default',...
    'Identifier','dstate_X');
```

8. Add signals to the code mappings and configure them with calls to `setSignal`.To add the port handles to add the signal data to the model code mappings with calls to `get_param`. Then, specify the block handles in a call to `addSignals`. To avoid optimizations, apply your default configuration for signals. Also, configure code identifers for the signals. In each call to `setSignal`, specify the object returned by `coder.mapping.api.get`, the path or block handle of the signal source, and name-value pair arguments for the storage class and code identifer property settings.

```
lut1D_ports = get_param('rtwdemo_configinterface/Table1','PortHandles');
lut2D_ports = get_param('rtwdemo_configinterface/Table2','PortHandles');

lut1D_outPort = lut1D_ports.Outport;
lut2D_outPort = lut2D_ports.Outport;

addSignal(cm,[lut1D_outPort,lut2D_outPort]);

setSignal(cm,lut1D_outPort,'StorageClass','Model default','Identifier','dout_Table1');
setSignal(cm,lut2D_outPort,'StorageClass','Model default','Identifier','dout_Table2');
```

Generate and View Code

1. Generate code.

```
slbuild(model)

### Starting build procedure for: rtwdemo_configinterface
### Successful completion of code generation for: rtwdemo_configinterface

Build Summary

Top model targets built:

Model                     Action           Rebuild Reason
==========================================================================================
```

```
rtwdemo_configinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.84s
```

2. View the code. For example, in rtwdemo_configinterface.c, find where these variables are used in the step entry-point function: `input1`, `input2`, `input3`, `input4`, `output`, `mp_K1`, `ds_mode`, `dstate_X`, `dout_Table1`, and `dout_Table2`.

Related Topics

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Root-Level Inport Blocks for C Code Generation” on page 33-30
- “Configure Root-Level Outport Blocks for C Code Generation” on page 33-41
- “Configure Signal Data for C Code Generation” on page 33-50
- “Configure Parameters for C Code Generation” on page 33-61
- “Configure Block States for C Code Generation” on page 33-74
- “Configure Data Stores for C Code Generation” on page 33-82
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- Code Mappings Editor – C
- Embedded Coder Dictionary

Migration of Model Data Configurations to Code Mappings

Starting in R2020b, the recommended approach to configure model data for code generation is to use the Code Mappings editor or code mappings API. From one interface, you can configure default settings for categories of data, and then override those settings, as needed, for specific data elements. For parameters, you can specify whether they are inlined or tunable by default. When producing code for data, the code generator uses the code definitions that you specify to determine how to represent the data in the generated code.

When you open a model created in a release before R2020b, Simulink migrates code definitions specified in the model to the model code mappings.

If necessary, as part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses. When you save the model, the code mappings are saved as part of the model.

Effects of Migration

For data configured within a model, the migration process replaces existing storage class specifications with equivalent code mappings. Data configured for code generation within a model includes data represented by:

- Outport blocks
- Signal lines
- Block states
- Data stores
- Parameter objects in the model workspace

This table lists specific model conditions and their corresponding migration results.

Condition	Migration Results
Model is a library model or a model configured with an AUTOSAR system target file.	Simulink discards the code configuration associated with data configured within the model.
Data element is configured for code generation within the model and the data element is associated with a `Simulink.CoderInfo` object, the `StorageClass` property for that object is set to `Custom`, and the `CustomStorageClass` property is set to `Default`.	Simulink changes the storage class setting from `Custom (Default)` to `ExportedGlobal` and displays a warning.

Condition	Migration Results
Model uses data objects configured for use with the `TypeQualifier` storage class property for a built-in storage class, such as `ExportedGlobal` or `ImportedExtern`. You might have set the property as: • `CoderInfo.TypeQualifier` property for data objects in a workspace or data dictionary • Port parameter `RTWStorageTypeQualifier` • Block parameter `RTWStateStorageTypeQualifier` for Data Store Memory, Discrete Filter, Discrete State-Space, Discrete-Time Integrator, Discrete Transfer Fcn, Discrete Zero-Pole, and Memory blocks	Simulink issues a warning indicating that the property is no longer supported. Other interfaces are available for achieving the same results.
Model uses the `Simulink.CoderInfo` object `Alignment` property for data configured within the model.	Simulink issues a warning indicating that the property is no longer supported for model-specific data. You can use the property for data stored in the base workspace or a data dictionary.
You initialize the model workspace for the model from an external data source, such as a MAT-file, MATLAB file, or data dictionary.	Simulink moves code configuration for the parameter object to the code mappings for that model.
For an individual subsystem from which you generate code and executable programs by right-clicking the Subsystem block, you configure the storage class and storage class properties for the subsystem input and output signals outside of the subsystem.	Generated code does not include variables for the input and output interface data.

If you perform these operations, you might need to take action:

- Use the Simulink Editor to copy a block line to another model. Simulink does not copy the code configuration information associated with the copied modeling element. For more information, see “Copy Blocks and Signal Lines Between Models” on page 33-107.
- Switch the system target file setting for a model between a GRT-based and ERT-based system target file. Simulink ignores the code configuration information for the system target file of the source version of the copied model. For more information, see “Switch Between GRT- and ERT-Based System Target Files” on page 33-108.

Considerations Before Migrating

Before migration, consider the preparation suggested in the following sections if these conditions exist:

- Data configured within a library or AUTOSAR model
- Storage class `TypeQualifier` property
- Data alignment for code generation

- Code configuration for parameter objects initialized in the model workspace
- Code configuration for subsystem I/O interface

Data Configured Within Library or AUTOSAR Model

If your model is a library model or a model configured with an AUTOSAR system target file, reconfigure code generation for data. For library models, reconfigure code generation for data in the context of models that use the library (see “Code Interface Configuration”). For AUTOSAR models, see “Map AUTOSAR Elements for Code Generation” (AUTOSAR Blockset).

To avoid losing the code configuration for data configured within the model, in an earlier release, load the model and create and run a MATLAB script that:

1. Loads the model data.
2. Finds data configured for code generation within the model.
3. Converts the representation of the data to data objects that you store in an external source, such as a MAT-file, MATLAB file, or data dictionary.
4. Updates the model to use the external data objects.

Storage Class TypeQualifier Property

If your model configures data objects with built-in storage classes, such as `ExportedGlobal` and `ImportedExtern`, and uses the storage class `TypeQualifier` property, when you load the model in release R2020b or later, Simulink issues a warning. To eliminate the warning and update a model to use a supported alternative, in an earlier release, load the model and create and run a MATLAB script that:

1. Loads the model data objects from the model workspace, base workspace, or data dictionary.
2. Finds data objects with the `TypeQualifier` property set to a non-empty string value.
3. Changes the storage class setting to a storage class predefined with the required type qualifier (for example, storage class `Const` includes qualifier `const` in data declarations and definitions).

This example script loads model data objects from the base workspace, finds instances of data configured with the storage class `TypeQualifier` property, and changes the storage class setting to `Const`.

```
vars = who;

  for idx = 1:length(vars)
    thisVar = evalin('base',vars{idx});
    if (isa(thisVar,'Simulink.Parameter') && (isequal(thisVar.CoderInfo.TypeQualifier, 'const')))
      thisVar.CoderInfo.TypeQualifier = '';
      thisVar.CoderInfo.StorageClass = 'Custom';
      thisVar.CoderInfo.CustomStorageClass = 'Const';
    end
  end
```

Starting in R2020b, to include type qualifiers in the generated code, use the Code Mappings editor or code mappings API to associate data with a storage class that specifies use of the `const` or `volatile` keyword (see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75). If none of the available storage classes meets your application requirements and you are using Embedded Coder, define a new storage class by using the Embedded Coder Dictionary (see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2). Then, map the model data to the new storage class.

Data Alignment for Code Generation

To preserve data alignment information configured for a model, represent data with data objects outside of the model. For example, open the model in a previous release of Simulink and perform the actions listed in this table.

Type of Data Element	Action
Parameter object in the model workspace	Use the Model Explorer to move the object from the model workspace to the base workspace or a data dictionary. Then, set the storage class and alignment properties of the object. Save the model.
Signal line	Open the Signal Properties dialog box, specify a name for the signal and select **Signal name must resolve to Simulink signal object** (storage class must be `Auto`). Create a `Simulink.Signal` object in the base workspace or a data dictionary and set the storage class and alignment properties of the object. Save the model.
Block state	Open the Block Parameters dialog box, specify a name for the state and select **State name must resolve to Simulink signal object** (storage class must be `Auto`). Create a `Simulink.Signal` object in the base workspace or a data dictionary and set the storage class and alignment properties. Save the model.

Alternatively, create and run a MATLAB script that:

1. Loads the model data.
2. Finds data objects that set `Simulink.CoderInfo.Alignment`.
3. Converts the representation of the data to data objects that you store in an external source, such as a MAT-file, MATLAB file, or data dictionary.
4. Sets `Simulink.CoderInfo` object property `Alignment` for the new data objects.

For example, this script checks whether `Alignment` is set for parameter, lookup table, and breakpoint data objects and captures the data objects found in MAT-file `savedData.mat`.

```
filename = 'savedData.mat';
varList = who;
tosaveVars = {};
for idx = 1 : length(varList)
   variable = eval(varList{idx});
   if ((isa(variable, 'Simulink.Parameter') || ...
       isa(variable, 'Simulink.LookupTable') || ...
       isa(variable, 'Simulink.Breakpoint')) && ...
       variable.CoderInfo.Alignment ~= -1)
       tosaveVars{end+1} = varList{idx};
   end
end
if ~isempty(tosaveVars)
  save('savedData.mat', tosaveVars{:});
end
```

Evaluate the script in the model workspace with these commands:

```
mws = get_param(bdroot, 'ModelWorkspace');
evalin(mws, 'SaveMWSParamsWithAlignment');
```

For parameter objects in the model workspace, alternatively, you can use the Model Explorer interface to move the objects from the model workspace to the base workspace or a data dictionary. Then, select the **Alignment** property for the storage class.

Code Configuration for Parameter Objects Initialized in Model Workspace

After loading a model in release R2020b or later and initializing the model workspace for the model from an external data source, such as a MAT-file, MATLAB file, or data dictionary, Simulink:

- Saves the contents of the `Simulink.CoderInfo` object to the model code mappings.
- Removes the `Simulink.CoderInfo` object from the `Simulink.Parameter` object defined in the external data source.

Use the Code Mappings editor or the code mappings API to get and set parameter code configuration settings. See “Code Interface Configuration”.

Code Configuration for Subsystem I/O Interface

In release R2020b or later, for individual subsystems from which you generate code and executable programs by right-clicking the Subsystem block, to include signal data in the generated subsystem code interface, configure the storage class and storage class properties for the subsystem input and output signals within the subsystem. If you configure the signals outside of the subsystem, the generated code does not include variables for the input and output interface data.

For more information, see “Generate Code and Executables for Individual Subsystems” on page 4-2.

Copy Blocks and Signal Lines Between Models

Starting in R2020b, when you use the Simulink Editor to copy a block or signal line to another model, Simulink does not copy the code configuration information associated with the copied modeling element.

To work around this change, copy and paste the code configuration information for data elements by using the code mappings API.

1. Open the source and destination models for the copy operation.
2. Use the code mappings API to get the code mappings object for the source model.
3. Use the code mappings API to get code mapping information for the model element that you are copying.
4. If a code mappings object does not exist for the destination model, use the code mappings API to create one.
5. For the model element that you are copying to the destination model, use the code mappings API to set code mapping information to the same values as for the source model.
6. Close the source model.
7. Save and close the destination model.

For example:

```
.
.
```

```
.
srcMdl = 'my_src_mdl';
destMdl = 'my_dest_mdl';

open_system(srcMdl);
open_system(destMdl);

% Get data configuration from source model
srcCM = coder.mapping.api.get(srcMdl);
srcSC = getSignal(srcCM, srcPort, 'StorageClass')

% Set data configuration for destination model to setting of source model
destCM = coder.mapping.api.get(destMdl);
setSignal(destCM, destPort, 'StorageClass', srcSC);

close_system(srcMdl);
save_system(destMdl);
close_system(destMdl);
.
.
.
```

Switch Between GRT- and ERT-Based System Target Files

Starting in R2020b, when you switch the system target file setting for a model between a GRT-based and ERT-based system target file, Simulink applies alternative code configurations for each system target file. This switch can occur if you:

- Alternate between the Simulink Coder and Embedded Coder apps.
- Change the active configuration set for a model.
- Change the setting of model configuration parameter **System target file**.

To preserve the code configuration information for data configured within the model, create and run a MATLAB script that uses the code mappings API to copy relevant code mappings.

1. Open the model with the source system target file and code mappings that you want to preserve.
2. Use the code mappings API to get the code mappings object for the model and get data mappings objects for the data elements.
3. Use the code mappings API to get the code mappings object for the model and set data mappings objects for the data elements.
4. Save and close the model.

This script fragment shows how to use the code mappings API to preserve the storage class of a signal object when switching from a GRT system target file to a an ERT system target file.

```
.
.
.
mymdl = 'my_shared_mdl';

open_system(mymdl);

% Get data configuration for Simulink Coder (GRT-based) code generation
scCodeMapping = coder.mapping.api.get(model, 'SimulinkCoderC');
scStorageClass = getSignal(scCodeMapping, srcPort, 'StorageClass')

% Set data configuration for Embedded Coder (ERT-based) code generation
ecCodeMapping = coder.mapping.api.get(model, 'EmbeddedCoderC');
```

```
setSignal(ecCodeMapping, srcPort, 'StorageClass', scStorageClass)

save_system(mymdl);
close_system(mymdl);

.
.
.
```

Workflow Changes

Starting in R2020b, configure model data elements for code generation by using the Code Mappings editor or code mappings API instead of or in addition to creating and configuring Simulink data objects. Use this workflow:

1. Open the **Embedded Coder** app.
2. In the Code Mappings editor, configure default behavior for storage representation. This capability was introduced in R2018a. See “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15..
3. Decide whether to override default configuration settings for individual data elements. If you choose not to override settings, skip to step 8.
4. Identify MATLAB variables that you want to configure individually and store the configurations in the model workspace. In the Model Explorer, convert the variables to `Simulink.Parameter` objects. The resulting data objects appear in the Code Mappings editor as model parameters.
5. Identify signals representing algorithm data that you want to monitor while the generated code executes and configure individually. For each signal that you identify, do one of the following:
   - For signal data that is specific to a model (not shared), add the signals to the Code Mappings editor. In the model, pause on the ellipsis above a signal line to open the action bar. Click the **Add Signal** button to add the signal to the Code Mappings editor.
   - For shared signal data, create and configure `Simulink.Signal` objects. See “C Code Generation Configuration for Model Interface Elements” on page 33-2 and “Control Signal and State Initialization in the Generated Code” on page 36-176.
6. Configure individual data elements. In the Code Mappings editor, click a category tab, select a data element, and set the storage class. In the Property Inspector, set storage class property values. To change the setting of model configuration parameter **Default parameter behavior** click the link indicating the behavior when the storage class is set to `Auto`. In the Model Configuration Parameters dialog box, change the parameter setting to `Tunable` or `Inline`.
7. To view external data objects that are stored in the base workspace or a data dictionary, in the Code Mappings editor, click the Code Mappings editor **Refresh** button.
8. Generate and review code.

See Also

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

- “Control Data and Function Interface in Generated Code” on page 36-44
- “Control Signal and State Initialization in the Generated Code” on page 36-176

Dimension Preservation of Multidimensional Arrays

Dimension Preservation in Generated Code

The code generator produces one-dimensional arrays in the C/C++ code for multidimensional model data. For example, consider matrix `matrixParam`.

```
matrixParam =
        1        2        3
        4        5        6
```

By default, MATLAB stores data in column-major format. The code generator defines matrix `matrixParam` in the generated code as:

```
/* const_params.c */
matrixParam[6] = {1, 4, 2, 5, 3, 6};
```

The code generator initializes matrix `matrixParam` in the generated code as:

```
/* model.c */
extern const real_T matrixParam[6];

for(int i = 0; i < 6; i++) {
    ... = matrixParamValue[i];
}
```

When you set the model configuration parameter **Array layout** to `Row-major`, you can preserve dimensions of multidimensional arrays in the generated code. Preserving array dimensions in generated code enhances integration with external code.

For example, in row-major array layout, the code generator preserves the array dimensions in type definition in the generated code as:

```
/* const_params.c */
const real_T matrixParam[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

The code for initializing the multidimensional data is nested as:

```
/* model.c */
extern const real_T matrixParam[2][3];

for(int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
         ... = matrixParam[i][j];
    }
}
```

Difference in Dimension Preservation Between MATLAB and C

When you define a multidimensional array in the MATLAB Command Window, MATLAB stores the array in column-major format. For example:

```
>> matrixParam.Value = [1 2 3; 4 5 6];  % Data inilitialized in row-major

>> matrixParam.Value                    % Data displayed in row-major
ans =
     1     2     3
     4     5     6
```

```
>> matrixParam.Value(:)'                % Data stored in column-major
ans =
     1     4     2     5     3     6
```

Even if the data is initialized in row-major format in the MATLAB Command Window, MATLAB serializes data in memory in column-major format.

When the code generator preserves dimensions, the generated code is:

```
real_T matrixParam[2][3] = { { 1.0, 2.0, 3.0 }, { 4.0, 5.0, 6.0 } } ;
```

For 2-D arrays, the index order and data definition are consistent with what you see in the MATLAB Command Window and the generated code.

For n-D arrays, when the code generator produces row-major code and preserves array dimensions, the index order is consistent with MATLAB. The data definition of n-D arrays in the generated code is not consistent with the data displayed in the MATLAB Command Window. For example, consider that `matrixParam` is a `Simulink.Parameter`:

```
>> matrixParam.Value = reshape(1:24,[4 3 2])

>> matrixParam.Value
ans(:,:,1) =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
ans(:,:,2) =
    13    17    21
    14    18    22
    15    19    23
    16    20    24

>> matrixParam.Value(:)'
ans =

  Columns 1 through 10

     1     2     3     4     5     6     7     8     9    10

  Columns 11 through 20

    11    12    13    14    15    16    17    18    19    20

  Columns 21 through 24

    21    22    23    24
```

In row-major array layout, the code generator preserves the array dimensions during type definition in the generated code as:

```
/* const_params.c */
const real_T matrixParam[4][3][2] = { { { 1.0, 13.0 }, { 5.0, 17.0 }, { 9.0, 21.0 } },
  { { 2.0, 14.0 }, { 6.0, 18.0 }, { 10.0, 22.0 } }, { { 3.0, 15.0 }, { 7.0, 19.0
    }, { 11.0, 23.0 } }, { { 4.0, 16.0 }, { 8.0, 20.0 }, { 12.0, 24.0 } } };
```

The code for initializing the multidimensional data is nested as:

```
/* model.c */
extern const real_T matrixParam[4][3][2];

for (i = 0; i < 3; i++) {
    for (i_1 = 0; i_1 < 4; i_1++) {
      for (i_0 = 0; i_0 < 2; i_0++) {
        ... = ... + matrixParam[i_1][i][i_0];
      }
    }
  }
```

See Also

Related Examples

- “Preserve Dimensions of Multidimensional Arrays in Generated Code” on page 47-2
- “Preserve Dimensions of Bus Elements in Generated Code” on page 47-6
- “Code Generation of Matrices and Arrays” on page 52-63
- “Select Array Layout for Matrices in Generated Code” (Stateflow)

Configure Generated C Function Interface for Model Entry-Point Functions

What Is an Entry-Point Function?

An entry point is a location in code where a transfer of program control (execution) occurs. The main function (`main()`) is the entry point to a C/C++ program and is called when the application starts executing. Calls to other functions, for example from the `main` function, provide entry points to function code. Program control is transferred to the called function. The function code executes, and when finished, returns control to the `main` or other calling function.

When producing code for a model, the code generator defines a set of entry-point functions that you can call to execute the generated code. You can call the generated functions from external code or from a version of the generated main function that you modify.

The Code Interface Report section of the code generation report lists the entry-point functions that the code generator produces for a model. For more information, see “Analyze Generated Data Code Interface” on page 54-13.

Types of Generated Entry-Point Functions

The code generator produces entry-point functions for these types of model functions.

Type of Model Function	Model Source Name	Default Generated Function Name	Description
Exported function (requires Embedded Coder)	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model	*`function-call-inport-block-name`* or *`signal-label`* (if specified)	For an exported-function model, the exported function for a subsystem.
Initialize function	`Initialize`	*`model`*`_initialize`	Initialization code for a model. At the start of the application code, call the function *once*. Do not use this function to reset the real-time model data structure (`rtM`).

Type of Model Function	Model Source Name	Default Generated Function Name	Description
Partition function	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink® Schedule Editor (for example, P1), or a task name in the Concurrent Execution dialog	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods	For a model partition, output and update code. Model configuration parameter **Single output/update function** is selected (the default).
Partition update function	`PartitionUpdate:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink® Schedule Editor (for example, P1), or a task name in the Concurrent Execution dialog	*`model`*`_output`*`n`* and *`model`*`_update`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods	For a model partition, output and update code in separate functions. Model configuration parameter **Single output/update function** is cleared.
Periodic multitasking function	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multitasking model (for example, D1)	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods	For blocks in a rate-based model configured for multitasking, output and update code. The code generator produces a function for each sample period. Model configuration parameter **Single output/update function** is selected (the default).
Periodic multitasking update function	`PeriodicUpdate:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multitasking model (for example, D1)	*`model`*`_output`*`n`* and *`model`*`_update`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods	For blocks in a rate-based model configured for multitasking, output and update code in separate functions. The code generator produces output and update functions for each sample period. Model configuration parameter **Single output/update function** is cleared.

Type of Model Function	Model Source Name	Default Generated Function Name	Description
Periodic single-tasking function	`Periodic`	*`model`*`_step`	For blocks in a rate-based model configured for single-tasking, output and update code. Model configuration parameter **Single output/update function** is selected (the default).
Periodic single-tasking update function	`PeriodicUpdate`	*`model`*`_output` and *`model`*`_update`	For blocks in a rate-based model configured for single-tasking, output and update code in separate functions. Model configuration parameter **Single output/update function** is cleared.
Reset function	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model	*`model_reset-function-name`*	If the model includes a Reset Function block, reset code generated. To reset conditions or state, call the function from the application code.
Simulink Function	`Simulink Function:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink Function block in the model	*`function-name`* for a global Simulink Function block or *`model_function-name`* for a scoped Simulink Function block	For an exported-function model, the exported function for a Simulink Function block.
Terminate function	`Terminate`	*`model`*`_terminate`	Code for turning off a system. For ERT-based models, you can suppress generation of this function by clearing the model configuration parameter **Terminate function required** (set by default).

If you have Embedded Coder software, to facilitate integration of external and generated code and achieve compliance with code standards and guidelines, you can configure how the code generator produces function interfaces from a model or subsystem.

Configure Whether Entry-Point Functions Are Reusable

By default, for top models, the code generator produces code that is not reusable or reentrant. Entry-point functions have a `void-void` interface. Code communicates with other code by directly accessing global data structures that reside in shared memory.

If your application requires reusable, multi-instance entry-point function code, you can configure the code generator to call each function (instance) with unique data. In this case, the code is reentrant.

You configure whether entry-point functions are reusable with the model configuration parameter **Code interface packaging** and related parameters. The parameter settings that you choose depend on factors such as configuration selections for the system target file, programming language, and argument interface.

Default Configurations for Single-Instance C Entry-Point Functions

By default, for GRT- and ERT-based system target files, the code generator produces single-instance C entry-point functions. The generated code:

- Creates an execution function without arguments (`void-void`).
- Allocates memory statically (at compile time) for model data structures.

The default model configuration parameter settings for configuring single-instance entry-point function code are:

- **Language** set to `C`.
- **Code interface packaging** set to `Nonreusable function`.

Generate Reusable, Multi-Instance C Entry-Point Functions

You can configure the code generator to produce reusable entry-point functions in C for either a GRT- or ERT-based system target file. However, the function interfaces that the code generator produces by default varies. Assuming that model configuration parameter **Language** is set to `C` and **Code interface packaging** is set to `Reusable function`, the code generator produces this entry-point function code for each system target file scenario.

System Target File	Interface
GRT-based	• Reusable, multi-instance C entry-point functions that are reentrant. • Packs values of model root-level Inport blocks and Outport blocks into the real-time model data structure. Passes that structure to the execution function as an argument by reference. • Allocates memory dynamically at runtime for the data of a model instance. Allocates the memory by calling a function, such as `malloc`.
ERT-based	• Reusable, multi-instance C entry-point functions that are reentrant. • Passes the value of each model root-level Inport block and Outport block to the execution function as a separate argument. • Allocates memory statically for model data structures.

If you are using an ERT-based system target file and want to generate reusable, multi-instance C entry-point functions that are reentrant, consider:

- Using dynamic memory allocation to initialize model data structures. Select **Use dynamic memory allocation for model initialization**.
- Packing values of model root-level Inport blocks into a structure, packing values of root-level Outport blocks into a second structure, and passing the structures to the execution function as arguments by reference. Set **Pass root-level I/O as** to `Structure reference`.
- Packing values of model root-level Inport blocks and Outport blocks into the real-time model data structure and passing that structure to the execution function as an argument by reference. Set **Pass root-level I/O as** to `Part of model data structure`.

Choose an Approach for Configuring Generated C Function Interfaces

To facilitate integration of external and generated code and achieve compliance with code standards and guidelines, you can use the Code Mappings editor or code mappings API to configure how the code generator produces function interfaces from a model or subsystem.

For models configured with a service interface, you can configure:

- Function name
- Function arguments for functions in subcomponents and functions in component models that are represented as Simulink Function blocks
- Memory section

Configuration	More Information
Default naming rules for categories of functions (initialize/terminate, execution, and shared utility) across a model	“Configure Default Code Generation for Functions” on page 33-20
Name for individual entry-point functions (override the default naming rule)	“Configure Names for Individual C Entry-Point Functions” on page 33-119
Step function interface (function name, return value, and argument C type qualifiers, names, and order) for the base rate step entry-point function interactively	“Configure Name and Arguments for Individual Step Functions” on page 33-120
Function interfaces (function name, return value, and argument C type qualifiers, names, and order) for Simulink Function and Function Caller blocks interactively	“Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2

You can use software-in-the-loop (SIL) testing to verify code generated for customized entry-point functions. Create a SIL block by using your generated code. Then, integrate the SIL block into a model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see “Choose a SIL or PIL Approach” on page 88-11.

Configure Default Code Generation Settings for a Function Category

Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of functions across a model. For example, you can configure function name rules and placement of function code in memory sections. Applying default configurations can save time and reduce the risk of introducing errors in code, especially for models that have a large number functions.

The code generator produces entry-point functions for these categories of model functions:

Function Category	Description
Initialize/Terminate	Functions that initialize and turn off a system
Execution	Functions that initiate execution and resets
Shared utility	Shared utility function code

Configure default code mappings for functions by using the **Code Mappings Editor – C** or code mappings API function `setFunctionDefault`. Using these interfaces, you can associate a function category with a function customization template defined in the model's Embedded Coder Dictionary.

If your application requires that you configure the placement of entry-point functions in memory, for example, to optimize the generated code for specific hardware, you can apply a default memory section to a category of functions across a model. In the Code Mappings editor, map a function category to a function customization template that is defined to use a specific memory section. See .

If a function customization template does not exist, you can create one by using the Embedded Coder Dictionary (see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2).

For an example of how to configure default configurations for function categories, see “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15.

Configure Names for Individual C Entry-Point Functions

For your generated C code to adhere to code standards and guidelines or to more easily integrate that code with external code, you can configure the name of an individual entry-point function. For example, you can name the initialize function generated for a model `myInitFunc`.

If your model meets at least one of these criteria, consider configuring the names of functions individually:

- Uses multiple functions that have unique naming requirements.
- Uses few functions.
- Has a default configuration for a function category and you need to override the configuration for a specific function.

Configure function names for individual step functions by using the Code Mappings editor or code mappings API function setFunction. Using these interfaces, you can associate a function with a function customization template that is defined in the model's Embedded Coder Dictionary and specify a function name.

This example shows how to change the name of the `initialize` function for model `rtwdemo_fcnprotoctrl`.

1. Open and save a copy of model `rtwdemo_fcnprotoctrl` in a writable location.
2. Open the **Embedded Coder** app.
3. On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.
4. In the Code Mappings editor, **Functions** tab, change the function name by using one of these methods:

- In the **Function Name** column, enter a name for the function.
- In the **Function Preview** column, click the function prototype hyperlink to open a configuration dialog box. In the **C Initialize Function Name** field, enter the function name. Click **Apply** to verify the change in the **C function prototype** field or click **OK** to exit the dialog box and view the function name in the **Function Preview** column of the Code Mappings editor.

For this example, change the name of the `initialize` function to `fcnprotoctrl_init`.

5. Save the model.
6. Generate code.
7. Verify the changes in the generated code. In the **Code** view, find instances of function name `fcnprotoctrl_init`. The function name appears in generated files `rtwdemo_fcnprotoctrl.h` and `rtwdemo_fcnprotoctrl.c`.

   - In `rtwdemo_fcnprotoctrl.h`:

     ```
     extern void fcnprotoctrl_init(void);
     ```

   - In `rtwdemo_fcnprotoctrl.c`:

     ```
     void fcnprotoctrl_init(void)
     {
       ...
     }
     ```

Configure Name and Arguments for Individual Step Functions

For your generated C code to adhere to code standards and guidelines or to more easily integrate the code with external code, you can configure the name and arguments of individual step functions generated for Simulink periodic functions. You can customize these aspects of a step function:

- Function name
- Argument names
- Order of the arguments
- Return value and argument data qualifiers
- Buffering optimizations for arguments

If your model meets at least one of these criteria, consider configuring the name and arguments of a step function individually:

- Is a periodic, single-tasking model for which the code generator produces one step function.
- Has a default configuration for **Execution** function category and you need to override the configuration for a specific step function.

Configure code mappings for individual step functions by using the Code Mappings editor or code mappings API function `setFunction`. Using these interfaces, you can associate a function with a function customization template that is defined in the model's Embedded Coder Dictionary and completely customize the step function prototype.

This example shows how to customize the step function name and arguments for a rate-based, single-tasking model.

Open the Configuration Dialog Box

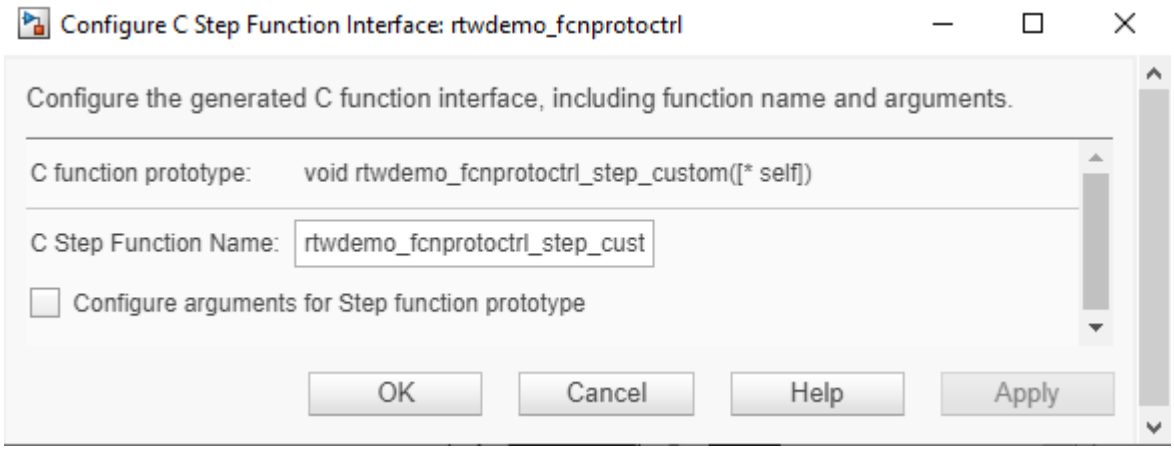
1. Open and save a copy of model rtwdemo_fcnprotoctrl to a writable location.

2. Open the **Embedded Coder** app.

3. On the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

4. In the Code Mappings editor, click the **Functions** tab.

5. For the step function row, in the **Function Preview** column, click the prototype hyperlink to open the Configure C Step Function Interface dialog box.

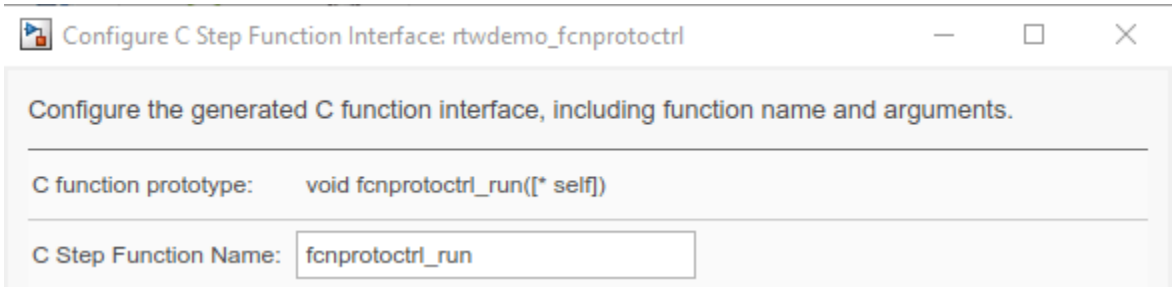


In the Configure C Step Function Interface dialog box, the **C function prototype** field displays a preview of changes that you make to the function prototype.

For models configured to generate multi-instance code, see “Generate Reentrant Code from Top Models” on page 8-13.

Customize the Function Name

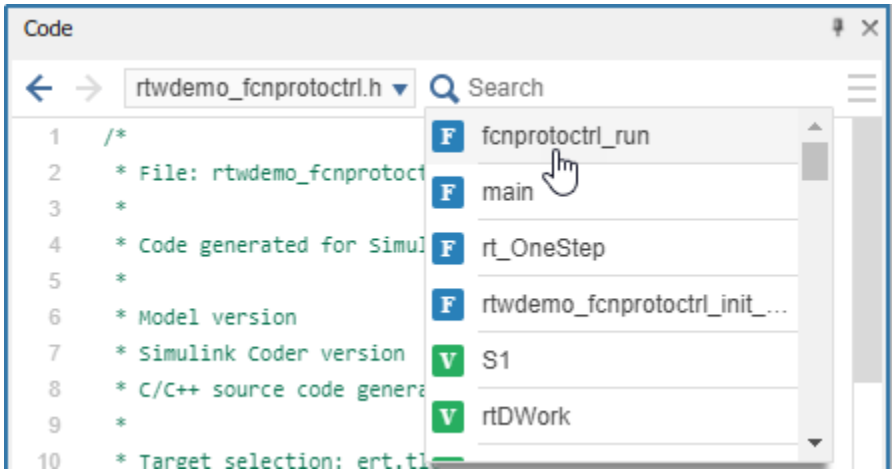
1. In the Configure C Step Function Interface dialog box, set **C Step Function Name** to `fcnprotoctrl_run`. The function preview is updated to reflect the new function name.



2. Click **Apply**, close the dialog box, and save the model.

3. Generate code and verify the name change.

- Click the **Code** tab. In the **Code** view file list, select file `rtwdemo_fcnprotoctrl.h`. In the **Search** field, verify that the list of code elements includes function name `fcnprotoctrl_run`. To view the declaration, select the function name.



The declaration is highlighted in the code:

```
/* Model entry point functions */
extern void rtwdemo_fcnprotoctrl_init_custom(void);
extern void fcnprotoctrl_run(void);
```

- Verify the name change in file `rtwdemo_fcnprotoctrl.c`. To view the source code (definition), in the **Search** field, select `fcnprotoctrl_run`.

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

For more examples, see “Configure Names for Individual C Entry-Point Functions” on page 33-119.

Customize Function Arguments

Configure Global Data Structures

By default, a top or referenced model uses a `void-void` step function to pass data. This type of function enables generated code to communicate with external code by accessing global data stored as data structures that reside in shared memory.

To configure a model to have a `void-void` step function, open the Configure C Step Function Interface dialog box and clear the **Configure arguments for Step function prototype** check box. Verify the change in the **C function prototype** preview.

Configure Arguments

Rate-based model step functions can use arguments to pass data. You can customize these argument settings:

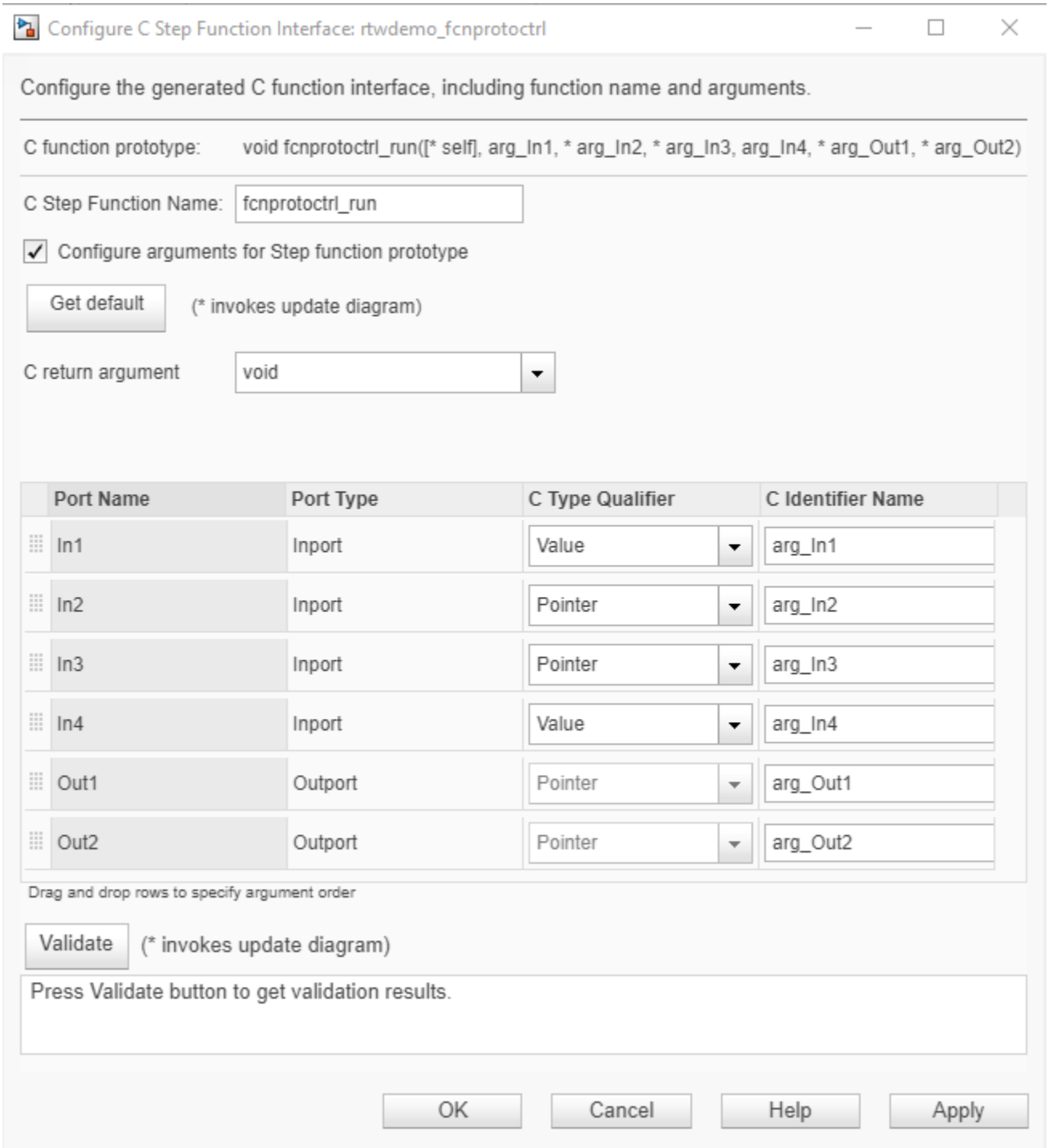
- Return value
- Type qualifiers
- Names
- Order

Example

1. Open the Configure C Step Function Interface dialog box. In the Code Mappings editor, click the **Functions** tab. Then, click the function preview for the step function.

2. In the Configure C Step Function Interface dialog box, select **Configure arguments for Step function prototype**.

3. Click **Get default.** The dialog box expands to show a field for configuring the function return value and a table that lists input and output arguments with default settings.



4. Customize the return value of the step function. Set **C return argument** to either `void` or one of the output arguments. For this example, select `void`.

5. For each argument, specify a **C Type Qualifier**.

Type Qualifiers:

- `Value`: Value, for example, `arg`
- `Const`: Value with the `const` qualifier, for example, `const arg`
- `Pointer to const`: Value with `const` qualifier and referenced by pointer, for example, `const *arg`
- `Pointer`: Reference by pointer, for example, `*arg`
- `Const pointer to const`: Value with `const` qualifier, reference by pointer, and the pointer itself, for example, `const *const arg`

Models that include a referenced model

- For a referenced model, the type qualifier for a root input argument in the step function interface is set to `Auto`. The code generator uses the interface specification by generating a type cast that discards the `const` qualifier from the source signal.
- For the parent of the reference model, the type qualifier for the source signal is set to a value other than `Auto`. To override this behavior, add a `const` type qualifier to the referenced model.

For this example, make no changes to the C type qualifier.

6. Customize the argument names by editing values in the **C Identifier Name** column.

For this example, change the names of the arguments by removing the underscores, for example, change arg_In1 to argIn1.

7. Customize the argument order by dragging table rows to new positions.

For this example, move Out1 to the first position. Move Out 2 to the third position.

8. Validate, apply your changes, close the dialog box, and save the model.

9. Generate code and verify the changes in the **Code** view.

- Verify your changes in file `rtwdemo_fcnprotoctrl.h` by checking that the **Search** field includes function `fcnprotoctrl_run`.To view the function declaration, select the function name.

```
/* Customized model step function */
extern void fcnprotoctrl_run(boolean_T *argOut1, real_T argIn1, BusObject
  *argOut2, BusObject *argIn2, BusObject *argIn3, uint8_T argIn4);
```

- Verify your changes in file `rtwdemo_fcnprotoctrl.c`.To view the definition for the function, select function name `fcnprotoctrl_run`.

```
/* Model step function */
void fcnprotoctrl_run(boolean_T *argOut1, real_T argIn1, BusObject *argOut2,
                      BusObject *argIn2, BusObject *argIn3, uint8_T argIn4)
{
```

Prototype Different from Preview

In the Configure C Step Function Interface dialog box, the function prototype preview for a referenced model can show fewer arguments than the function prototype in the generated code.

For example, consider a model named `mdlref_counter` with an inport (`arg_input`), an outport (`arg_output`), and a saturation block with limits that have workspace parameter argument names `lower_saturation_limit` and `upper_saturation_limit`. The Configure C Step Function Interface dialog box shows a function prototype preview as:

```
mdlref_counter_custom(arg_input, arg_output)
```

In the generated code, the prototype includes the parameter arguments:

```
mdlref_counter_custom(real_T arg_input, real_T arg_output, real_T
  rtp_lower_saturation_limit, real_T rtp_upper_saturation_limit)
```

Optimize Buffering for Arguments

You can optimize the buffering required for I/O arguments in the step function for a model. If you adhere to these requirements when configuring an inport and outport pair, the code generator merges the corresponding arguments and reuses the associated buffer.

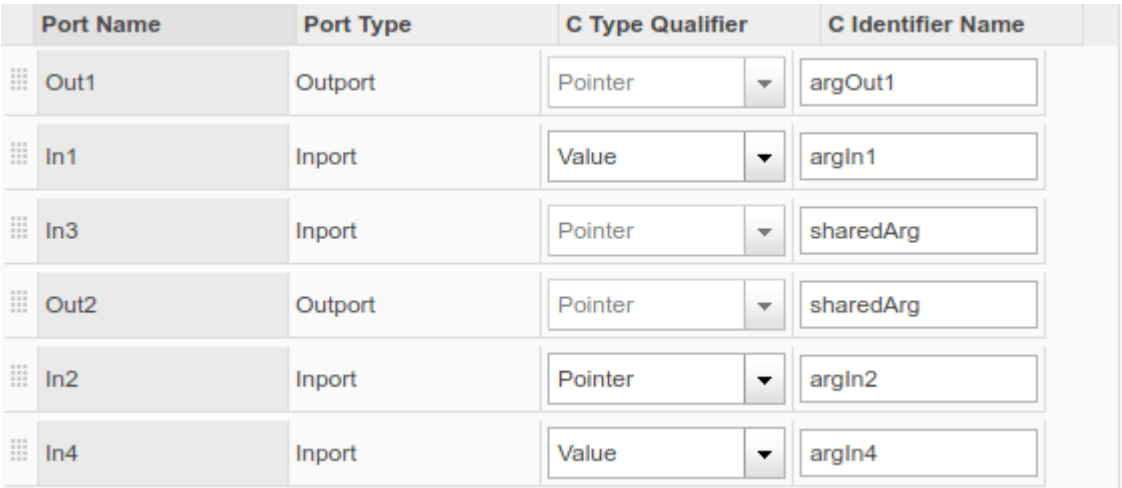
- The Inport and Outport blocks must have the same properties, including data type, dimension, and sample rate.
- The sample rate of the Inport and Outport blocks must be the same as the base rate of the model.
- A conditionally executed subsystem cannot drive the Outport block.
- A single, nonvirtual block output must drive the Outport block. For example, a Mux block, which merges multiple buffers, cannot drive the Outport block.
- In the Configure C Step Function Interface dialog box, you must configure the Inport and Outport with the same C type qualifier and identifier name.

This example shows how to merge arguments for ports `In3` and `Out2` for example model rtwdemo_fcnprotoctrl.

1. Open the Configure C Step Function Interface dialog box for the model `rtwdemo_fcnprotoctrl`.

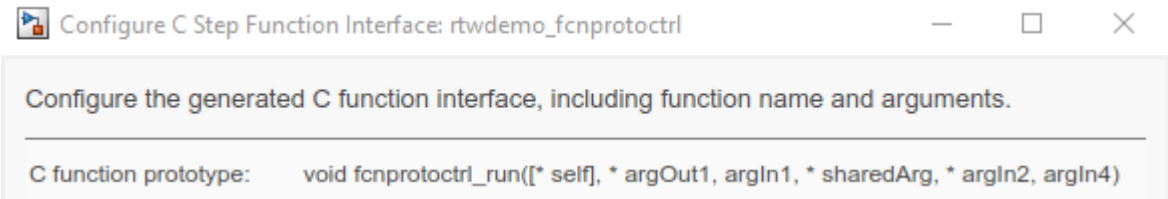
2. Open the display for the arguments by selecting **Configure arguments for Step function prototype**.

3. For ports `In3` and `Out2`, set **C Type Qualifier** to `Pointer` and **C Identifier Name** to `sharedArg`.



Port Name	Port Type	C Type Qualifier	C Identifier Name
Out1	Outport	Pointer	argOut1
In1	Inport	Value	argIn1
In3	Inport	Pointer	sharedArg
Out2	Outport	Pointer	sharedArg
In2	Inport	Pointer	argIn2
In4	Inport	Value	argIn4

4. Verify that the preview shows the merged argument for `argIn3` and `argOut2`.



5. Validate, apply your changes, close the dialog box, and save the model.

6. Generate and view the code. In the **Code** view, search for the `fcnprotoctrl_run` function and check the function interface. For this example, the shared argument appears in read code for the Inport block and write code for the Outport block.

```
/* Model step function */
void fcnprotoctrl_run(boolean_T *argOut1, real_T argIn1, BusObject *sharedArg,
                      BusObject *argIn2, uint8_T argIn4)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Constant: '<Root>/Constant2'
   *  Inport: '<Root>/In1'
   *  Logic: '<Root>/LogOp'
   *  RelationalOperator: '<Root>/RelOp1'
   *  RelationalOperator: '<Root>/RelOp2'
   *  UnitDelay: '<Root>/Unit Delay'
   */
  *argOut1 = ((rtDWork.UnitDelay_DSTATE > 10.0) || (argIn1 < 2.0));

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In4'
   */
  if (argIn4 != 0) {
    S1 = *argIn2;
  } else {
    S1 = *sharedArg;
  }

  /* End of Switch: '<Root>/Switch' */

  /* BusCreator: '<Root>/BusConversion_InsertedFor_Out2_at_inport_0' */
  *sharedArg = S1;

  /* Update for UnitDelay: '<Root>/Unit Delay' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDWork.UnitDelay_DSTATE = argIn1;
}
```

Configure Code Generation Settings for C Entry-Point Functions Programmatically

To automate configuration of model functions for C code generation, use the programming interface for code mappings. For example, when creating custom block libraries or part of an application test environment, use the programming interface to automate function configuration.

Typical uses of these functions include:

- Create a function interface.
- Modify an existing function interface.
- Create a function interface, starting with default configuration information from a model.
- Reset the model function interface to the default ERT function configuration.

For an example of how to configure code generation settings for C entry-point functions programmatically, see “Configure Default Data and Function Code Generation Programmatically” on page 33-24. That example shows how to configure default settings for function categories and individual functions.

How to Interface with Generated Entry-Point Functions

1. After generating code for a model, use the **Code** view to review the generated entry-point functions and, if applicable, variables representing external input and output ports.
2. Add `#include` statements to your external code that include the generated header files that declare the model entry-point functions.
3. Add an `#include` statement that includes the generated file `rtwtypes.h`. This file provides type definitions, `#define` statements, and enumerations.
4. Initialize target-specific data structures and hardware, such as ADCs or DACs.
5. If applicable, initialize data for each instance of a reusable model.
6. If applicable, write input data to generated variables that represent model Inport blocks.
7. Call the generated entry-point functions or set up use of the `rt_OneStep` function.
8. If applicable, read data from generated variables that represent model Outport blocks.

For more information, see “Deploy Applications to Target Hardware” on page 71-2.

C Function Interface Customization Limitations

These limitations apply to customizations for generated C function interfaces:

- You must select the model configuration parameter **Single output/update function**.
- Multirate models are supported, but you must configure the models for single tasking.
- You must configure root-level inports and outports to use the `Default` storage class.
- If you choose to customize a function interface, you must provide your own custom `main` program. You cannot configure the function interface with the static `rt_main.c` that MathWorks provides. Specifying a function interface configuration other than the default creates a mismatch between the generated code and the default static `rt_main.c`.
- The code generator removes the data structure for the root inports of the model unless a subsystem implemented by a nonreusable function uses the value of one or more of the inports.
- The code generator removes the data structure for the root outports of the model except when you enable MAT-file logging or if the sample time of one or more of the outports is not the fundamental base rate (including a constant rate).
- If you copy a subsystem block to create a block in a new model or the same model, the function interface information from the original subsystem block is not copied to the new subsystem block.

- If you have Stateflow, for a Stateflow chart that uses a model root inport value or that calls a subsystem that uses a model root inport value, do one of the following to generate code:
  - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
  - Make the Stateflow function a nonreusable function.
  - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- If a model root inport value connects to a Simscape conversion block, insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured with a function interface, do not use virtual buses as inputs or outputs to the referenced model. Use nonvirtual buses instead.
- If the C function interface is not the default, the value is ignored for the model configuration parameter **Pass fixed-size scalar root inputs by value for code generation**. For more information, see Pass fixed-size scalar root inputs by value for code generation.

See Also

Simulink Function | Function Caller

More About

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “Simulink Function Blocks and Code Generation” on page 8-6

Configure Multi-Instance Code Generation

By default, for top models, the code generator produces code that is not reentrant. Entry-point functions have a `void-void` interface. Code communicates with other code by sharing access to global data structures that reside in shared memory.

For applications that can benefit from code reuse and require that each use or instance of the code maintains its own unique data, configure a model such that the code generator produces reentrant, multi-instantiable code. Multiple programs can use reentrant code simultaneously. When you configure a model for reentrancy, the execution (step) entry-point function uses root-level input and output arguments instead of global data structures.

Configure a Top Model for Multi-Instance Code Generation

Set model configuration parameter **Code interface packaging** to `Reusable function`.

Apply additional diagnostic and code generation control by setting these model configuration parameters:

- To select the severity level for diagnostic messages that the code generator displays when a model does not meet requirements for multi-instance code, set parameter **Multi-instance code error diagnostic** to `None`, `Warning`, or `Error`.
- To control how the generated code passes root-level model input and output to the reusable execution (step) function (requires Embedded Coder), set parameter **Pass root-level I/O as** to `Individual arguments`, `Structure reference`, or `Part of model data structure`.
- To include a function in the generated file *`model`*`.c` that uses `malloc` to dynamically allocate memory for model data structures (requires Embedded Coder), set parameter **Use dynamic memory allocation for model initialization** to `On`.

For more information, see “Generate Reentrant Code from Top Models” on page 8-13.

Configure a Referenced Model for Multi-Instance Code Generation

Set model configuration parameter **Total number of instances allowed per top model** to `Multiple`. With this parameter setting, a model can be referenced more than once in a model hierarchy, if it contains not constructs that preclude multiple references. An error occurs if the model cannot be referenced multiple times, even if only one reference exists.

For more information, see “Generate Reentrant Code from Simulink Function Blocks” on page 8-18.

Configure Data Interfaces

Configure Default Code Generation for Data

1. Click the **Code Mappings - C** tab.
2. In the Code Mappings editor, update the model to include the latest model data by clicking the update model data button.
3. Click the **Data Defaults** tab.
4. In the **Data Defaults** table, select a row for a model data element category.
5. In the **Storage Class** column, select a storage class. If your model is linked to a Simulink data dictionary that includes a shared coder dictionary for which defaults are set up, you can select `Dictionary Default`. For help on choosing a storage class, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75. For information on using dictionary defaults, see “Configure Model Elements for Code Generation with Definitions Stored in Shared Data Dictionary” on page 33-22. If you need to define a new storage class, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.
6. In the Property Inspector, add values for unspecified properties. For example, if you select storage class `ImportFromFile`, specify a value for the property **HeaderFile**.
7. Save the model.

Override Default Data Interface for Individual Data Elements

1. Click the **Model Data Editor** tab.
2. Update the model to include the latest model data by clicking the update model data button.
3. In the Model Data Editor, with the view set to **Code**, click the tab for the type of data element that you want to configure.
4. In the filter text box, enter a string that will help you find the row for that data element.
5. In the **Storage Class** column, select a storage class. For example, you might want to change the storage class setting for Model Workspace parameters to `Model default`.
6. In the table or Property Inspector, set values for available properties.
7. Save the model.

Configure Entry-Point Function Interfaces

Define Function Customization Templates

Unless someone has defined function customization templates for a model, the Code Mappings editor presents `Default` as the only template option. To define a template, use the Embedded Coder Dictionary.

1. Open the Embedded Coder Dictionary. In the **C Code** tab, select **Settings** > **Embedded Coder Dictionary**.
2. In the Embedded Coder Dictionary, click the **Function Customization Templates** tab.
3. Click **Add**.
4. In the new row of the table, name the template.
5. As required by your function interface:
   - In the **Function Name** column, specify a default naming rule for entry-point functions.
   - In the **Memory Section** column, specify the name of an available memory section.

   The preview shows the results of your changes and additions.
6. Close the dictionary.

Configure Default Code Generation for Entry-Point Functions

1. Define a function customization template in the Embedded Coder Dictionary.
2. In the Code Mappings editor, click the **Function Defaults** tab.
3. In the **Function Defaults** table, select a row for a model function category.
4. In the **Function Customization Template** column, select a customization template. If a Simulink data dictionary that includes a shared coder dictionary for which defaults are set up is attached to your model, you can select `Dictionary Default`. For information on using dictionary defaults, see “Configure Model Elements for Code Generation with Definitions Stored in Shared Data Dictionary” on page 33-22. If you need to define a new customization template, for more information, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.
5. Save the model.

Override Default Code Generation for Entry-Point Functions

1. In the Code Mappings editor, click the **Functions** tab.
2. In the **Source** column, select an entry-point function for which you want to override the default interface configuration.
3. As required by your function interface:
   - Select a function customization template.
   - Specify a unique function name.
4. If your entry-point function is an execution function, you can customize the interface arguments. By default, the code generator produces a `void-void` interface (without arguments) and

assumes variables are global. To configure the interface to pass arguments, in the **Function Preview** column, click the prototype hyperlink to open a configuration dialog box.

5. In the function interface configuration dialog box, select **Configure arguments for Step function prototype**.
6. Click **Get Default**. A table shows a list of current argument settings.
7. In the table, you can change C type qualifiers, identifier names, and argument order as required by the function interface. Change the order of the arguments by dragging them to locations in the list. As you make changes, use the prototype preview near the top of the dialog box to verify the results.
8. Validate your changes and save the model.

Configure Internal Data

Configure Default Representation of Internal Data

1. In the Code Mapping Editor, click the **Data Defaults** tab.
2. Select the row representing the category of internal data that you want to configure.
3. In the **Storage Class** column, select a storage class. If a Simulink data dictionary that includes a shared coder dictionary for which defaults are set up is attached to your model, you can select `Dictionary Default`. For help on choosing a storage class, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75. For information on using dictionary defaults, see “Configure Model Elements for Code Generation with Definitions Stored in Shared Data Dictionary” on page 33-22. If you need to define a new storage class, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.
4. In the Property Inspector, add values for unspecified properties. For example, if you select storage class `Const`, specify values for properties **HeaderFile** and **DefinitionFile**.
5. Save the model.

Override Default Representation for Individual Internal Data Elements

1. Click the **Model Data Editor** tab.
2. In the Model Data Editor, with the view set to **Code**, click the tab for the type of data element that you want to configure.
3. Update the model to include the latest model data by clicking the update model data button.
4. In the filter text box, enter a string that will help you find the row for the data element.
5. In the **Storage Class** column, select a storage class. For example, you might want to change the storage class setting for Model Workspace parameters to `Model default`.
6. In the table or Property Inspector, set values for available properties.
7. Save the model.

Configure Subsystem Function Packaging and Interface

1. In the Simulink Editor, select the subsystem that you want to configure.
2. In the Property Inspector, expand **Code Generation**.
3. Set **Function packaging** to `Nonresuable function` or `Reusable function`.
4. Configure the function name. Set **Function name options** to `Use subsystem name` or `User specified`. If you select `User specified`, specify a value for the **Function name** property.
5. Configure the file name for the function code. Set **File name options** to `Use subsystem name`, `Use function name`, or `User specified`. If you select `User specified`, specify a value for the **File name (no extension)** property.
6. If you set **Function packaging** to `Nonresuable function`, configure the function interface. Set **Function interface** to `void-void`, `Allow arguments (Optimized)`, or `Allow arguments (Match graphical interface)`.
7. If you want the code generator to declare subsystem data independently from parent model data, select property **Function with separate data**.
8. Save the model.

For more information, see “Generate Subsystem Code as Separate Function and Files” on page 4-7.

Configuring Service Interfaces in the Generated Code

- “Generate Sender and Receiver C Interface Code for Component Deployment” on page 34-2
- “Generate C Data Transfer Service Interface Code for Component Deployment” on page 34-6
- “Generate C Timer Service Interface Code for Component Deployment” on page 34-13
- “Generate C Parameter Tuning Service Interface Code for Component Deployment” on page 34-20
- “Generate C Measurement Service Interface Code for Component Deployment” on page 34-24

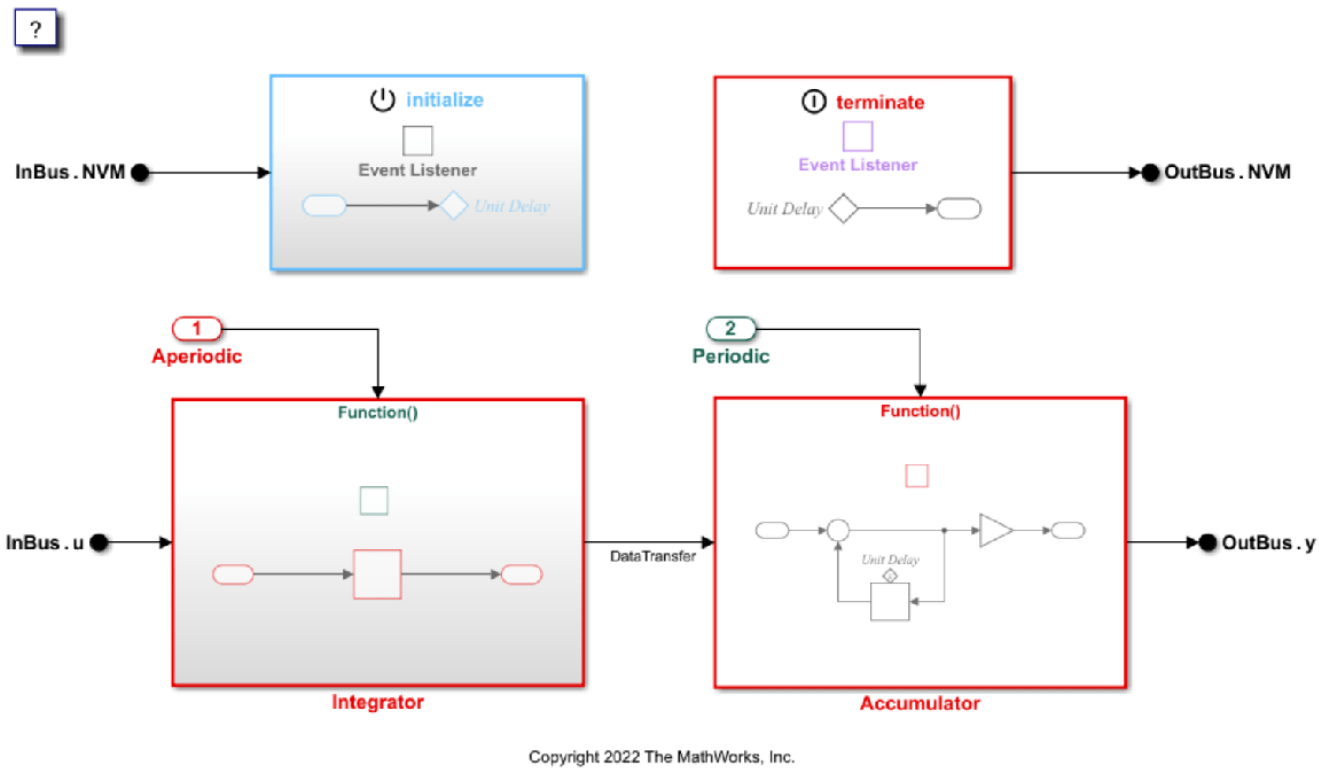
Generate Sender and Receiver C Interface Code for Component Deployment

This example shows how to generate sender and receiver service functions to send and receive data to and from the target environment by using environment-specific data communication methods. Use data communication methods that match the needs of the target environment. For more information about data communication methods, see “Data Communication Methods” on page 1-14.

Inspect Model

Open the model.

```
model = "ComponentDeploymentFcn";
open_system(model);
```



The model contains mixed periodic and aperiodic rates, which require data concurrency safeguards. To enforce data concurrency safeguards, use the `Outside Execution` or `During Execution` data communication methods. With these data communication methods, the code generator uses a buffer rather than writing to memory directly. For models that do not require data concurrency safeguards, consider using the `Direct Access` data communication method, which allows the generated code to read from and write to the target environment memory directly.

Inspect Embedded Coder Dictionary

Inspect the shared Embedded Coder Dictionary and verify the sender and receiver service configuration.

1. Open the Embedded Coder Dictionary. From the **C Code** tab, under **Code Interface**, select `Embedded Coder Dictionary (Shared)`.

2. In the Embedded Coder Dictionary, in the left pane, select **Service Interfaces**. The center pane populates with service interfaces.

3. In the center pane, under **Receiver**, verify that the `ReceiverOutsideExe` service interface has the **Dictionary Default** check box selected.

4. Click the `ReceiverOutsideExe` row. The right pane populates with information about the service interface. In this case, the service interface uses the `Outside Execution` data communication method and the function naming rule `get_$X_input`. The `$X` token represents the name of the entry-point function that encloses the access function.

5. Under **Sender**, verify that the `SenderOutsideExe` row is selected as **Dictionary Default** and that the service uses the `Outside Execution` data communication method and the function naming rule `set_$X_output`.

When you select the `Outside Execution` data communicaiton method, the generated code applies data concurrency safeguards by using a buffer provided by the target environment. `Outside Execution` allows communication with the target environment to occur only between task iterations.

Inspect Code Mappings

Inspect the Code Mappings editor and verify that the model root inports and outports map to the default sender and receiver services.

1. From the **C Code** tab, under **Code Interface**, select **Component Interface**. The Code Mappings editor pane opens.

2. In the Code Mappings editor, select the **Inports** tab. The model contains two Bus Element Inport blocks, `InBus.NVM` and `InBus.u`. Verify that, for both inports, the **Receiver Service** column is set to `Dictionary default: ReceiverOutsideExe`.

3. From the **Outports** tab, verify that both of the Bus Element Outport blocks, `OutBus.NVM` and `OutBus.y`, use the `Dictionary default: SenderOutsideExe` sender service.

Generate Code

Verify that the generated code uses the `Outside Execution` communication method.

1. Generate the code.

```
slbuild(model);
```

```
### Starting build procedure for: ComponentDeploymentFcn
### Successful completion of code generation for: ComponentDeploymentFcn

Build Summary

Top model targets built:
```

```
Model                    Action           Rebuild Reason
==========================================================================================
ComponentDeploymentFcn  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 23.454s
```

2. Inspect the `services.h` file.

```
/* receiver services */
extern const real_T * get_CD_integrator_input(void);
extern const real_T * get_CD_initialize_input(void);

/* sender services */
extern real_T * set_CD_accumulator_output(void);
extern real_T * set_CD_terminate_output(void);
```

With the `Outside Execution` data communication method, the code generator expects the target environment to provide a buffer for data communication. The sender and receiver function services do not need a pointer to a generated buffer; each service function takes no arguments.

In the service function names, the code generator replaces the `$X` token with the name of the enclosing entry-point function, `CD_accumulator`.

3. Inspect the `CD_accumulator` function in `ComponentDeploymentFcn.c`.

```
void CD_accumulator(void)
{
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_sig.delay[i] += tmpIrvIRead[i];
    (set_CD_accumulator_output())[i] = CD_param.tunable_gain * CD_sig.delay[i];
  }
}
```

With the `Outside Execution` data communication method, the generated code calls the sender service inside the `for` loop. The sender service populates the provided buffer or gives access to that buffer for the entry-point function to populate. The target environment reads the buffer after the entry-point function executes and uses it for the next task iteration.

Modify Code Mappings and Regenerate Code

1. Open the Code Mappings editor and navigate to the **Inports** tab. For each inport, change the **Receiver Service** column from `Dictionary default: SenderOutsideExe` to `ReceiverDuringExe`. When you select this service, the sender services use the `During Execution` data communication method. When using this data communication method, the code generator still enforces data concurrency safeguards by using a buffer, but generates the buffer in the code rather than expecting the buffer to be provided by the target environment. The `During Execution` data communication method allows services to transfer data during a task iteration.

2. Navigate to the **Outports** tab and map each outport to `SenderDuringExe`.

3. Generate the code.

```
slbuild(model);

### Starting build procedure for: ComponentDeploymentFcn
### Generated code for 'ComponentDeploymentFcn' is up to date because no structural, parameter o
### Successful completion of code generation for: ComponentDeploymentFcn

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 7.8389s
```

4. Inspect the `CD_accumulator` function in `ComponentDeploymentFcn.c`.

```
void CD_accumulator(void)
{
  real_T OutBus_y[10];
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_sig.delay[i] += tmpIrvIRead[i];
    OutBus_y[i] = CD_param.tunable_gain * CD_sig.delay[i];
  }

  set_CD_accumulator_output(&OutBus_y[0]);
}
```

With the `During Execution` data communication method, the generated code includes the same sender and receiver services as before, but these services can now execute during task iterations. The code generator generates a data buffer, `OutBus_y`, in the code. The generated code populates this buffer inside the `for` loop, then sends this buffer through the sender service after the `for` loop. This means that the sender service is called only once during the function execution.

5. Inspect the `services.h` file.

```
/* receiver services */
extern void get_CD_integrator_input(real_T *const rty_InBus_u_value);
extern void get_CD_initialize_input(real_T *const rty_InBus_NVM_value);

/* sender services */
extern void set_CD_accumulator_output(const real_T *const rtu_OutBus_y);
extern void set_CD_terminate_output(const real_T *const rtu_OutBus_NVM);
```

The generated sender and receiver services each take a pointer to a buffer in the generated code as an argument.

Related Examples

- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Generate C Data Transfer Service Interface Code for Component Deployment

This example shows how to generate calls to target platform data transfer service functions to exchange data between callable entry-point functions generated from a component model. The generated interface code must align with environment-specific service function prototypes and data communication methods.

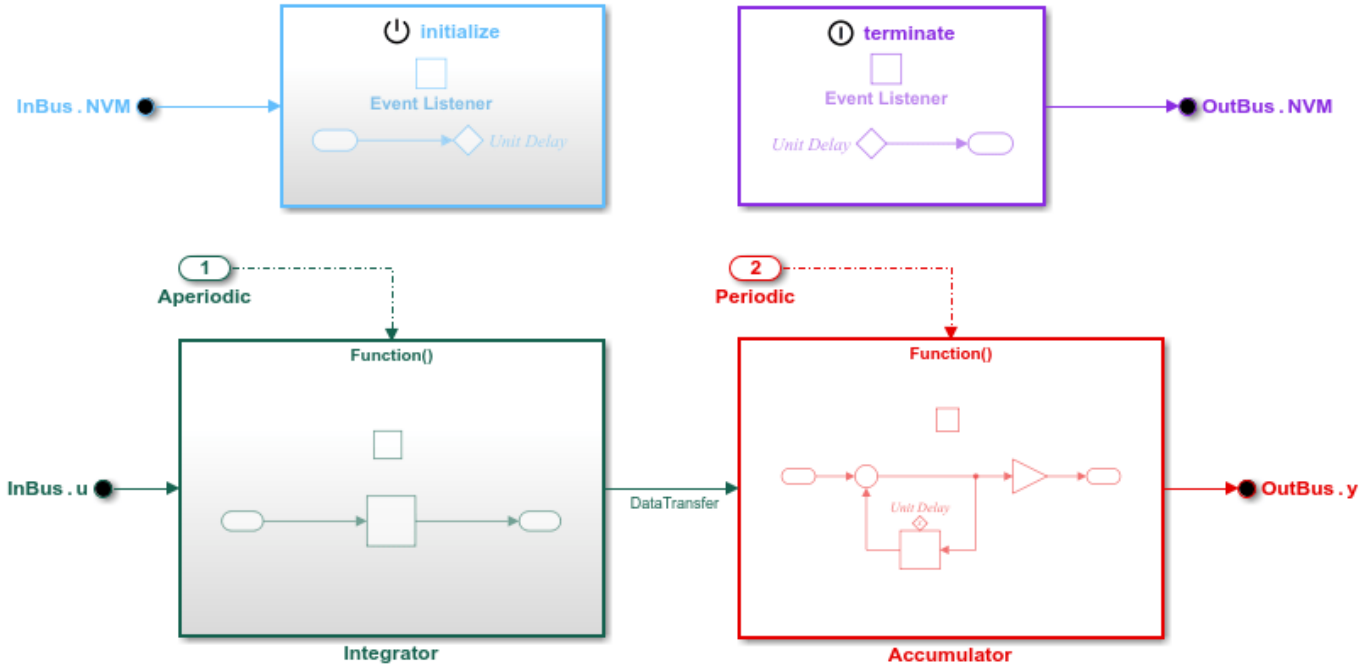
In the example you:

1. Represent a data transfer request in the top model.
2. Configure the code generator to apply the default data transfer code interface.
3. Generate and inspect the interface code.
4. Configure the data transfer code interface for during-execution data access.
5. Regenerate and reinspect the interface code.

Represent Data Transfer Request in Top Model

Open example component model `ComponentDeploymentFcn`.

```
open_system('ComponentDeploymentFcn');
```



The signal line that connects the output port of the `Integrator` function to the input port of the `Accumulator` function represents a data transfer. The integrator function calls the data transfer service to send data to the accumulator function. The accumulator function calls the data transfer service to receive the data from the integrator function. For more information about modeling data transfers, see “cgsl_0409: Data transfer for component deployment”.

Configure Data Transfer Code Interface

The example model is linked to shared Embedded Coder Dictionary file `ComponentDeploymentCoderDictionary.sldd`, which defines a service code interface configuration. That configuration defines two data transfer service interfaces, one for outside-execution data access (the default) and one for during-execution data access. Both interfaces specify naming rules `set_$X$N` and `get_$X$N`, where `$X` is the name of the service function and `$N` is the name of the signal that represents the data transfer.

Configure the model such that the code generator applies the coder dictionary default interface for data transfers.

1. Open the Embedded Coder app.
2. Select **Code Interface** > **Component Interface**.
3. In the Code Mappings editor, click the **Data Transfers** tab.
4. Select the row for `Data Transfer`.
5. From the menu in the **Data Transfer Service** column of the selected row, select `Dictionary default: DataTransferOutsideExe`. As the name indicates, the default interface is configured to use outside-execution data communication.

Generate and Inspect Interface Code

Generate code from the example model. The code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. Model data and entry-point functions are accessible to a caller by including the header file in target environment software.

The code generator also produces an interface header file, which by default the code generator names `services.h`. This file includes declarations for platform service interface functions called by the component code.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with code interface requirements.

For a data transfer service interface, the report provides:

- Receiver and sender function prototypes
- Data communication method used
- Generated callable entry-point functions that call the service functions

Header File

The header file `ComponentDeploymentFcn.h` includes interface header file `services.h`, defines an instance of the real-time model data structure, and declares structures and callable entry-point functions for data and function interface elements represented in the model.

This code fragment shows lines of code in the header file that are relevant to data transfer requests: the inclusion of `services.h` and the generated entry-point functions that call data transfer service functions.

```
#include "services.h"
.
```

```
.
.
extern void CD_initialize(void);
extern void CD_accumulator(void);
extern void CD_integrator(void);
```

Source Code File

When the model is configured for the code generator to produce interface code that uses outside-execution data communication, entry-point functions in the generated source code file `ComponentDeploymentFcn.c` call these target platform data transfer service functions:

- `get_CD_accumulator_DataTransfer`
- `set_CD_integrator_DataTransfer`
- `set_ModelInitialize_DataTransfer`

When a service interface uses the outside-execution data communication method, calls to the service functions take no arguments.

```
void CD_accumulator(void)
{
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += tmpIrvIRead[i];
    (getref_CD_accumulator_OutBus_y())[i] = CD_tunable.k * CD_measured.delay[i];
  }
}

void CD_integrator(void)
{
  real_T tmp;
  real_T *tmp_0;
  int32_T i;
  uint32_T Integrator_ELAPS_T;
  tmp_0 = set_CD_integrator_DataTransfer();
.
.
.
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) {
      CD_measured.dti[i] += tmp * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.discreteTimeIntegrator_PREV_U[i] = (get_CD_integrator_InBus_u())[i];
}

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  memcpy(&tmp_0[0], &CD_measured.dti[0], (uint32_T)(10U * sizeof(real_T)));
}

void CD_initialize(void)
{
  {
    real_T *tmp;
    int32_T i;
```

```
    tmp = set_ModelInitialize_DataTransfer();
    for (i = 0; i < 10; i++) {
      tmp[i] = 5.0;
    }
.
.
.
}
```

The accumulator function calls the data transfer service function `get_CD_accumulator_DataTransfer` to read an input value transferred from the integrator function. The accumulator function uses the input to compute the accumulated value.

The integrator function calls the data transfer service function `set_CD_integrator_DataTransfer` to transfer an integrated output value to the accumulator function. The initialize function calls the service function `set_ModelInitialize_DataTransfer` to initialize state data that is read from nonvolatile memory. When a service interface uses outside-execution data communication, the generated code calls these data transfer service functions before the `for` loop. The target platform data transfer service:

- Gives access to the buffer for the entry-point functions to populate.
- Reads the buffer after the entry-point function executes.
- Uses the buffered value for the next task iteration.

Services Header File

The services header file `services.h` declares prototypes for calling services that run as platform software in a target environment. This code fragment shows the prototype declarations for calling data transfer service functions.

```
/* data transfer services */
const real_T * get_CD_accumulator_DataTransfer(void);
real_T * set_CD_integrator_DataTransfer(void);
real_T * set_ModelInitialize_DataTransfer(void);
```

For an interface that uses the outside-execution data communication method, the code generator expects the target environment to provide a buffer for data communication. Because the target environment software creates and manages access to the buffer, the data transfer service functions take no arguments.

The function names reflect the naming rules defined for the data transfer service interface in the linked Embedded Coder Dictionary. The naming rules `get_$X$N` and `set_$X$N` instruct the code generator to `$X` with the name of the calling entry-point function and `$N` with the name of the source or destination signal port for the data transfer.

Configure Data Transfer Code Interface for During-Execution Data Access

Configure the example model such that the code generator applies the coder dictionary interface for during-execution data transfers.

1. If the Embedded Coder app is not open, open it and select **Code Interface** > **Component Interface**.
2. In the Code Mappings editor, click the **Data Transfers** tab.
3. Select the row for `Data Transfer`.

4. From the menu in the **Data Transfer Service** column of the selected row, select service interface `DataTransferDuringExe`.
5. Save the configuration change by saving the model.

Regenerate and Reinspect Interface Code

Regenerate code from the example model.

The generated code shown in the following sections assumes that you have configured the inports, outports, data transfers, and the timer service for the integrator function with a service interface that applies the during-execution data communication method.

Header File

The lines of code relevant to the data transfer code interface in header file `ComponentDeploymentFcn.h` are the same as for the outside-execution data transfer interface.

```
#include "services.h"
.
.
.
extern void CD_initialize(void);
extern void CD_terminate(void);
extern void CD_accumulator(void);
extern void CD_integrator(void);
```

Source Code File

When the model is configured for the code generator to produce interface code that uses during-execution data communication, entry-point functions in the generated source code file `ComponentDeploymentFcn.c` call these target platform data transfer service functions:

- `get_CD_accumulator_DataTransfer`
- `set_CD_integrator_DataTransfer`
- `set_ModelInitialize_DataTransfer`

When a service interface uses the during-execution data communication method, the call to service function `get_CD_accumulator_DataTransfer` takes a pointer to a buffer as an argument. Calls to service functions `set_CD_integrator_DataTransfer` and `set_ModelInitialize_DataTransfer` take a constant buffer as an argument.

```
void CD_accumulator(void)
{
  real_T OutBus_y[10];
  int32_T i;
  get_CD_accumulator_DataTransfer(OutBus_y);
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += OutBus_y[i];
    OutBus_y[i] = CD_tunable.k * CD_measured.delay[i];
  }

  set_CD_accumulator_output(&OutBus_y[0]);
}

void CD_integrator(void)
{
```

```
  real_T tmp[10];
  real_T tmp_0;
  int32_T i;
  uint32_T Integrator_ELAPS_T;
  tmp_0 = set_CD_integrator_DataTransfer();
.
.
.
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) {
      CD_measured.dti[i] += tmp_0 * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.discreteTimeIntegrator_PREV_U[i] = tmp[i];
  }

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  set_CD_integrator_DataTransfer(CD_measured.dti);
}

void CD_initialize(void)
{
  {
    real_T tmp[10];
    int32_T i;
    get_CD_initialize_InBus_NVM(&CD_measured.delay[0]);
    for (i = 0; i < 10; i++) {
      tmp[i] = 5.0;
    }

    set_ModelInitialize_DataTransfer(tmp);
.
.
.
  }
}
```

When a service interface uses the during-execution data communcation method, the generated code calls the same data transfer service as when you use the outside-execution mentod. The target platform service can execute while generated entry-point function execute. For each of the interface functions, the code generator produces a data buffer (`OutBus_y`, `CD_measured.dti`, and `tmp)`. The entry-point function code populates the buffer inside the `for` loop.

The accumulator function calls the data transfer service function `get_CD_accumulator_DataTransfer` to read an input value transferred from the integrator function to data buffer `OutBus_y`. The accumulator function uses the input to compute the accumulated value.

The integrator function calls the data transfer service function `set_CD_integrator_DataTransfer` to transfer the integrated output value stored in data buffer `CD_measured_dti` to the accumulator function. The initialize function calls service function `set_ModelInitialize_DataTransfer` to initialize state data that is read from nonvolatile memory data buffer `tmp`. When using outside-execution data communication, the generated code calls these data transfer service functions after the `for` loop. Thus, the target platform data transfer services called once during execution of each entry-point function.

Services Header File

This code fragment shows the prototype declarations for calling data transfer service functions when when the interface uses during-execution data communication.

```
/* data transfer services */
void get_CD_accumulator_DataTransfer(real_T *aValPtr);
void set_CD_integrator_DataTransfer(const real_T aVal[10]);
void set_ModelInitialize_DataTransfer(const real_T aVal[10]);
```

The generated data transfer service function interfaces each take an argument. Function `get_CD_accumulator_DataTransfer` takes a pointer to a buffer as an argument. Functions `set_CD_integrator_DataTransfer` and `set_ModelInitialize_DataTransfer` take a constant buffer as an argument.

Related Examples

- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Generate C Timer Service Interface Code for Component Deployment

This example shows how to generate calls to a target platform timer service function to get the value of the target environment function clock tick for an aperiodic component. The generated interface code must align with the environment-specific service function prototype and data communication method.

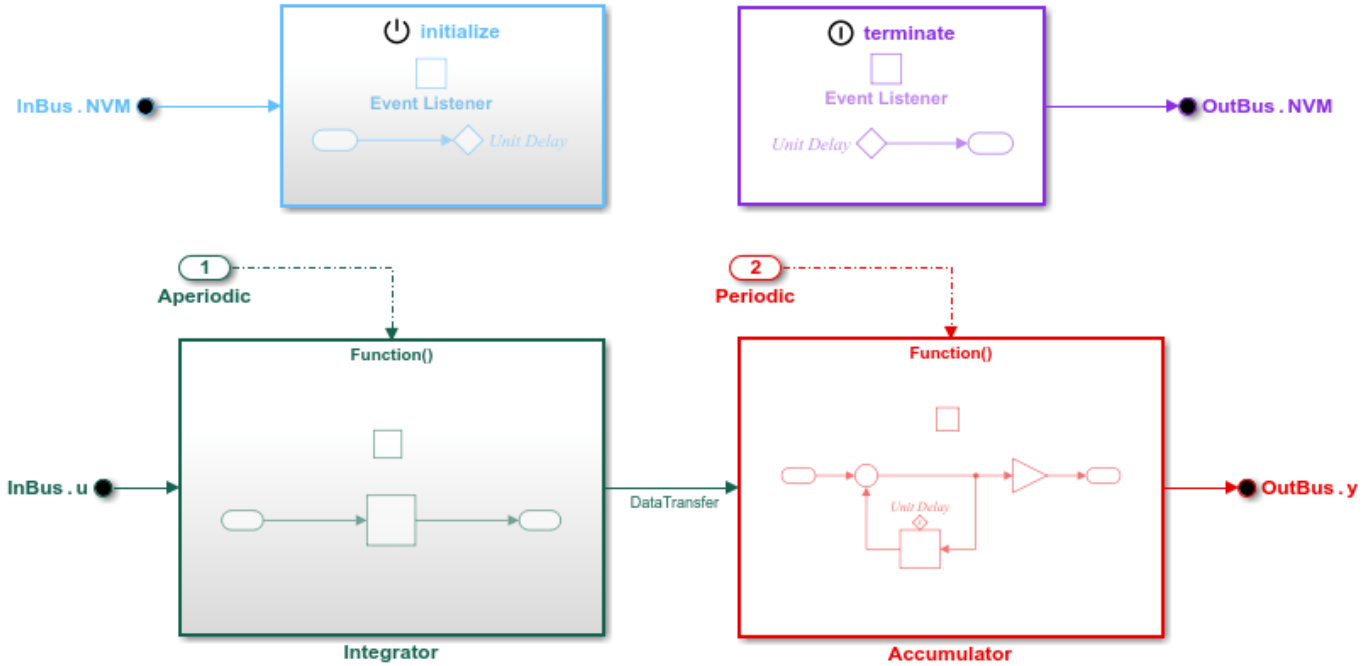
In the example you:

1. Represent a timer service request in the top model.
2. Configure the code generator to apply the default timer service code interface.
3. Generate and inspect the interface code.
4. Configure the timer service code interface for during-execution data access.
5. Regenerate and reinspect the interface code.

Represent Request for Function Clock Tick in Top Model

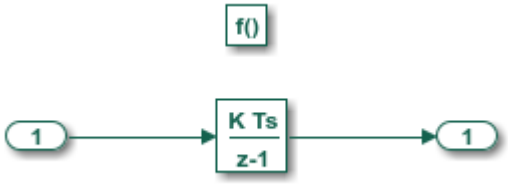
Open example component model `ComponentDeploymentFcn`.

```
open_system('ComponentDeploymentFcn');
```



To model requests for the value of a target environment function clock tick, include a Discrete Time Integrator or Weighted Sample Time block in a top model function. In this example model, the aperiodic integrator function includes a Discrete Time Integrator block.

Open the function-call subsystem `Integrator`.



The function applies a forward integration method and a gain value of 1.25 to the value of state variable `x` each time the function executes.

Configure Timer Service Code Interface

The example model is linked to the shared Embedded Coder Dictionary file `ComponentDeploymentCoderDictionary.sldd`, which defines a service code interface configuration. That configuration defines the timer service interfaces `get_tick_outside` and `get_tick_during`. Interface function `get_tick_outside`, the default, specifies outside-execution data communication. Interface `get_tick_during` specifies during-execution data communication. Both interfaces specify naming rule `$G$X`, where `$G` is the service interface name and `$X` is the name of the callable entry-point function requesting the tick value.

Configure the model such that the code generator applies the coder dictionary default interface for accessing the function clock tick value.

1. Open the Embedded Coder app.
2. Select **Code Interface** > **Component Interface**.
3. In the Code Mappings editor, click the **Functions** tab.
4. Select the row for `Exported Function:Aperiodic` and click the pencil icon.
5. In the dialog box that appears, set the **Timer Service** property to `Dictionary default: get_tick_outside`. As the name indicates, the default interface is configured to use outside-execution data communication. Alternatively, you can set the property in the Property Inspector.
6. Save the configuration change by saving the model.

Generate and Inspect Interface Code

Generate code from the example model. The code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. Model data and entry-point functions are accessible to a caller by including the header file target environment software.

The code generator also produces an interface header file, which by default the code generator names `services.h`. This file includes declarations for platform service interface functions called by the component code.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with code interface requirements. The report provides the timer service interface information in the section for the function that calls the service function. Information provided includes:

- Timer function prototype

- Data communication method used
- Service header file that declares the interface function prototype

Header File

The header file `ComponentDeploymentFcn.h` includes interface header file `services.h`, defines an instance of the real-time model data structure, and declares structures and callable entry-point functions for data and function interface elements represented in the model.

This code fragment shows lines of code in the header file that are relevant to function clock tick requests: the inclusion of `services.h`, variable declarations, and the a declaration for the generated entry-point function that calls the timer service function.

```
#include "services.h"
.
.
.
typedef struct {
  real_T DataTransfer_WriteBuf[10];
  real_T DiscreteTimeIntegrator_PREV_U[10];
  uint32_T Interator_PREV_T;
  uint8_T DiscreteTimeIntegrator_SYSTEM_E;
  boolean_T Integrator_RESET_ELAPS_T;
} D_Work;

typedef struct {
  real_T delay[10];
  real_T dti[10];
} CD_measured_T;
.
.
.
extern void CD_integrator(void);
```

Source Code File

When the model is configured for the code generator to produce interface code that uses outside-execution data communication, the integrator entry-point function in the generated source code file `ComponentDeploymentFcn.c` calls target platform timer service function `get_tick_outside_CD_integrator`.

When a service interface uses the outside-execution data communication method, calls to the service function take no arguments.

```
CD_measured_T CD_measured;
.
.
.
void CD_integrator(void)
{
  real_T tmp;
  real_T *tmp_0;
  int32_T i;
  uint32_T Integrator_ELAPS_T;
  tmp_0 = set_CD_integrator_DataTransfer();
  if (rtDwork.Integrator_RESET_ELAPS_T) {
    Integrator_ELAPS_T = 0U;
```

```
  } else {
    Integrator_ELAPS_T = (uint32_T)(get_tick_outside_CD_integrator() -
      rtDWork.Integrator_PREV_T);
  }

  rtDWork.Integrator_PREV_T = get_tick_outside_CD_integrator();
  rtDwork.Integrator_RESET_ELAPS_T = false;
  tmp = 1.25 * (real_T)Integrator_ELAPS_T;
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) {
      CD_measured.dti[i] += tmp * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.DiscreteTimeIntegrator_PREV_U[i] = (get_CD_integrator_InBus_u())[i];
}

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  memcpy(&tmp_0[0], &CD_measured.dti[0], (uint32_T)(10U * sizeof(real_T)));
}
```

The integrator function calls the timer service function `get_tick_outside_CD_integrator` to get the current tick value to use for calculating the current elapsed time and to cache the previous tick value.

The code generator assumes that the resolution of the clock is the fundamental step size of the model.

Services Header File

Services header file `services.h` declares prototypes for calling services that run as platform software in a target environment. This code fragment shows the prototype declaration for calling timer service function.

```
/* timer services */
extern uin32_T get_tick_outside_CD_integrator(void);
```

For an interface that uses the outside-execution data communication method, the code generator expects the target environment to provide a buffer for data communication. Because the target environment software creates and manages access to the buffer, the timer service function takes no arguments.

The function name reflects the naming rule defined for the timer service interface in the linked Embedded Coder Dictionary. The naming rule `$G$X` instructs the code generator to replace `$G` with the name of the service interface (`get_tick_outside`) and `$X` with the name of the calling entry-point function (`CD_integrator`).

Configure Timer Code Interface for During-Execution Data Access

Configure the example model such that the code generator applies the coder dictionary interface for during-execution access to the function clock tick value.

1. If the Embedded Coder app is not open, open it and select **Code Interface > Component Interface**.
2. In the Code Mappings editor, click the **Functions** tab.
3. Select the row for `Data Transfer`.

4. Select the row for `Exported Function:Aperiodic` and click the pencil icon.
5. In the dialog box that appears, set the **Timer Service** property to `get_tick_during`. As the name indicates, the default interface is configured to use during-execution data communication. Alternatively, you can set the property in the Property Inspector.
6. Save the configuration change by saving the model.

Regenerate and Reinspect Interface Code

Regenerate code from the example model.

The generated code shown in the following sections assumes that you have configured the inports, outports, data transfers, and the timer service for the integrator function with a service interface that applies the during-execution data communication method.

Header File

The lines of code relevant to the timer code interface in the header file `ComponentDeploymentFcn.h` are the same as for the outside-execution data transfer interface.

```
#include "services.h"
.
.
.
typedef struct {
  real_T DataTransfer_WriteBuf[10];
  real_T DiscreteTimeIntegrator_PREV_U[10];
  uint32_T Interator_PREV_T;
  uint8_T DiscreteTimeIntegrator_SYSTEM_E;
  boolean_T Integrator_RESET_ELAPS_T;
} D_Work;

typedef struct {
  real_T delay[10];
  real_T dti[10];
} CD_measured_T;
.
.
.
struct tag_RTM {
  struct {
    uint32_T clockTick2;
    .
    .
    .
  } Timing;
};
.
.
.
extern void CD_integrator(void);
```

Source Code File

When the model is configured for the code generator to produce interface code that uses during-execution data communication, the integrator entry-point function in the generated source code file `ComponentDeploymentFcn.c` calls target platform timer service function `get_tick_during_CD_integrator`.

When using the during-execution data communication method, calls to the service function take a pointer to a buffer in the generated code as an argument.

```
void CD_integrator(void)
{
  real_T tmp[10];
  real_T tmp_0;
  int32_T i;
  uint32_T Integrator_ELAPS_T;
  rtM->Timing.clockTick2 = get_tick_during_CD_integrator();
  if (rtDWork.Interator_RESET_ELAPS_T) {
    Integrator_ELAPS_T = 0U;
  } else {
    Integrator_ELAPS_T = (uint32_T)(rtM->Timing.clockTick2 -
      rtDWork.Integrator_PREV_T);
  }

  get_CD_integrator_input_(&tmp[0]);
  rtDWork.Integrator_PREV_T = rtM->Timing.clockTick2;
  rtDWork.Integrator_RESET_ELAPS_T = false;
  tmp_0 = 1.25 * (real_T)Integrator_ELAPS_T;
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) {
      CD_measured.dti[i] += tmp_0 * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.discreteTimeIntegrator_PREV_U[i] = tmp[i];
}

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  set_CD_integrator_DataTransfer(CD_measured.dti);
}
```

When using the during-execution data communication method, the generated code calls the timer service function `get_tick_during_CD_integrator` to get the current tick value to use for calculating the current elapsed time.

The code generator assumes that the resolution of the clock is the fundamental step size of the model.

Services Header File

This code fragment shows the prototype declaration for calling the timer service function when using during-execution data communication.

```
/* timer services */
extern uint32_T get_tick_during_CD_integrator(void);
```

Like when using the outside-execution data communication method, the code generator expects the target environment to provide a buffer for data communication. Because the target environment software creates and manages access to the buffer, the timer service function takes no arguments.

The function name reflects the naming rule defined for the timer service interface in the linked Embedded Coder Dictionary. The naming rule `$G$X` instructs the code generator to replace `$G` with the name of the service interface (`get_tick_during`) and `$X` with the name of the calling entry-point function (`CD_integrator`).

Related Examples

- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Generate C Parameter Tuning Service Interface Code for Component Deployment

This example shows how to generate component code that includes interface support for tunable parameters. The generated interface code must align with target environment interface requirements.

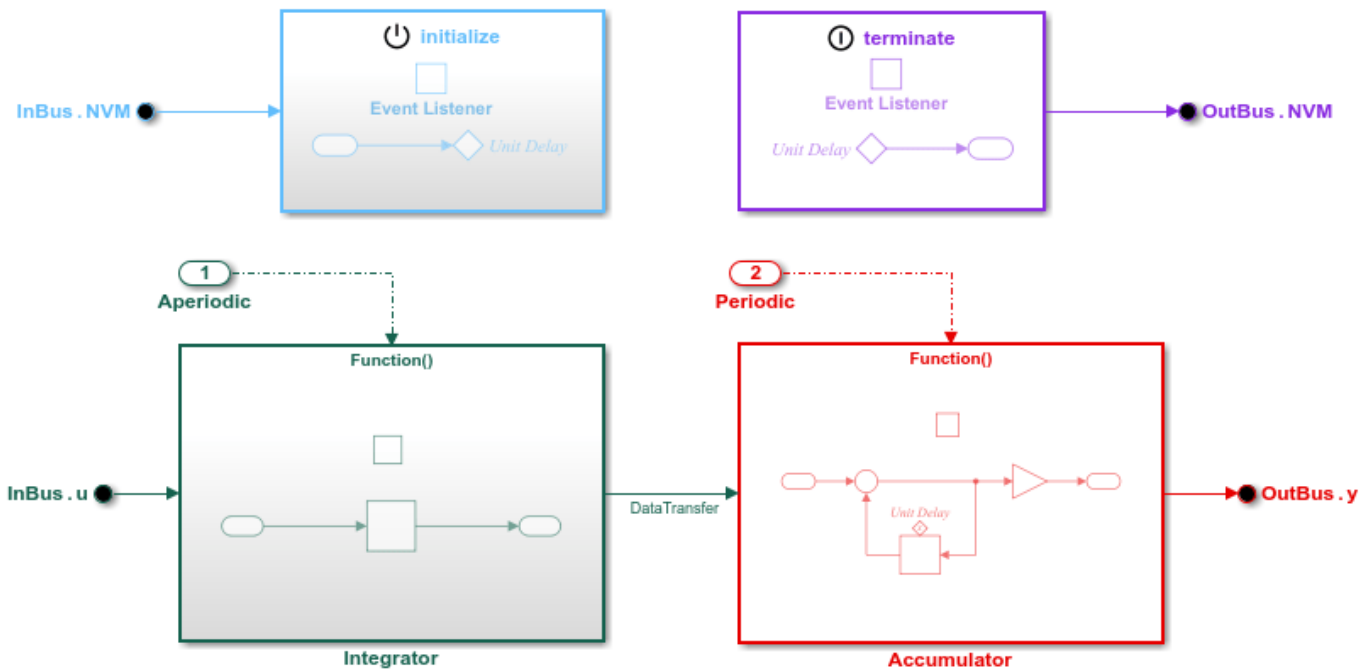
In the example you:

1. Represent a tunable parameter in the model.
2. Configure the code generator to apply a code interface.
3. Generate and inspect the interface code.

Represent Tunable Parameter in Top Model

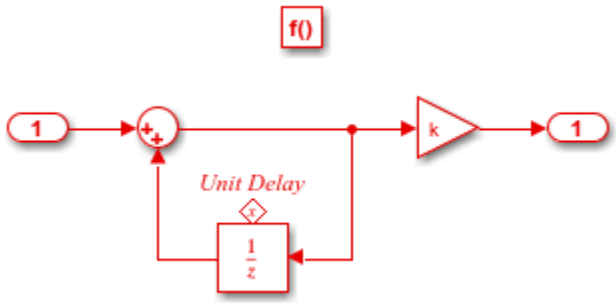
Open example component model `ComponentDeploymentFcn`.

```
open_system('ComponentDeploymentFcn');
```



The accumulator function receives the value of a state variable from the integrator function, increments the value by one, and applies a gain value, which is tunable during execution.

Open the function-call subsystem `Accumulator`.



In the subsystem, the tunable parameter is represented as the gain parameter `k`. The target environment expects the name of the corresponding variable in the generated code to be `CD_tunable.gain`.

Configure Tunable Parameter Code Interface

The example model is linked to the shared Embedded Coder Dictionary file `ComponentDeploymentCoderDictionary.sldd`, which defines a service code interface configuration. That configuration defines a parameter tuning service interface named `ParameterTuningService`. That service is configured to apply storage class `TuningStruct`. That storage class is defined with these property settings:

- **Access** is set to `Direct`.
- **Data Scope** is set to `Exported`.
- **Header File** is set to `$N.h`, where `$N` is the model name.
- **Definition File** is set to `$N.c`, where `$N` is the model name.
- **Use different property settings for single-instance and multi-instance data** is cleared.
- **Storage Type** is set to `Structured`.
- **Type Name** is set to naming rule `CD_tunable_T$M.`
- **Instance Name** is set to naming rule `CD_tunable`.
- **Data Initialization** is set to `Static`.
- **Memory Section** is set to `None`.
- **Preserve array dimensions** is cleared.
- No qualifiers are specified.

Configure the model such that the code generator applies the coder dictionary default interface for tunable parameters.

1. Open the Embedded Coder app.
2. Select **Code Interface** > **Component Interface**.
3. In the Code Mappings editor, click the **Parameters** tab.
4. Expand the **Model Parameters** node and select the row for `k`.
5. From the menu in the **Parameter Tuning Service** column of the selected row, select `Dictionary default: ParameterTuningService`.
6. In the selected row, click the pencil icon.

7. In the dialog box that appears, set the **Identifier** property to `gain`. Alternatively, you can set the property in the Property Inspector.
8. Save configuration changes by saving the model.

Generate and Inspect Interface Code

Generate code from the example model. The code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. Model data and entry-point functions are accessible to a caller by including the header file in target environment software.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with code interface requirements.

For a parameter tuning service interface, the report provides this information for each tunable parameter:

- Parameter source
- Code identifier
- Data type
- Dimension

Header File

The header file `ComponentDeploymentFcn.h` includes interface header file `services.h`, defines an instance of the real-time model data structure, and declares structures and callable entry-point functions for data and function interface elements represented in the model.

This code fragment shows code in the header file that is relevant to tunable parameters.

```
typedef struct {
  real_T k;
} CD_tunable_T;
.
.
.
extern void CD_accumulator(void);
.
.
.
extern CD_tunable_T CD_tunable;
```

Source Code File

This code fragment in the generated source code file `ComponentDeploymentFcn.c` shows code relevant to the parameter tuning service interface.

```
CD_tunable_T CD_tunable = {

  3.0
};
.
.
```

```
.
void CD_accumulator(void)
{
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += tmpIrvIRead[i];
    (getref_CD_accumulator_OutBus_y())[i] = CD_tunable.k * CD_measured.delay[i];
  }
}
```

The code initializes `CD_tunable` to a value of three. In the `for` loop, the accumulator function applies the gain value when computing the accumulation results.

Related Examples

- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Generate C Measurement Service Interface Code for Component Deployment

This example shows how to generate component code that includes interface support for measuring signal, state, and datastore data. The generated interface code must align with target environment interface requirements.

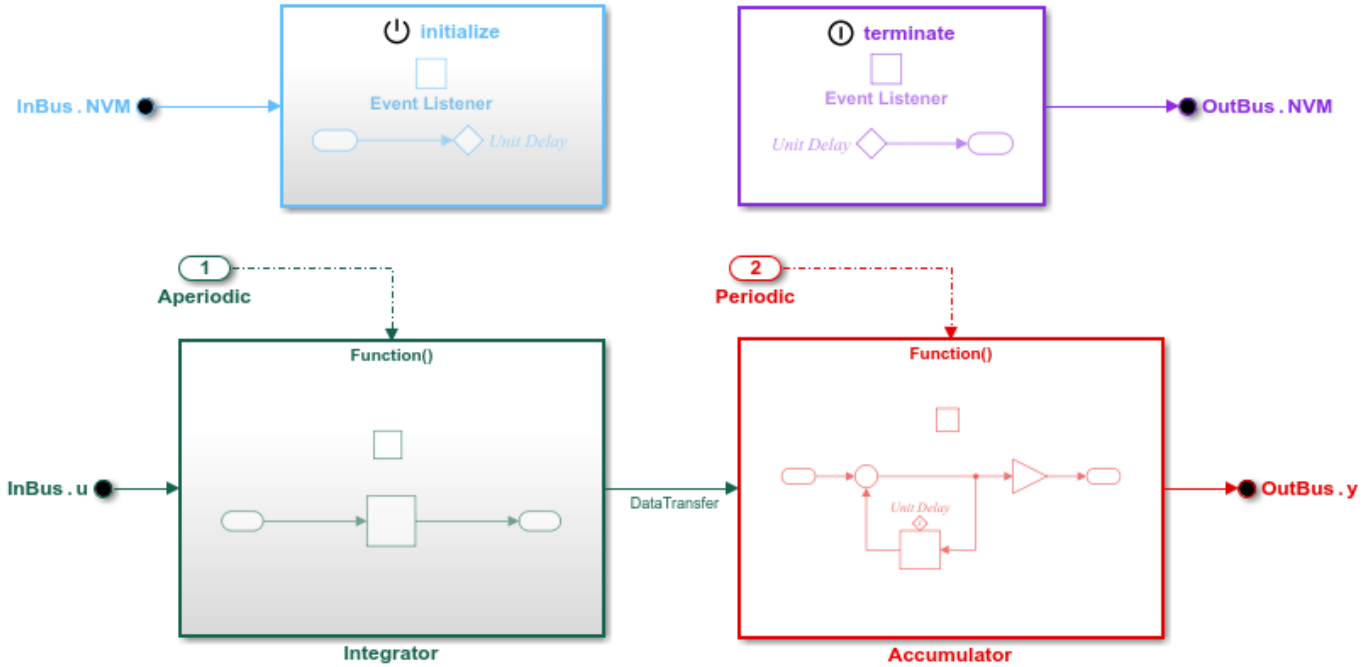
The example shows how to:

1. Represent a measurable signal, state, and datastore data in the top model.
2. Configure the code generator to apply a code interface.
3. Generate and inspect the interface code.

Represent Measurement Data in Top Model

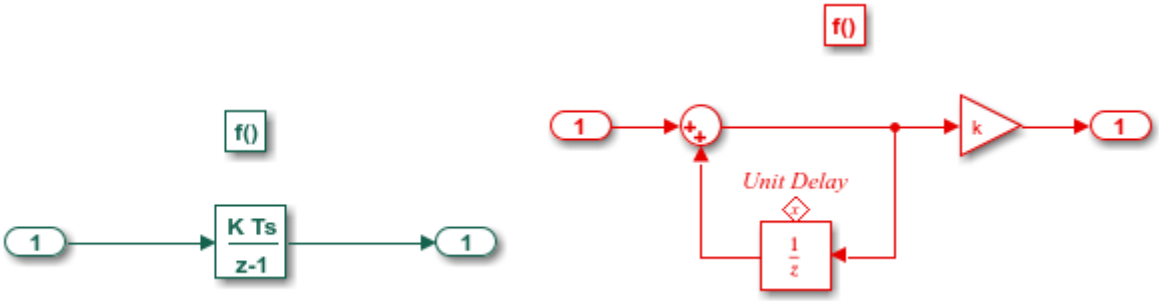
Open example component model `ComponentDeploymentFcn`.

```
open_system('ComponentDeploymentFcn');
```



The integrator and accumulator functions operate on state variables that you want to measure.

Open the function-call subsystems `Integrator` and `Accumulator`.



The integrator function uses a state variable to represent the discrete time integrator. The accumulator function uses a state variable to represent the delay. The target environment expects the names of the corresponding variables in the generated code to be `CD_measured.dti` and `CD_measured.delay`.

Configure Measurement Code Interface

The example model is linked to the shared Embedded Coder Dictionary file `ComponentDeploymentCoderDictionary.sldd`, which defines a service code interface configuration. That configuration defines a measurement service interface named `MeasurementService`. That service is configured to apply storage class `SignalStruct`. That storage class is defined with these property settings:

- **Access** is set to `Direct`.
- **Data Scope** is set to `Exported`.
- **Header File** is set to `$N.h`, where `$N` is the model name.
- **Definition File** is set to `$N.c`, where `$N` is the model name.
- **Use different property settings for single-instance and multi-instance data** is cleared.
- **Storage Type** is set to `Structured.`
- **Type Name** is set to naming rule `CD_measured_T$M`.
- **Instance Name** is set to naming rule `CD_measured`.
- **Data Initialization** is set to Dynamic.
- **Memory Section** is set to `None`.
- **Preserve array dimensions** is cleared.
- No qualifiers are specified.

Configure the model such that the code generator applies the coder dictionary default interface for measured data elements.

1. Open the Embedded Coder app.
2. Select **Code Interface** > **Component Interface**.
3. In the Code Mappings editor, click the **Signals/States** tab.
4. Expand the **States** node and select the row for `Discrete-Time Integrator`.
5. From the menu in the **Measurement Service** column of the selected row, select `Dictionary default: MeasurementService`.
6. In the selected row, click the pencil icon.

7. In the dialog box that appears, set the **Identifier** property to `dti`. Alternatively, you can set the property in the Property Inspector.
8. Select the row for `Unit Delay`.
9. Repeat steps six and seven to set the **Identifier** property for the state to `delay`.
10. Save configuration changes by saving the model.

Generate and Inspect Interface Code

Generate code from the example model. The code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. Model data and entry-point functions are accessible to a caller by including the header file in the target environment software.

To inspect the generated code, use the code generation report or, in the Embedded Coder app, use the Code view.

Header File

The header file `ComponentDeploymentFcn.h` includes interface header file `services.h`, defines an instance of the real-time model data structure, and declares structures and callable entry-point functions for data and function interface elements represented in the model.

This code fragment shows code in the header file that is relevant to measureable data.

```
typedef struct {
  real_T delay[10];
  real_T dti[10];
} CD_measured_T;
.
.
.
extern void CD_accumulator(void);
extern void CD_integrator(void);
extern CD_measured_T CD_measured;
```

Source Code File

This code fragment in generated source code file `ComponentDeploymentFcn.c` shows code relevant to the parameter tuning service interface.

```
CD_measured_T CD_measured;
.
.
.
void CD_accumulator(void)
{
  const real_T *tmpIrvIRead;
  int32_T i;
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
  for (i = 0; i < 10; i++) {
    CD_measured.delay[i] += tmpIrvIRead[i];
    (getref_CD_accumulator_OutBus_y())[i] = CD_tunable.k * CD_measured.delay[i];
  }
}
```

```
void CD_integrator(void)
{
  real_T tmp;
  real_T *tmp_0;
  int32_T i;
.
.
.
  tmp = 1.25 * (real_T)Integrator_ELAPS_T;
  for (i = 0; i < 10; i++) {
    if ((int32_T)rtDWork.DiscreteTimeIntegrator_SYSTEM_E == 0) }
      CD_measured.dti[i] += tmp * rtDWork.DiscreteTimeIntegrator_PREV_U[i];
    }

    rtDWork.DiscreteTimeIntegrator_PREV_U[i] = get_CD_integrator_InBus_u())[i];
  }

  rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 0U;
  memcpy(&tmp_0[0], &CD_measured.dti[0], (uint32_T)(10U * sizeof(real_T)));

void CD_initialize(void)
{
  {
    real_T *tmp;
    int32_T i;
    tmp = set_ModelInitialize_DataTransfer();
    for (i = 0; i < 10; i++) {
      tmp[i] = 5.0;
    }

    memcpy(&CD_measured.delay[0], &(get_CD_initialize_InBus_NVM())[0], (uint32_T)
            (10U * sizeof(real_T)));
    rtDWork.Integrator_RESET_ELAPS_T = true;
    rtDWork.DiscreteTimeIntegrator_SYSTEM_E = 1U;

  }
}

void CD_terminate(void)
{
  memcpy(&(getref_CD_terminate_OutBus_NVM())[0], &CD_measured.delay[0],
        (uint32_T)(10U * sizeof(real_T)));
}
```

The accumulator and integrator functions use the state variables `CD_states.dti` and `CD_states.delay` in the `for` loops. The integrator, initialize, and terminate functions, use the variables for the memory copy operations.

Related Examples

- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

More About

- “Embedded Coder Fundamentals”
- “Service Interfaces” on page 3-21

Configuring Service Interfaces in the Generated Code

- “Configure Sender and Receiver Service Interfaces for Model Inports and Outports” on page 35-2
- “Configure Data Transfer Service Interfaces for Data Transfer Signals” on page 35-5
- “Configure Timer Service Interfaces” on page 35-9
- “Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments” on page 35-13
- “Configure Measurement Service Interfaces for Signals, States, and Data Stores” on page 35-16

Configure Sender and Receiver Service Interfaces for Model Inports and Outports

Within a target environment, a component receives data from and sends data to other components by calling the target platform receiver and sender services. In a component model, use an Inport block or an In Bus Element block to represent calls to the receiver services. Use an Outport block or an Out Bus Element block to represent calls to the sender services. Use the **Code Mappings editor** to configure the sender and receiver service interfaces when you do not want to use the default value specified in a shared coder dictionary.

You can configure the data communication methods of receiver and sender service interfaces for inputs and outputs to one of the following:

- *Outside execution*: The platform communicates data with other functions outside (before and after) function execution.
- *During execution*: The platform service communicates data with other functions immediately during function execution.
- *Direct access*: Generated function code communicates with other functions directly by using memory that platform services manage for the target execution environment.

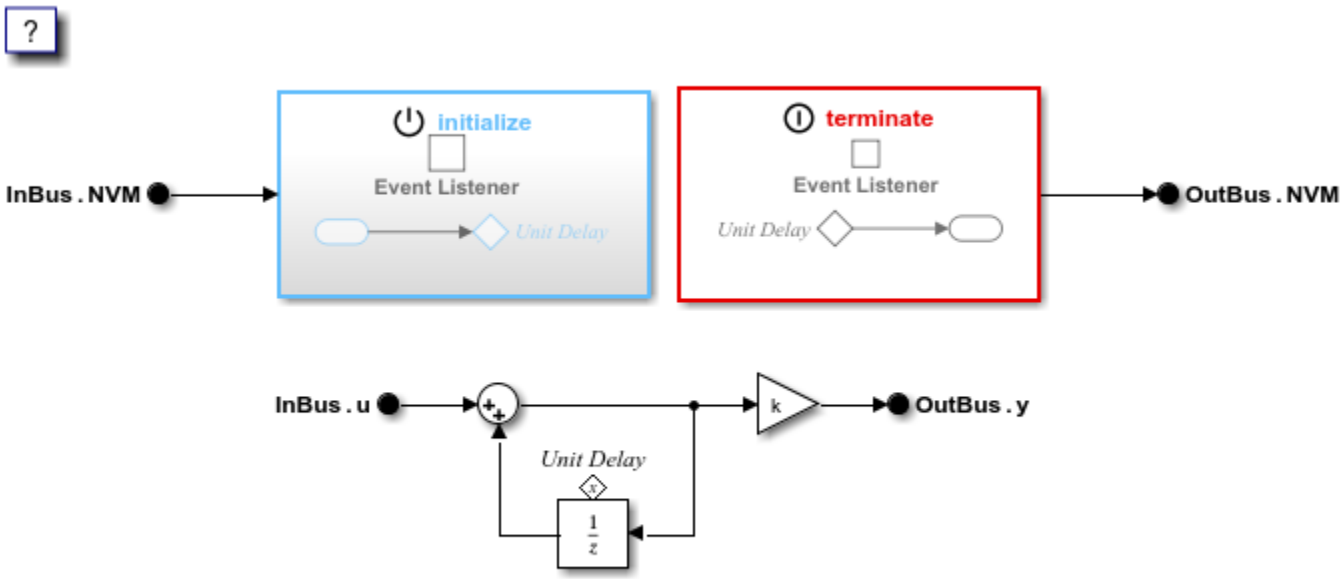
For more information on data communication methods, see “Data Communication Methods” on page 1-14.

Configure Sender and Receiver Service Interfaces by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default sender and receiver service interfaces for model outputs and inputs, respectively. To configure a model input or output to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentRate`.

```
openExample('ComponentDeploymentRate');
```



Copyright 2022 The MathWorks, Inc.

In this model, the component inputs are represented by In Bus Element blocks, `InBus.u` and `InBus.NVM`. Component outputs are represented by Out Bus Element blocks, `OutBus.y` and `OutBus.NVM`.

Verify that the component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`.

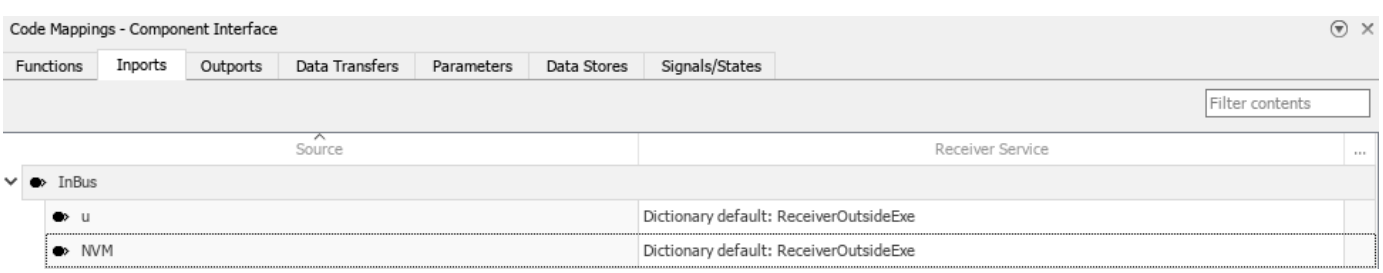
In the Model Configuration Parameters dialog box, verify that the **Shared coder dictionary** parameter is set to `ComponentDeploymentCoderDictionary.sldd`. This dictionary defines default service interfaces, including the default sender and receiver service interfaces for the inputs and outputs of the component model.

Open the Embedded Coder app.

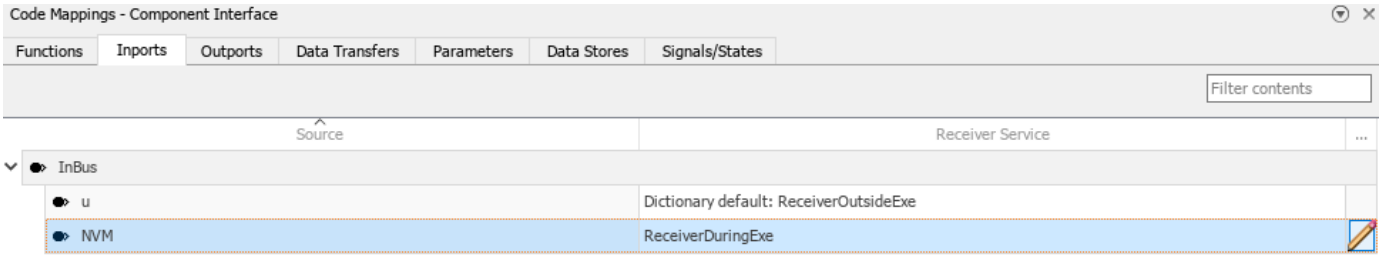
To configure an input or output to use a receiver or sender service interface other than the default defined by the coder dictionary, use the Code Mappings editor.

To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

On the Inports tab, notice that the **Receiver Service** of the component inputs, `InBus.u` and `InBus.NVM`, is set to `Dictionary default: ReceiverOutsideExecution`.



To configure the input `InBus.NVM` to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, on the **Inports** tab of the Code Mappings editor, set the **Receiver Service** of `InBus.NVM` to `ReceiverDuringExe`.



Configure Sender and Receiver Service Interfaces Programmatically

To configure the sender and receiver service interfaces of a component programmatically, access the code mapping properties of the component model outputs and inputs, respectively.

Open the model.

```
openExample('ComponentDeploymentRate');
```

Get the code mappings for the model by using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get('ComponentDeploymentRate');
```

Get the current receiver service of the input, `InBus.NVM`, by using the `getInport` function.

```
receiver = getInport(cm, 'InNVM', 'ReceiverService')
```

```
receiver =

    'Dictionary default'
```

The model input is configured to use the default receiver service interface defined in the shared coder dictionary.

To configure the input `InBus.NVM` to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, use the `setInport` function and set the `ReceiverService` property to `ReceiverDuringExe`.

```
setInport(cm, 'InNVM', 'ReceiverService', 'ReceiverDuringExe')
```

See Also

Code Mappings Editor | `coder.mapping.api.CodeMapping` | `getInport` | `setInport`

Related Examples

- “Receiver and Sender Service Interfaces” on page 3-21
- “Sample Target Environment Service Implementations of Data Communication Methods” on page 75-2
- “Configure Code Interfaces for Code Generation” on page 3-4

Configure Data Transfer Service Interfaces for Data Transfer Signals

Within a target environment, execution entry-point functions communicate with other execution entry-point functions by calling the target platform data transfer services. In a component model, signal lines connecting communicating functions represent data transfers in the generated code. Communicating functions are modeled by using Function-Call Subsystem blocks that are aperiodic or periodic functions. Use the **Code Mappings editor** to configure the data transfer service interfaces when you do not want to use the default value specified in a shared coder dictionary.

You can configure the data transfer service interfaces to one of the following:

- *Outside execution*: The platform communicates data with other functions outside (before and after) function execution.
- *During execution*: The platform service communicates data with other functions immediately during function execution.

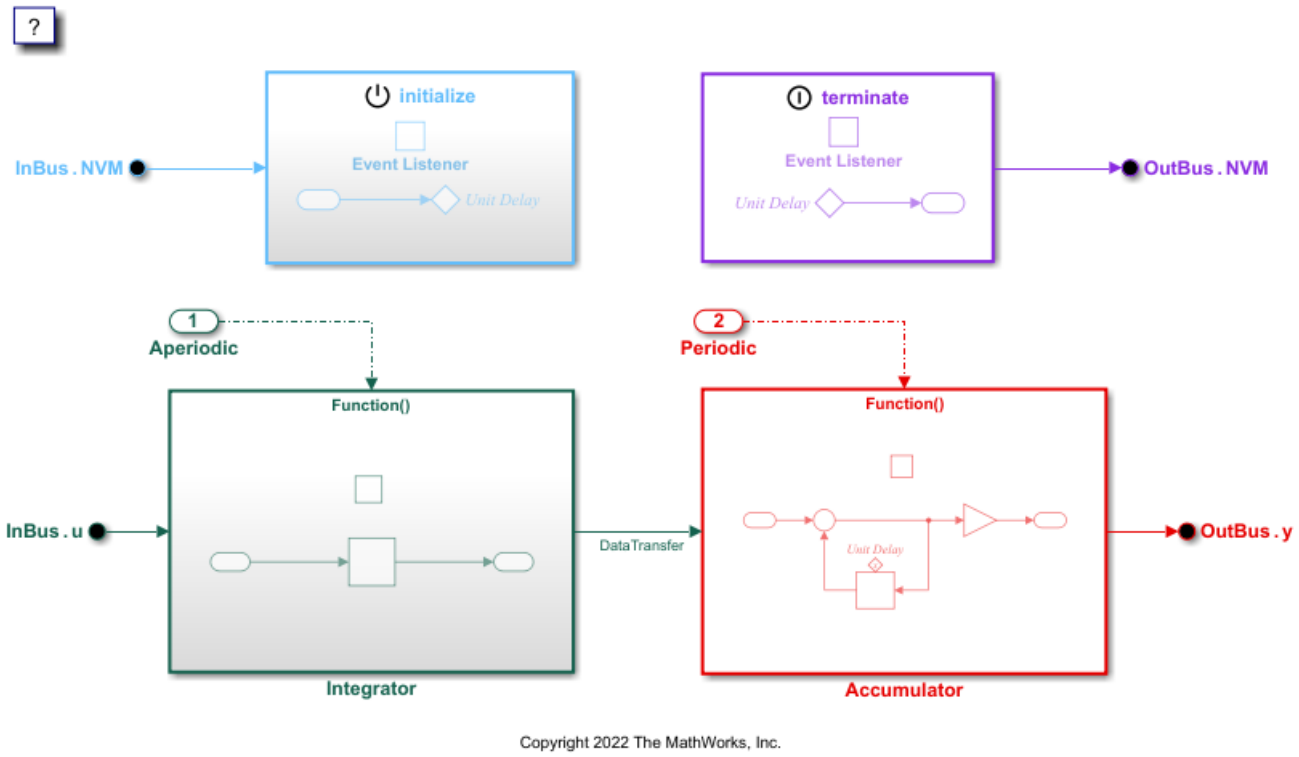
For more information on data communication methods, see “Data Communication Methods” on page 1-14.

Configure Data Transfer Service Interfaces by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default data transfer service interface for signals representing data transfers. To configure the data transfer to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentFcn`.

```
openExample('ComponentDeploymentFcn');
```



In this model, the data transfer is represented by the signal connecting the two Function-Call Subsystem blocks, `Accumulator` and `Integrator`.

Verify that the component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`.

In the Model Configuration Parameters dialog box, verify that the **Shared coder dictionary** parameter is set to `ComponentDeploymentCoderDictionary.sldd`. This dictionary defines default service interfaces, including the default data transfer service interface, of the component model.

Open the Embedded Coder app.

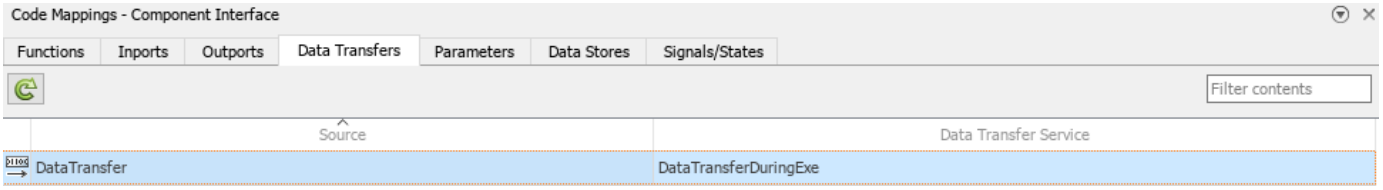
To configure a signal representing a data transfer to use a data transfer service interface other than the default defined by the shared coder dictionary, use the Code Mappings editor.

To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

To populate the data transfer mappings, update the model diagram.

On the **Data Transfers** tab, notice that the **Data Transfer Service** of the signal connecting the Function Call Subsystems, `DataTransfer`, is set to `Dictionary default: DataTransferOutsideExe`.

To configure the signal to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, on the **Data Transfers** tab of the Code Mappings editor, set the **Data Transfer Service** of `DataTransfer` to `DataTransferDuringExe`.



Configure Data Transfer Service Interfaces Programmatically

To configure the data transfer service interfaces of a component programmatically, access the code mapping properties of signals representing data transfers in the component model.

Open the model.

```
openExample('ComponentDeploymentFcn');
```

Get the code mappings for the model by using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get('ComponentDeploymentFcn');
```

In this model, the signal connecting the two Function-Call Subsystem blocks, `Accumulator` and `Integrator` represents a data transfer. Get the current data transfer service interface configuration by using the `getDataTransfer` function.

```
ph = get_param('ComponentDeploymentFcn/Integrator', 'PortHandles');
dataTransServ = getDataTransfer(cm, ph.Outport, 'DataTransferService')
```

```
dataTransServ =

    'Dictionary default'
```

The signal representing the data transfer is configured to use the default data transfer service interface defined in the shared coder dictionary.

To configure the signal to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, use the `setDataTransfer` function and set the `DataTransferService` property to `DataTransferDuringExe`.

```
setInport(cm, ph.Outport, 'DataTransferService',...
 'DataTransferDuringExe')
```

See Also

Code Mappings Editor | `coder.mapping.api.CodeMapping`

Related Examples

- “Data Transfer Service Interfaces” on page 3-22
- “Sample Target Environment Service Implementations of Data Communication Methods” on page 75-2
- “Configure Code Interfaces for Code Generation” on page 3-4
- “Generate C Data Transfer Service Interface Code for Component Deployment” on page 34-6

Configure Timer Service Interfaces

For component deployment models, you can generate code that accesses the function clock tick provided by the target environment. A function clock tick value is the number of time ticks from the start of program execution to the present time for a function. Generated code for a function can access the target platform function clock tick value by calling a function provided by the platform timer service.

You represent requests for the function clock tick in a model implicitly when you include blocks that rely on elapsed or absolute time. The code generator produces code that calls a timer service function, depending on the block context (for example, for a block that relies on elapsed time and is used in an aperiodic context). If you do not want to use the default timer service interface specified in the Embedded Coder Dictionary, you can configure the timer service interface by using the Code Mappings Editor or code mappings programming interface.

Timer service interfaces can be configured to access timer services:

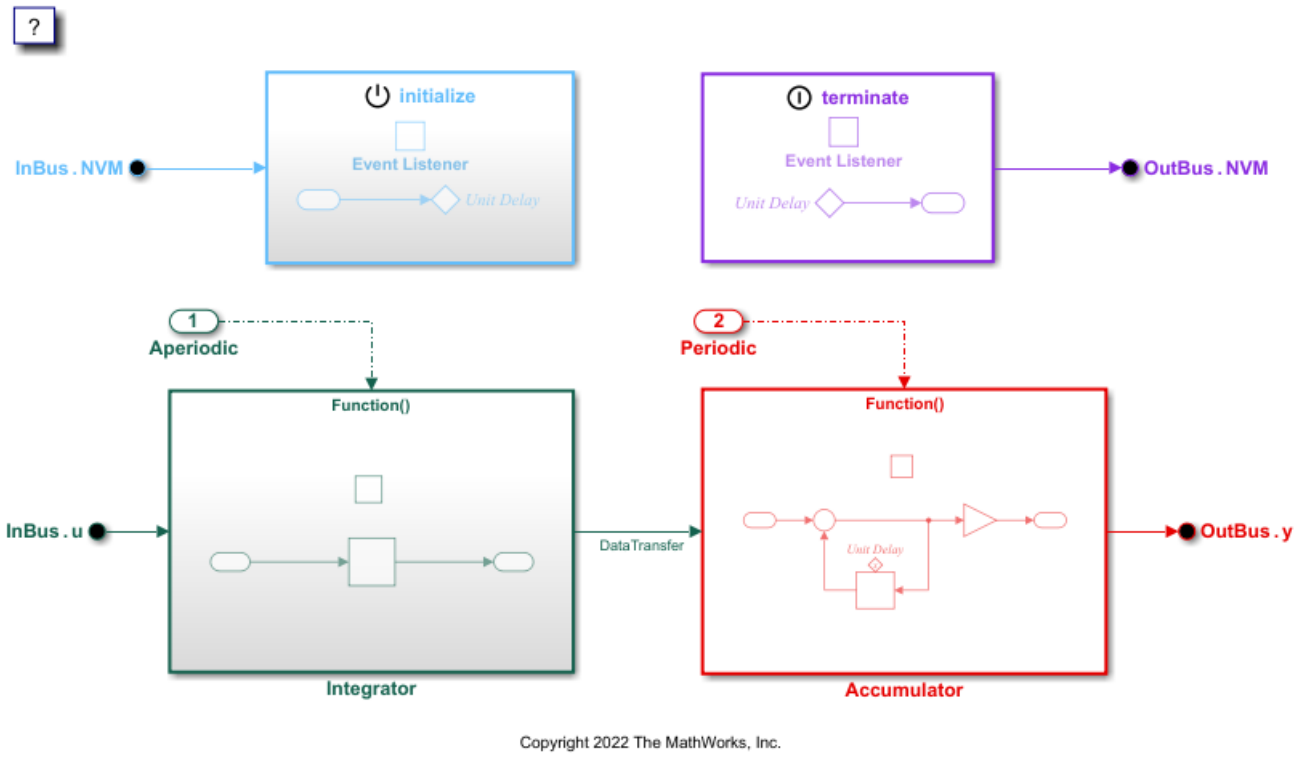
- *Outside execution*: The platform service communicates clock tick value outside (before and after) function execution.
- *During execution*: The platform communicates clock tick value immediately during function execution.

Configure Timer Service Interfaces by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default timer service interfaces for exported functions in the component model. To configure the timer to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentFcn`.

```
openExample('ComponentDeploymentFcn');
```



In this model, the Function-Call Subsystem block `Integrator` uses a timer service because it contains a Discrete-Time Integrator block.

Verify that the component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`.

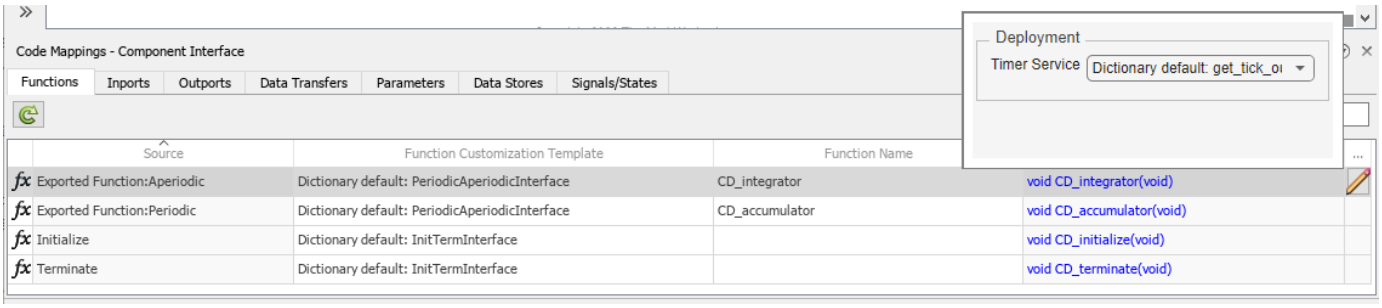
In the Model Configuration Parameters dialog box, verify that the **Shared coder dictionary** parameter is set to `ComponentDeploymentCoderDictionary.sldd`. This dictionary defines default service interfaces, including the timer services, of the component model.

Open the Embedded Coder app.

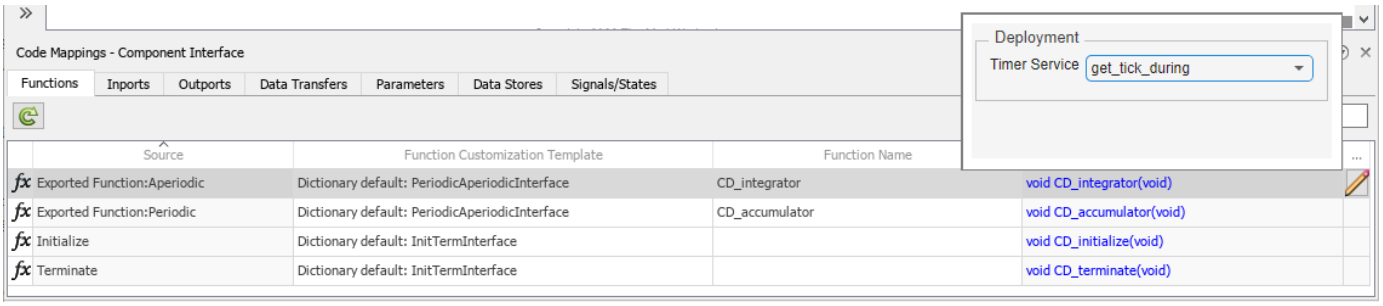
To configure the integrator function to use a timer service interface other than the default defined by the shared coder dictionary, use the Code Mappings editor.

To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

On the **Functions** tab, on the table row for the `Exported Function:Aperiodic` function, click the pencil icon. The **Timer Service**, is set to `Dictionary default: get_tick_outside`.



To configure the function to access clock tick value immediately during function execution instead of before or after execution as specified in the dictionary, set the **Timer Service** of `Exported Function:Aperiodic` to `get_tick_during`.



Configure Timer Service Interfaces Programmatically

To configure timer services of a function component programmatically, access the code mapping properties of the component model.

Open the model.

```
openExample('ComponentDeploymentFcn');
```

Get the code mappings for the model by using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get('ComponentDeploymentFcn');
```

In this model, the exported function `Aperiodic`, represented by the Function-Call Subsystem, `Integrator`, requires timer services in the platform environment. Get the current timer service interface configuration by using the `getFunction` function.

```
getFunction(cm, 'ExportedFunction:Aperiodic', 'TimerService')
```

```
ans =

    'Dictionary default'
```

The function is configured to use the default timer service interface defined in the shared coder dictionary.

To configure the function to access timer services during function execution instead of before or after execution as specified in the dictionary, use the `setFunction` function and set the `TimerService` property to `get_tick_during`, which is defined in the dictionary.

```
setFunction(cm, 'ExportedFunction:Aperiodic',...
 'TimerService', 'get_tick_during')
```

See Also

Code Mappings Editor | `coder.mapping.api.CodeMapping` | `getFunction` | `setFunction`

Related Examples

- "Timer Service Interfaces" on page 3-25
- "Generate C Timer Service Interface Code for Component Deployment" on page 34-13

Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments

Some components require interface support for tunable parameters and parameter arguments. When a parameter or parameter argument is tunable, you can change the value that is stored in memory during program execution.

To configure parameter and parameter argument tuning service interfaces, configure how a parameter or parameter argument is stored in memory during program execution by defining a storage class in the dictionary.

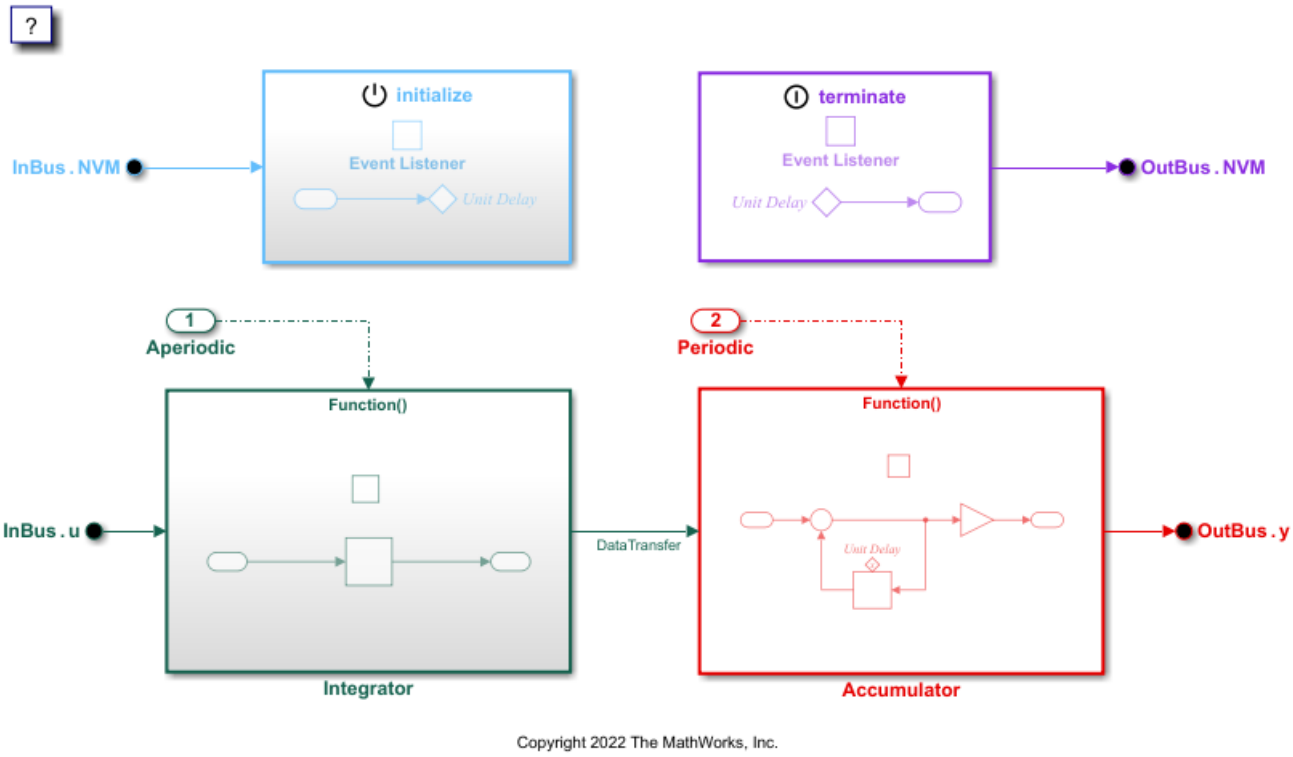
Use the **Code Mappings editor** to configure the parameter tuning service interface when you do not want to use the default value specified in a shared coder dictionary.

Configure Parameter Tuning Service Interfaces by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default parameter tuning service interfaces for model parameters. To configure the model to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentFcn`.

```
openExample('ComponentDeploymentFcn');
```



In this model, model parameter `k` is tunable and defines the gain of the Gain block within the `Accumulator` Function-Call Subsystem block.

Verify that the component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`.

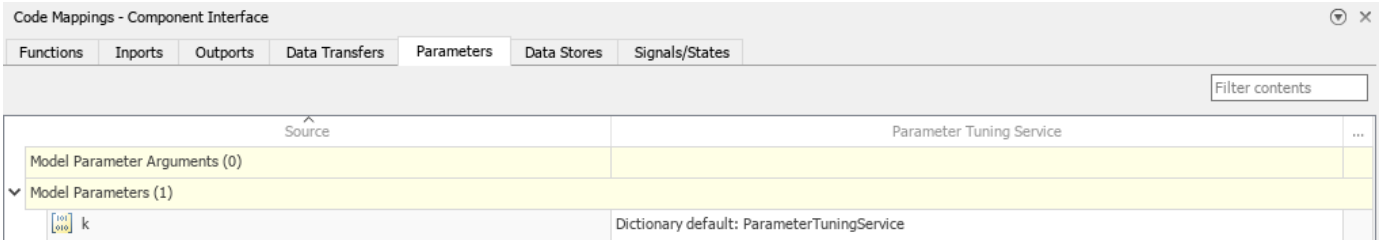
In the Model Configuration Parameters dialog box, verify that the **Shared coder dictionary** parameter is set to `ComponentDeploymentCoderDictionary.sldd`. This dictionary defines default service interfaces, including parameter tuning service interfaces, of the component model.

Open the Embedded Coder app.

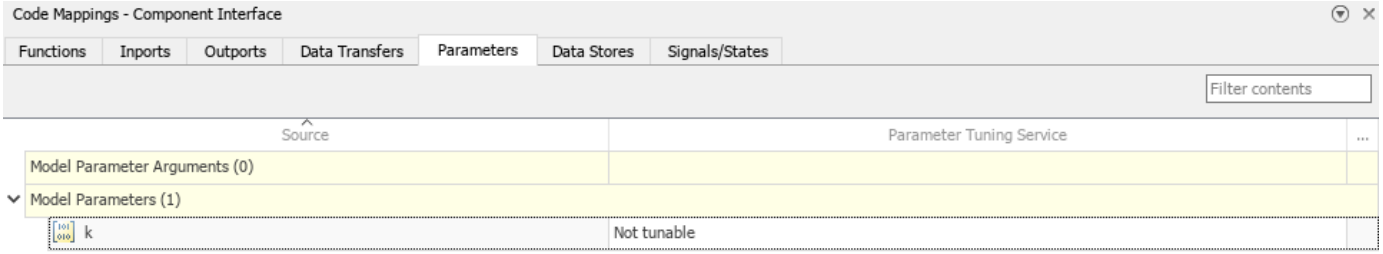
To configure the functions to use a parameter tuning service interface other than the default defined by the shared coder dictionary, use the Code Mappings editor.

To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

On the **Parameters** tab, notice that the **Parameter Tuning Service** of the parameter `k`, is set to `Dictionary default: ParameterTuningService`.



To specify that a parameter or parameter argument is not tunable and therefore does not need a tuning service interface, set the **Parameter Tuning Service** to `Not tunable`.



To configure the parameter to use a tuning service interface other than the default defined by the dictionary, you must create a new storage class in the dictionary, and reference the storage class in a new parameter tuning service interface definition in the dictionary.

For information on how to create a new parameter tuning service interface definition in the dictionary, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

Configure Parameter Tuning Service Interfaces Programmatically

To configure parameter tuning service interfaces of a parameter programmatically, access the code mapping properties of the component model.

Open the model.

```
openExample('ComponentDeploymentFcn');
```

Get the code mappings for the model by using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get('ComponentDeploymentFcn');
```

In this model, model parameter `k` is tunable and defines the gain of the Gain block within the `Accumulator` Function-Call Subsystem block. Get the current parameter tuning service interface configuration by using the `getModelParameter` function.

```
getModelParameter(cm, 'k', 'ParameterTuningService')
```

```
ans =

    'Dictionary default'
```

The parameter is configured to use the default parameter tuning service interface defined in the shared coder dictionary.

To specify that a parameter or parameter argument is not tunable and therefore does not need a tuning service interface, use the `setModelParameter` function to set the parameter tuning service interface to `Not tunable`.

```
setModelParameter(cm, 'k', 'ParameterTuningService', 'Not tunable')
```

To configure the parameter to use a different parameter tuning service interface, create a new parameter tuning service interface definition in the dictionary. Then, use the `setModelParameter` function and specify the `ParameterTuningService` property to the new service interface you defined in the dictionary.

See Also

Code Mappings Editor | `coder.mapping.api.CodeMapping` | `getModelParameter` | `setModelParameter`

Related Examples

- “Parameter and Parameter Argument Tuning Service Interfaces” on page 3-27
- “Generate C Parameter Tuning Service Interface Code for Component Deployment” on page 34-20
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2

Configure Measurement Service Interfaces for Signals, States, and Data Stores

By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To configure measurement service interfaces, configure how the variable is stored in memory during program execution by defining a storage class in the shared coder dictionary. The dictionary specifies the default measurement service interface. Use the **Code Mappings editor** to configure the measurement service interfaces when you do not want to use the default value specified in a shared coder dictionary.

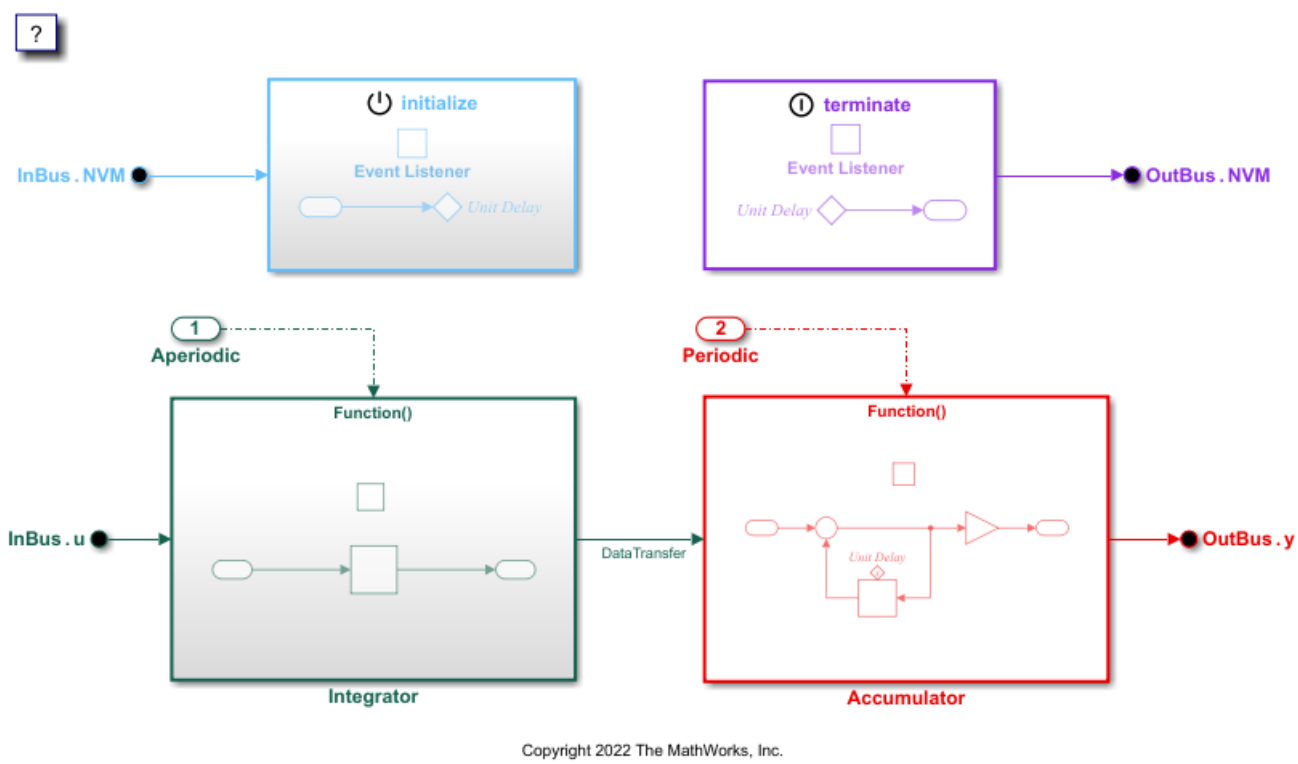
If data from a signal, state, or data store does not need to be preserved in the generated code, in the Code Mappings editor, you can set the measurement service interface to `Not measured`. This setting can be useful for debugging.

Configure Measurement Service Interfaces for States by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default measurement service interfaces for states. To configure the model to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentFcn`.

```
openExample('ComponentDeploymentFcn');
```



In this model, the Unit Delay block within the `Accumulator` Function-Call Subsystem, and the Discrete-Time Integrator block within the `Integrator` Function-Call Subsystem each have a state.

Verify that the component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`.

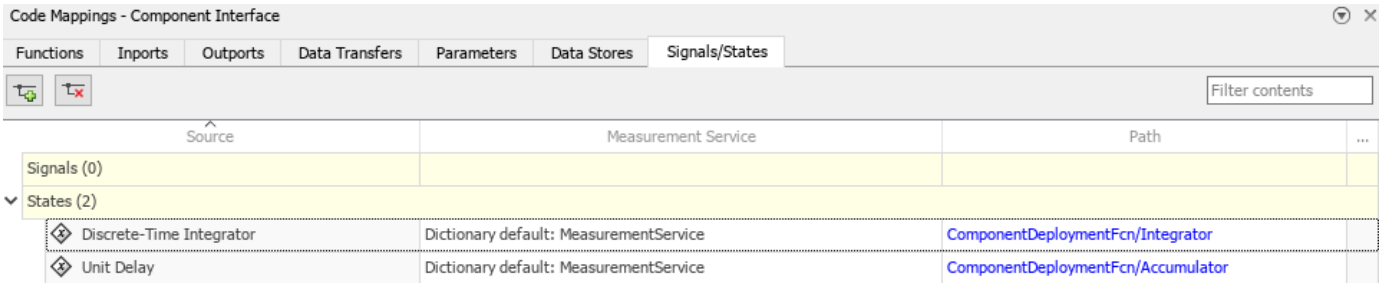
In the Model Configuration Parameters dialog box, verify that the **Shared coder dictionary** parameter is set to `ComponentDeploymentCoderDictionary.sldd`. This dictionary defines default service interfaces, including the measurement service interface, of the component model.

Open the Embedded Coder app.

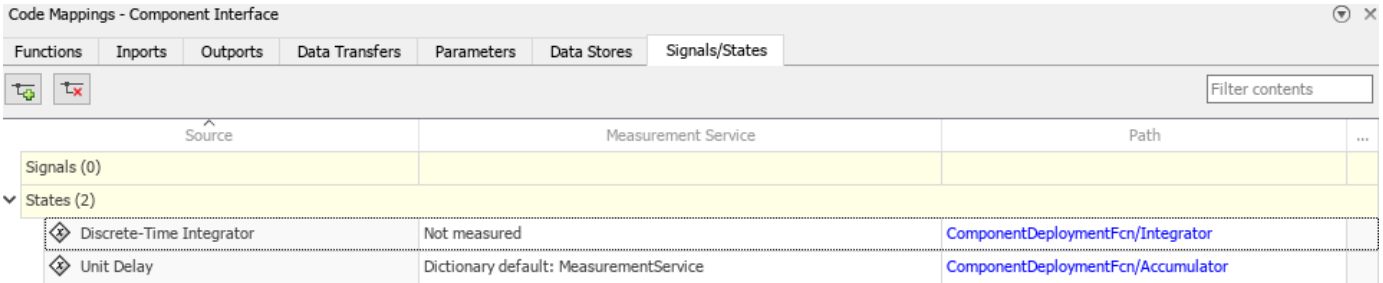
To configure the block states to use a measurement service interface other than the default defined by the shared coder dictionary, use the Code Mappings editor.

To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

On the **Signals/States** tab, notice that the **Measurement Service** of the two block states, is set to `Dictionary default: MeasurementService`.



To specify that a signal, state, or data store does not require a measurement service interface, set the **Measurement Service** to `Not measured`.



To configure the block states to use a measurement service interface other than the default defined by the dictionary, you must create a new storage class in the dictionary, and reference the new storage class in a new measurement service interface definition in the dictionary.

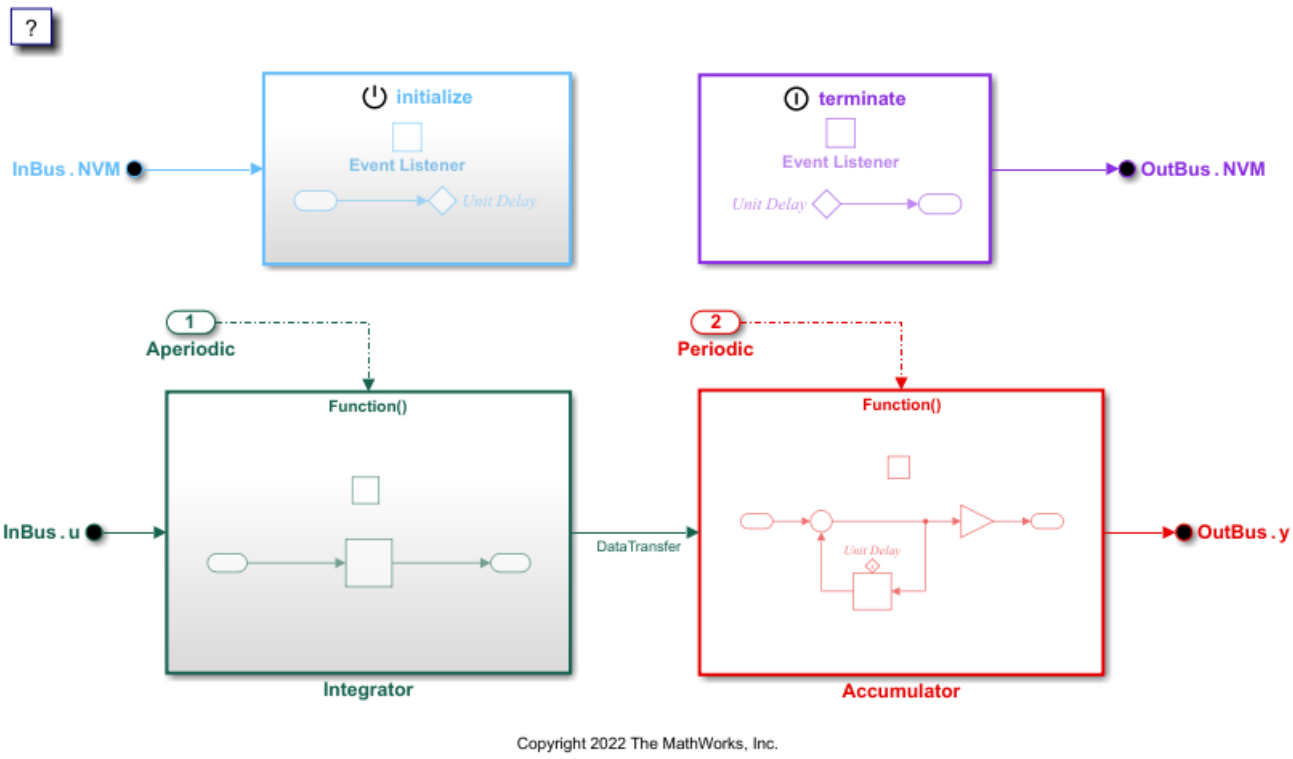
For more information, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

Configure Measurement Service Interfaces for Signals by Using the Code Mappings Editor

In this example, the component model references a shared coder dictionary that defines the default measurement service interface for signals. To configure the model to use something other than the default service interface, use the **Code Mappings editor**.

Open the component model, `ComponentDeploymentFcn`.

```
openExample('ComponentDeploymentFcn');
```



The component model references the shared coder dictionary, `ComponentDeploymentCoderDictionary.sldd`, which defines default service interfaces, including the measurement services, of the component model.

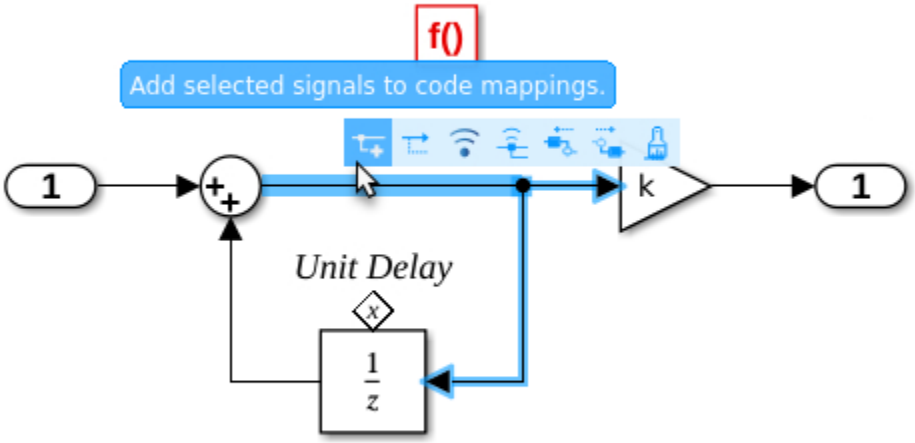
Open the Embedded Coder app.

To configure signals to use a measurement service interface other than the default defined by the shared coder dictionary, use the Code Mappings editor.

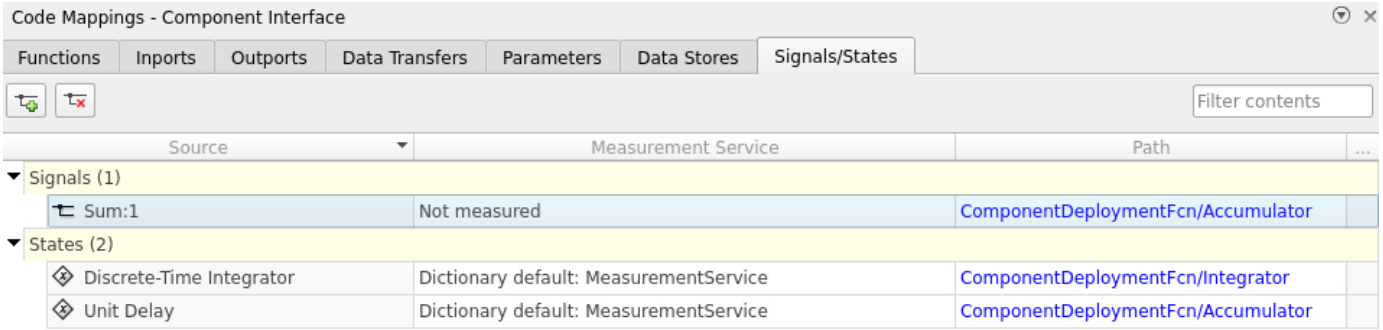
To open the Code Mappings editor, in the **C Code** tab, click **Code Interface** > **Component Interface**.

Open the **Signals/States** tab. Currently there are no signals in the mapping. To add a signal to the mapping, click on the signal and select **Add selected Signals to code mappings**. In this example,

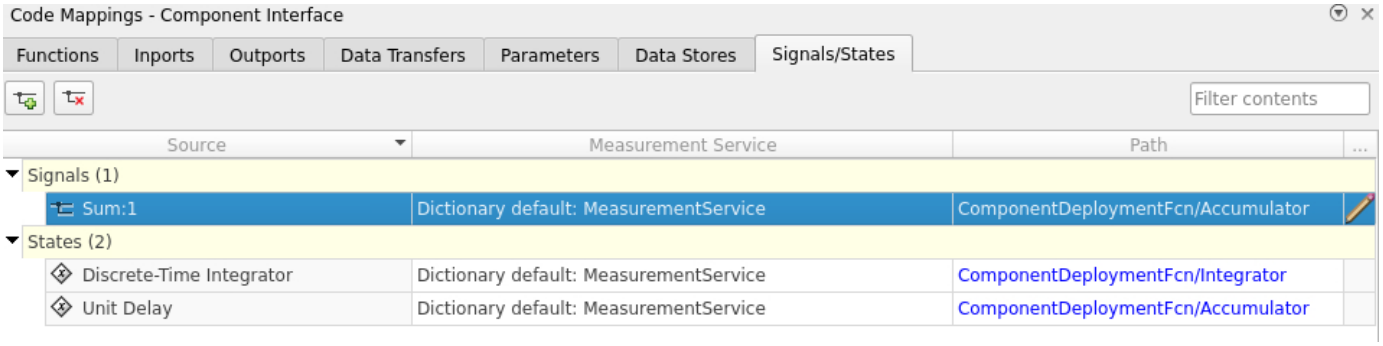
add the signal at the output of the Sum block in the `Accumulator` subsystem to the code mappings of the model.



The signal now appears in the Code Mappings editor. The **Measurement Service** property of the signal is set to `Not measured`.



To specify that the signal use the default measurement service interface as defined in the dictionary, set the **Measurement Service** to `Dictionary default: MeasurementService`.



Configure Measurement Service Interfaces for States and Signals Programmatically

To configure the measurement service interface of a signal, state, or data store programmatically, access the code mapping properties of the component model.

Open the model.

```
openExample('ComponentDeploymentFcn');
```

Get the code mappings for the model by using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get('ComponentDeploymentFcn');
```

In this model, the Unit Delay block within the `Accumulator` Function-Call Subsystem, and the Discrete-Time Integrator block within the `Integrator` Function-Call Subsystem each have an associated state. Get the current measurement service interface configuration by using the `getState` function.

```
getState(cm, ...
'ComponentDeploymentFcn/Integrator/Discrete-Time Integrator',...
 'MeasurementService')

ans =

    'Dictionary default'
```

The block state is configured to use the default measurement service interface defined in the shared coder dictionary.

To specify that a signal, state, or data store does not require a measurement service interface, set the measurement service interface to `'Not measured'`.

```
setState(cm,...
 'ComponentDeploymentFcn/Integrator/Discrete-Time Integrator',...
 'MeasurementService', 'Not measured')
```

To configure the block state to use a different measurement service interface, create a new measurement service interface definition in the dictionary. Then, use the `setState` function and specify the `MeasurementService` property to the new service interface you defined in the dictionary.

To configure the measurement service of the signal at the output of the Sum block, add the signal to the code mappings by using the `addSignal` function and specify the measurement service interface property.

```
portHandle = get_param('ComponentDeploymentFcn/Accumulator/Sum',...
 'PortHandles');
addSignal(cm, portHandle.Outport, ...
'MeasurementService', 'Dictionary default')
```

See Also

`getState` | `setState` | `getDataStore` | `setDataStore` | `getSignal` | `setSignal`

Related Examples

- “Measurement Service Interfaces” on page 3-28
- “Generate C Measurement Service Interface Code for Component Deployment” on page 34-24

Data Representation in Simulink Coder

- “Preserve Variables in Generated Code” on page 36-2
- “Access Signal, State, and Parameter Data During Execution” on page 36-17
- “Data Structures in the Generated Code” on page 36-31
- “How Generated Code Exchanges Data with an Environment” on page 36-35
- “Control Data and Function Interface in Generated Code” on page 36-44
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Choose Data Configuration Approach” on page 36-70
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Use Enumerated Data in Generated Code” on page 36-86
- “Data Stores in Generated Code” on page 36-94
- “Specify Single-Precision Data Type for Embedded Application” on page 36-104
- “Tune Phase Parameter of Sine Wave Block During Code Execution” on page 36-107
- “Switch Between Output Waveforms During Code Execution for Waveform Generator Block” on page 36-109
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “Limitations for Block Parameter Tunability in Generated Code” on page 36-122
- “Code Generation of Parameter Objects With Expression Values” on page 36-125
- “Configure Packaging of Parameter Arguments in Generated Code” on page 36-127
- “Parameter Data Types in the Generated Code” on page 36-131
- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 36-135
- “Reuse Parameter Data in Different Data Type Contexts” on page 36-143
- “Organize Data into Structures in Generated Code” on page 36-147
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 36-161
- “Design Data Interface by Configuring Inport and Outport Blocks” on page 36-169
- “Generate Efficient Code for Bus Signals” on page 36-173
- “Control Signal and State Initialization in the Generated Code” on page 36-176
- “Initialization of Signal, State, and Parameter Data in the Generated Code” on page 36-187
- “Optimize Generated Code Using Fixed-Point Data with Simulink, Stateflow, and MATLAB” on page 36-196
- “Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box” on page 36-197
- “Share Data Between Code Generated from Simulink, Stateflow, and MATLAB” on page 36-200
- “Air-Fuel Ratio Control System with Fixed-Point Data” on page 36-202

Preserve Variables in Generated Code

As you iteratively develop a model, you capture the values of signals and states that model execution generates. You also tune parameter values during execution to observe results on the signals and states. You can then base your design decisions upon analysis of these outputs. To access signal, state, and parameter data in a rapid prototyping environment, you can configure the generated code to store the data in addressable memory.

Depending on the system target file that you use (such as `grt.tlc`), by default, optimization settings can make the generated code more efficient by:

- Eliminating unnecessary global and local storage for signals and some states.

  However, the optimizations do not eliminate storage for root-level Inport and Outport blocks, which represent the primary inputs and outputs of a system. To access this data, you do not need to take optimizations into consideration.
- Inlining the numeric values of block parameters. The code does not store the parameters in memory, so you cannot interact with the parameters during code execution.

To generate code that instead allocates addressable memory for this data, you can disable the optimizations, or override the optimizations by specifying code generation settings for individual data items.

Access Signal, State, and Parameter Data During Execution

As you iteratively develop a model, you capture output signal and state data that model execution generates. You also tune parameter values during execution to observe results on the outputs. You can then base your design decisions upon analysis of these outputs. To access this signal, state, and parameter data in a rapid prototyping environment, you can configure the generated code to store the data in addressable memory.

By default, optimization settings make the generated code more efficient by eliminating unnecessary signal storage and inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for this data, you can disable the optimizations or specify code generation settings for individual data items.

Explore Example Model

Run a script that prepares the model rtwdemo_configrpinterface for this example.

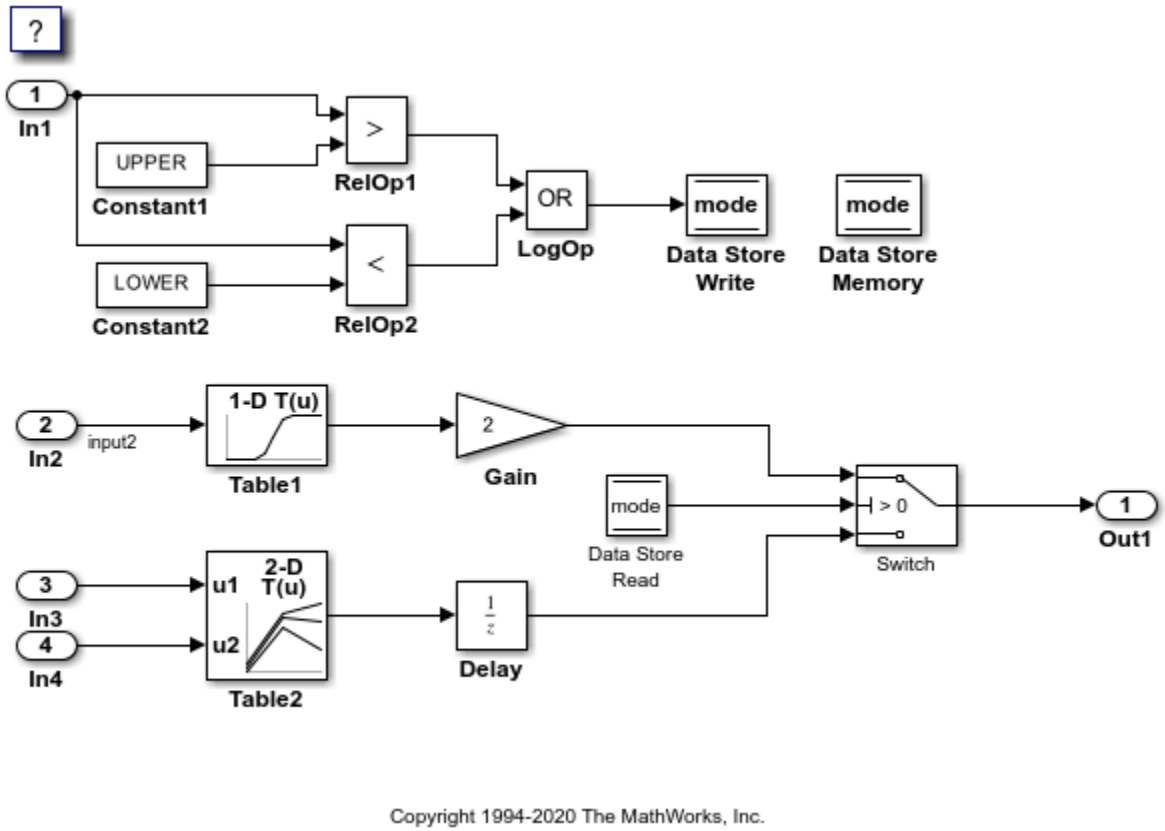
```
prepare_rtwdemo_configrpinterface
```

This data is configured for code generation:

- Parameters `UPPER`, `LOWER`, `K1`, `Table1`, and `Table2`
- Signals `In1`, `In2`, `In3`, `In4`, and `Out1`
- States `X` (delay) and `mode` (data store memory write)

Open the example model, `rtwdemo_configrpinterface`.

```
rtwdemo_configrpinterface
```



The model loads numeric MATLAB® variables into the model workspace. The workspace variables set some block parameters in the model. However, the Gain block in the model uses the literal value 2.

Disable Optimizations

1. In the model, clear the model configuration parameter **Signal storage reuse**. When you clear this optimization and other optimizations such as **Eliminate superfluous local variables (expression folding)**, the generated code allocates memory for signal lines. Clearing **Signal storage reuse** disables most of the other optimizations.

```
set_param('rtwdemo_configrpinterface','OptimizeBlockIOStorage','off')
```

2. Set the model configuration parameter **Default parameter behavior** to `Tunable`. When set to `Tunable`, this configuration parameter causes the generated code to allocate memory for block parameters and workspace variables.

```
set_param('rtwdemo_configrpinterface','DefaultParameterBehavior','Tunable')
```

3. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary
```

```
Top model targets built:

Model                       Action                        Rebuild Reason
==========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Code generation information file does no

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 37.816s
```

4. In the code generation report, view the file `rtwdemo_configrpinterface.h`. This header file defines a structure type that contains signal data. The structure contains fields that each represent a signal line in the model. For example, the output signal of the Gain block, whose name is `Gain`, appears as the field `Gain`.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.h');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T;',1,1)
```

```
/* Block signals (default storage) */
typedef struct {
  real_T Delay;                        /* '<Root>/Delay' */
  real_T Table2;                       /* '<Root>/Table2' */
  real_T Table1;                       /* '<Root>/Table1' */
  real_T Gain;                         /* '<Root>/Gain' */
  boolean_T RelOp1;                    /* '<Root>/RelOp1' */
  boolean_T RelOp2;                    /* '<Root>/RelOp2' */
  boolean_T LogOp;                     /* '<Root>/LogOp' */
  boolean_T DataStoreRead;             /* '<Root>/Data Store Read' */
} B_rtwdemo_configrpinterface_T;
```

The file defines a structure type that contains block parameter data. For example, the **Gain** parameter of the Gain block appears as the field `Gain_Gain`. The other fields of the structure represent other block parameters and workspace variables from the model, including initial conditions for signals and states.

```
rtwdemodbtype(file,'/* Parameters (default storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (default storage) */
struct P_rtwdemo_configrpinterface_T_ {
  Table1_Type Table1;                  /* Variable: Table1
                                        * Referenced by: '<Root>/Table1'
                                        */
  Table2_Type Table2;                  /* Variable: Table2
                                        * Referenced by: '<Root>/Table2'
                                        */
  real_T LOWER;                        /* Variable: LOWER
                                        * Referenced by: '<Root>/Constant2'
                                        */
  real_T UPPER;                        /* Variable: UPPER
                                        * Referenced by: '<Root>/Constant1'
                                        */
  real_T Gain_Gain;                    /* Expression: 2
                                        * Referenced by: '<Root>/Gain'
                                        */
```

```
  real_T Delay_InitialCondition;       /* Expression: 0
                                        * Referenced by: '<Root>/Delay'
                                        */
  uint32_T Table2_maxIndex[2];         /* Computed Parameter: Table2_maxIndex
                                        * Referenced by: '<Root>/Table2'
                                        */
  boolean_T DataStoreMemory_InitialValue;
                             /* Computed Parameter: DataStoreMemory_InitialValue
                              * Referenced by: '<Root>/Data Store Memory'
                              */
};
```

5. View the file `rtwdemo_configrpinterface_data.c`. This source file allocates global memory for a parameter structure and initializes the field values based on the parameter values in the model.

6. View the source file `rtwdemo_configrpinterface.c`. The code allocates global memory for a structure variable that contains signal data.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T rtwdemo_configrpinterface_B;',1,1)
```

```
/* Block signals (default storage) */
B_rtwdemo_configrpinterface_T rtwdemo_configrpinterface_B;
```

The code algorithm in the model `step` function calculates the signal values. It then assigns these values to the fields of the signal structure. To perform the calculations, the algorithm uses the parameter values from the fields of the parameter structure.

Exclude Data Items from Optimizations

When you want to select code generation optimizations such as **Signal storage reuse**, you can preserve individual data items from the optimizations. The generated code then allocates addressable memory for the items.

Select the optimizations that you previously cleared.

```
set_param('rtwdemo_configrpinterface','OptimizeBlockIOStorage','on')
set_param('rtwdemo_configrpinterface','LocalBlockOutputs','on')
set_param('rtwdemo_configrpinterface','DefaultParameterBehavior','Inlined')
```

1. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

2. In the Code Mappings editor, inspect the **Signals/States** tab.

3. In the model, select the output signal of the Gain block.

4. Pause on the ellipsis that appears above or below the signal line to open the action bar. Click the **Add Signal** button. In the Code Mappings editor, the **Signals** node expands and lists the signal that you added.

```
cm = coder.mapping.api.get('rtwdemo_configrpinterface');
gain_ports = get_param('rtwdemo_configrpinterface/Gain','PortHandles');
gain_outPort = gain_ports.Outport;
addSignal(cm,gain_outPort);
```

5. Apply the default storage class for Outports to the signal by setting **Storage Class** to `Model default`.

```
setSignal(cm,gain_outPort,'StorageClass','Model default','Identifier','Gain');
```

6. In the model, select the Gain block.

7. In the Property Inspector, set the gain value to the model workspace parameter object `K1`.

8. In the Code Mappings editor, on the **Parameters** tab, expand **Model Parameters**.

9. Select parameter `K1`. Apply a storage class other than `Auto` by setting **Storage class**. For example, use the storage class `Model default`. By using `Model default`, `K1` acquires the default storage class specified for model parameters, `Default`, and appears as a field of the global parameters structure.

```
set_param('rtwdemo_configrpinterface/Gain','Gain','K1');
setModelParameter(cm,'K1','StorageClass','Model default');
```

10. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                       Action                        Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.563s
```

11. In the code generation report, view the file `rtwdemo_configrpinterface.h`. The structure that contains signal data now defines only one field, `Gain`, which represents the test-pointed output of the Gain block.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.h');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T;',1,1)
```

```
/* Block signals (default storage) */
typedef struct {
  real_T Gain;                          /* '<Root>/Gain' */
} B_rtwdemo_configrpinterface_T;
```

The structure that contains block parameter data defines one field, `K1`, which represents the parameter object `K1`.

```
rtwdemodbtype(file,'/* Parameters (default storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (default storage) */
```

```
struct P_rtwdemo_configrpinterface_T_ {
  int8_T K1;                            /* Variable: K1
                                         * Referenced by: '<Root>/Gain'
                                         */
};
```

Access Data Through Generated Interfaces

You can configure the generated code to contain extra code and files so that you can access model data through standardized interfaces. For example, use the C API to log signal data and tune parameters during execution.

Copy this custom source code into a file named `ex_myHandCode.c` in your current folder.

```
#include "ex_myHandHdr.h"

#define paramIdx 0 /* Index of the target parameter,
determined by inspecting the array of structures generated by the C API. */
#define sigIdx 0 /* Index of the target signal,
determined by inspecting the array of structures generated by the C API. */

void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the parameter value only at
       the beginning of simulation and at the 5-second mark. */
    if (*tPtr == 0 || *tPtr == 5) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_ModelParameters *params;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        params = rtwCAPI_GetModelParameters(mmi);
        addrIdx = rtwCAPI_GetModelParameterAddrIdx(params,paramIdx);
        dTypeIdx =  rtwCAPI_GetModelParameterDataTypeIdx(params,paramIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'int8'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                /* At the 5-second mark, increment the parameter value by 1. */
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %f\n", *dataAddress);
                break;
            }
```

```
            case SS_INT8: {
                int8_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %i\n", *dataAddress);
                break;
            }
        }
    }
}

void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the signal value only when
       the simulation time is an integer value. */
    if (*tPtr-(int_T)*tPtr == 0) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_Signals *sigs;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        sigs = rtwCAPI_GetSignals(mmi);
        addrIdx = rtwCAPI_GetSignalAddrIdx(sigs,sigIdx);
        dTypeIdx =  rtwCAPI_GetSignalDataTypeIdx(sigs,sigIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'single'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }

            case SS_SINGLE: {
                real32_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }
        }
    }
}
```

2. Copy this custom header code into a file named `ex_myHandHdr.h` in your current folder.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
/* Include rtw_modelmap.h for definitions of C API macros. */
#include "rtw_modelmap.h"
#include "builtin_typeid_types.h"
#include "rtwtypes.h"
void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
```

These files use the C API to access signal and parameter data in the code that you generate from the example model.

3. Set model configuration parameters **Include headers** and **Source files** to `#include "ex_myHandHdr.h"` and `ex_myHandCode.c`, respectively.

```
set_param('rtwdemo_configrpinterface','CustomHeaderCode','#include "ex_myHandHdr.h"')
set_param('rtwdemo_configrpinterface','CustomSource','ex_myHandCode.c')
```

4. Select model configuration parameter **MAT-file Logging**. The generated executable runs only until the simulation stop time (which you set in the model configuration parameters).

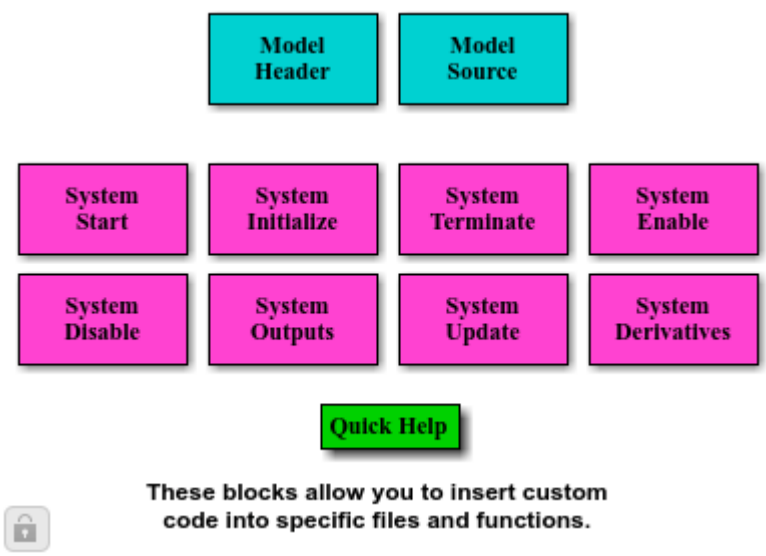
```
set_param('rtwdemo_configrpinterface','MatFileLogging','on')
```

5. Select C API model configuration parameters **parameters**, **signals**, **states**, and **root-level I/O**.

```
set_param('rtwdemo_configrpinterface','RTWCAPIParams','on')
set_param('rtwdemo_configrpinterface','RTWCAPISignals','on')
set_param('rtwdemo_configrpinterface','RTWCAPIStates','on')
set_param('rtwdemo_configrpinterface','RTWCAPIRootIO','on')
```

6. Load the Custom Code block library.

```
custcode
```



7. Add a System Outputs block to the model.

```
add_block('custcode/System Outputs','rtwdemo_configrpinterface/System Outputs')
```

8. In the System Outputs block dialog box, set **System Outputs Function Execution Code** to this code:

```
{
rtwdemo_configrpinterface_U.input2++;
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);
tuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}
```

9. In the block dialog box, set **System Outputs Function Exit Code** to this code:

```
{
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);
logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}
```

Alternatively, to configure the System Outputs block, at the command prompt, use these commands:

```
temp.TLCFile = 'custcode';
temp.Location = 'System Outputs Function';
temp.Middle = sprintf(['{\nrtwdemo_configrpinterface_U.input2++;'...
    '\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);'...
    '\ntuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));\n}']);
temp.Bottom = sprintf(['{\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);'...
    '\nlogFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));\n}']);
set_param('rtwdemo_configrpinterface/System Outputs','RTWdata',temp)
```

10. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                      Action                        Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.705s
```

11. In the code generation report, view the interface file `rtwdemo_configrpinterface_capi.c`. This file initializes the arrays of structures that you can use to interact with data items through the C API. For example, in the array of structures `rtBlockSignals`, the first structure (index 0) describes the test-pointed output signal of the Gain block in the model.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface_capi.c');
rtwdemodbtype(file,'/* Block output signal information */',...
    '/* Individual block tuning',1,0)
```

```
/* Block output signal information */
```

```
static const rtwCAPI_Signals rtBlockSignals[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 0, 0, TARGET_STRING("rtwdemo_configrpinterface/Gain"),
    TARGET_STRING(""), 0, 0, 0, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

The fields of the structure, such as `addrMapIndex`, indicate indices into other arrays of structures, such as `rtDataAddrMap`, that describe the characteristics of the signal. These characteristics include the address of the signal data (a pointer to the data), the numeric data type, and the dimensions of the signal.

12. In the file `rtwdemo_configrpinterface.c`, view the code algorithm in the model `step` function. The algorithm first executes the code that you specified in the System Outputs block.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* user code (Output function Body) */',...
    '/* Logic: ''<Root>/LogOp'' incorporates:',1,0)
```

```
  /* user code (Output function Body) */

  /* System '<Root>' */
  {
    rtwdemo_configrpinterface_U.input2++;
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
      (rtwdemo_configrpinterface_M).mmi);
    tuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
  }

  /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Constant: '<Root>/Constant2'
   *  Inport: '<Root>/In1'
   *  Logic: '<Root>/LogOp'
   *  RelationalOperator: '<Root>/RelOp1'
   *  RelationalOperator: '<Root>/RelOp2'
   */
  mode = ((rtwdemo_configrpinterface_U.In1 > 10.0) ||
          (rtwdemo_configrpinterface_U.In1 < -10.0));

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In2'
   *  Lookup_n-D: '<Root>/Table1'
   */
  rtwdemo_configrpinterface_B.Gain = (real_T)rtwdemo_configrpinterface_P.K1 *
    look1_binlc(rtwdemo_configrpinterface_U.input2, rtCP_Table1_bp01Data,
                rtCP_Table1_tableData, 10U);

  /* Switch: '<Root>/Switch' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   */
  if (mode) {
```

```
  /* Outport: '<Root>/Out1' */
  output = rtwdemo_configrpinterface_B.Gain;
} else {
  /* Outport: '<Root>/Out1' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  output = rtwdemo_configrpinterface_DW.X;
}

/* End of Switch: '<Root>/Switch' */

/* Lookup_n-D: '<Root>/Table2' incorporates:
 *  Inport: '<Root>/In3'
 *  Inport: '<Root>/In4'
 */
rtwdemo_configrpinterface_DW.X = look2_binlc(rtwdemo_configrpinterface_U.In3,
  rtwdemo_configrpinterface_U.In4, rtCP_Table2_bp01Data, rtCP_Table2_bp02Data,
  rtCP_Table2_tableData, rtCP_Table2_maxIndex, 3U);

/* user code (Output function Trailer) */

/* System '<Root>' */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
    (rtwdemo_configrpinterface_M).mmi);
  logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}

/* Matfile logging */
rt_UpdateTXYLogVars(rtwdemo_configrpinterface_M->rtwLogInfo,
                    (&rtwdemo_configrpinterface_M->Timing.taskTime0));

/* signal main to stop simulation */
{                                   /* Sample time: [1.0s, 0.0s] */
  if ((rtmGetTFinal(rtwdemo_configrpinterface_M)!=-1) &&
      !((rtmGetTFinal(rtwdemo_configrpinterface_M)-
         rtwdemo_configrpinterface_M->Timing.taskTime0) >
        rtwdemo_configrpinterface_M->Timing.taskTime0 * (DBL_EPSILON))) {
    rtmSetErrorStatus(rtwdemo_configrpinterface_M, "Simulation finished");
  }
}

/* Update absolute time for base rate */
/* The "clockTick0" counts the number of times the code of this task has
 * been executed. The absolute time is the multiplication of "clockTick0"
 * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
 * overflow during the application lifespan selected.
 * Timer of this task consists of two 32 bit unsigned integers.
 * The two integers represent the low bits Timing.clockTick0 and the high bits
 * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
 */
if (!(++rtwdemo_configrpinterface_M->Timing.clockTick0)) {
  ++rtwdemo_configrpinterface_M->Timing.clockTickH0;
}

rtwdemo_configrpinterface_M->Timing.taskTime0 =
  rtwdemo_configrpinterface_M->Timing.clockTick0 *
  rtwdemo_configrpinterface_M->Timing.stepSize0 +
```

```
    rtwdemo_configrpinterface_M->Timing.clockTickH0 *
    rtwdemo_configrpinterface_M->Timing.stepSize0 * 4294967296.0;
}

/* Model initialize function */
void rtwdemo_configrpinterface_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)rtwdemo_configrpinterface_M, 0,
                sizeof(RT_MODEL_rtwdemo_configrpinte_T));
  rtmSetTFinal(rtwdemo_configrpinterface_M, 10.0);
  rtwdemo_configrpinterface_M->Timing.stepSize0 = 1.0;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    rtwdemo_configrpinterface_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogT(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogX(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogXFinal(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(rtwdemo_configrpinterface_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(rtwdemo_configrpinterface_M->rtwLogInfo, 0);
    rtliSetLogMaxRows(rtwdemo_configrpinterface_M->rtwLogInfo, 1000);
    rtliSetLogDecimation(rtwdemo_configrpinterface_M->rtwLogInfo, 1);
    rtliSetLogY(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogYSignalInfo(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogYSignalPtrs(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
  }

  /* block I/O */
  (void) memset(((void *) &rtwdemo_configrpinterface_B), 0,
                sizeof(B_rtwdemo_configrpinterface_T));

  /* Storage classes */
  output = 0.0;

  /* states (dwork) */
  (void) memset((void *)&rtwdemo_configrpinterface_DW, 0,
                sizeof(DW_rtwdemo_configrpinterface_T));

  /* exported global states */
  mode = false;

  /* external inputs */
  (void)memset(&rtwdemo_configrpinterface_U, 0, sizeof
               (ExtU_rtwdemo_configrpinterfac_T));
```

```
  /* Initialize DataMapInfo substructure containing ModelMap for C API */
  rtwdemo_configrpinterface_InitializeDataMapInfo();

  /* Matfile logging */
  rt_StartDataLoggingWithStartTime(rtwdemo_configrpinterface_M->rtwLogInfo, 0.0,
    rtmGetTFinal(rtwdemo_configrpinterface_M),
    rtwdemo_configrpinterface_M->Timing.stepSize0, (&rtmGetErrorStatus
    (rtwdemo_configrpinterface_M)));

  /* Start for DataStoreMemory: '<Root>/Data Store Memory' */
  mode = false;

  /* InitializeConditions for Lookup_n-D: '<Root>/Table2' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  rtwdemo_configrpinterface_DW.X = 0.0;
}

/* Model terminate function */
void rtwdemo_configrpinterface_terminate(void)
{
  /* (no terminate code required) */
}
```

This code first perturbs the input signal `input2` by incrementing the value of the signal each time the `step` function executes. The code then uses the built-in macro `rtmGetDataMapInfo` to extract model mapping information from the model data structure `rtwdemo_configrpinterface_M`. The pointer `MMI` points to the extracted mapping information, which allows the functions `tuneFcn` and `logFcn` to access the information contained in the arrays of structures that the C API file `rtwdemo_configrpinterface_capi.c` defines.

13. View the function `tuneFcn` in the file `ex_myHandCode.c`. This function uses the C API (through the model mapping information `mmi`) and a pointer to the simulation time to print the value of the parameter `K1` at specific times during code execution. When the simulation time reaches 5 seconds, the function changes the parameter value in memory. By using a `switch case` block, the function can access the parameter data whether the data type is `int8` or `double`.

14. View the code algorithm in the model `step` function again. Near the end of the function, the algorithm executes the code that you specified in the System Outputs block. This code calls the function `logFcn`.

```
rtwdemodbtype(file,'/* user code (Output function Trailer) */',...
    '/* Matfile logging */',1,0)
```

```
  /* user code (Output function Trailer) */

  /* System '<Root>' */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
      (rtwdemo_configrpinterface_M).mmi);
    logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
  }
```

15. View the function `logFcn` in the file `ex_myHandCode.c`. The function uses the C API to print the value of the test-pointed signal. The function can access the signal data whether the data type is `single` or `double`.

16. At the command prompt, run the generated executable `rtwdemo_configrpinterface.exe`.

```
system('rtwdemo_configrpinterface')
```

The parameter and signal values appear in the Command Window output.

For more information about data interfaces, including the C API, see “Generated Code Interfacing”.

Configure Data Accessibility

Goal	Considerations and More Information
Configure signals as accessible and parameters as tunable by default	Clear the model configuration parameter **Signal storage reuse** and set the configuration parameter **Default parameter behavior** to `Tunable`. These settings prevent the elimination of storage for signals and prevent parameter inlining. Each block parameter and signal line appears in the generated code as a field of a structure. For more information about these data structures, see “How Generated Code Exchanges Data with an Environment” on page 36-35 and “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49. For more information about **Default parameter behavior**, see Default parameter behavior. For more information about optimizations that eliminate storage for signals, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49 and “Minimize Computations and Storage for Intermediate Results at Block Outputs”.
Preserve accessibility and tunability of a data item after you select optimizations	You can generate more efficient code by selecting optimizations such as **Signal storage reuse**, but the code generator eliminates storage for as many data items as possible. To exclude individual data items from the optimizations: • Configure signals in the Code Mappings editor. In the editor you can set the default configuration for signals, add signals of interest to the code mappings, and then set the storage class for individual signals to the default storage class or another storage class. Signals can be named by setting the `Identifier` storage class property. • Apply a storage class, such as `ExportedGlobal`, to states and parameters. See “C Code Generation Configuration for Model Interface Elements” on page 33-2.
Represent a data item as a separate global variable in the generated code	When you disable optimizations, signal lines, block states, and parameters appear in the generated code as fields of structures. You cannot control the names of the structures without Embedded Coder. To instead store a data item in a separate global variable whose name, file placement, and other characteristics you can control, apply a storage class to a signal, state, or `Simulink.Parameter` object. See “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Goal	Considerations and More Information
Generate a standardized C-code interface for accessing data	You can configure the generated code to contain extra code and files so that you can access model data through standardized interfaces. For more information, see “Exchange Data Between Generated and External Code Using C API” on page 62-2.
Tune parameters and monitor signals during external mode simulation	When you generate code and an external executable from a model, you can simulate the model in external mode to communicate with the running executable. You can tune parameters and monitor signals during the simulation. However, in this simulation mode, parameter tunability limitations that apply to code generation also apply to the simulation. For information about the code generation limitations, see “Limitations for Block Parameter Tunability in Generated Code” on page 36-122. For information about external mode, see “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2.
Tune parameters and monitor signals with Simulink Real-Time	If you have Simulink Real-Time, you can tune parameters and monitor signals during execution of your real-time application. Make signals accessible and parameters tunable by clearing optimizations and applying test points and storage classes. See “Parameter Tuning with Simulink Real-Time Explorer” (Simulink Real-Time).

Limitations

For information about limitations that apply to the tunability of parameters in the generated code, see “Limitations for Block Parameter Tunability in Generated Code” on page 36-122.

See Also

Related Examples

- “Deploy Algorithm Model for Real-Time Rapid Prototyping” on page 70-2
- “Access Signal, State, and Parameter Data During Execution” on page 36-17
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 36-161

Access Signal, State, and Parameter Data During Execution

As you iteratively develop a model, you capture output signal and state data that model execution generates. You also tune parameter values during execution to observe results on the outputs. You can then base your design decisions upon analysis of these outputs. To access this signal, state, and parameter data in a rapid prototyping environment, you can configure the generated code to store the data in addressable memory.

By default, optimization settings make the generated code more efficient by eliminating unnecessary signal storage and inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for this data, you can disable the optimizations or specify code generation settings for individual data items.

Explore Example Model

Run a script that prepares the model rtwdemo_configrpinterface for this example.

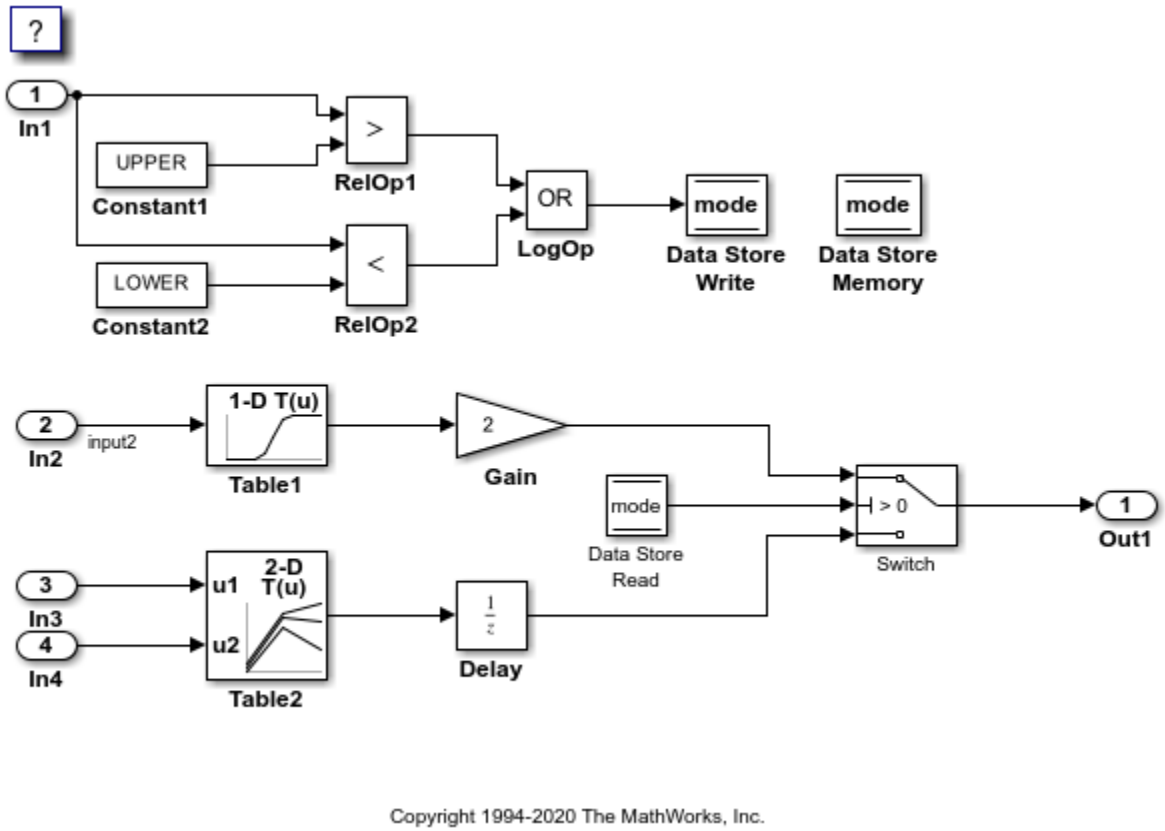
```
prepare_rtwdemo_configrpinterface
```

This data is configured for code generation:

- Parameters `UPPER`, `LOWER`, `K1`, `Table1`, and `Table2`
- Signals `In1`, `In2`, `In3`, `In4`, and `Out1`
- States `X` (delay) and `mode` (data store memory write)

Open the example model, `rtwdemo_configrpinterface`.

```
rtwdemo_configrpinterface
```



The model loads numeric MATLAB® variables into the model workspace. The workspace variables set some block parameters in the model. However, the Gain block in the model uses the literal value 2.

Disable Optimizations

1. In the model, clear the model configuration parameter **Signal storage reuse**. When you clear this optimization and other optimizations such as **Eliminate superfluous local variables (expression folding)**, the generated code allocates memory for signal lines. Clearing **Signal storage reuse** disables most of the other optimizations.

```
set_param('rtwdemo_configrpinterface','OptimizeBlockIOStorage','off')
```

2. Set the model configuration parameter **Default parameter behavior** to `Tunable`. When set to `Tunable`, this configuration parameter causes the generated code to allocate memory for block parameters and workspace variables.

```
set_param('rtwdemo_configrpinterface','DefaultParameterBehavior','Tunable')
```

3. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary
```

```
Top model targets built:

Model                       Action                        Rebuild Reason
==========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Code generation information file does no

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 37.816s
```

4. In the code generation report, view the file `rtwdemo_configrpinterface.h`. This header file defines a structure type that contains signal data. The structure contains fields that each represent a signal line in the model. For example, the output signal of the Gain block, whose name is `Gain`, appears as the field `Gain`.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.h');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T;',1,1)
```

```
/* Block signals (default storage) */
typedef struct {
  real_T Delay;                        /* '<Root>/Delay' */
  real_T Table2;                       /* '<Root>/Table2' */
  real_T Table1;                       /* '<Root>/Table1' */
  real_T Gain;                         /* '<Root>/Gain' */
  boolean_T RelOp1;                    /* '<Root>/RelOp1' */
  boolean_T RelOp2;                    /* '<Root>/RelOp2' */
  boolean_T LogOp;                     /* '<Root>/LogOp' */
  boolean_T DataStoreRead;             /* '<Root>/Data Store Read' */
} B_rtwdemo_configrpinterface_T;
```

The file defines a structure type that contains block parameter data. For example, the **Gain** parameter of the Gain block appears as the field `Gain_Gain`. The other fields of the structure represent other block parameters and workspace variables from the model, including initial conditions for signals and states.

```
rtwdemodbtype(file,'/* Parameters (default storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (default storage) */
struct P_rtwdemo_configrpinterface_T_ {
  Table1_Type Table1;                  /* Variable: Table1
                                        * Referenced by: '<Root>/Table1'
                                        */
  Table2_Type Table2;                  /* Variable: Table2
                                        * Referenced by: '<Root>/Table2'
                                        */
  real_T LOWER;                        /* Variable: LOWER
                                        * Referenced by: '<Root>/Constant2'
                                        */
  real_T UPPER;                        /* Variable: UPPER
                                        * Referenced by: '<Root>/Constant1'
                                        */
  real_T Gain_Gain;                    /* Expression: 2
                                        * Referenced by: '<Root>/Gain'
                                        */
```

```
  real_T Delay_InitialCondition;       /* Expression: 0
                                        * Referenced by: '<Root>/Delay'
                                        */
  uint32_T Table2_maxIndex[2];         /* Computed Parameter: Table2_maxIndex
                                        * Referenced by: '<Root>/Table2'
                                        */
  boolean_T DataStoreMemory_InitialValue;
                             /* Computed Parameter: DataStoreMemory_InitialValue
                              * Referenced by: '<Root>/Data Store Memory'
                              */
};
```

5. View the file `rtwdemo_configrpinterface_data.c`. This source file allocates global memory for a parameter structure and initializes the field values based on the parameter values in the model.

6. View the source file `rtwdemo_configrpinterface.c`. The code allocates global memory for a structure variable that contains signal data.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T rtwdemo_configrpinterface_B;',1,1)
```

```
/* Block signals (default storage) */
B_rtwdemo_configrpinterface_T rtwdemo_configrpinterface_B;
```

The code algorithm in the model `step` function calculates the signal values. It then assigns these values to the fields of the signal structure. To perform the calculations, the algorithm uses the parameter values from the fields of the parameter structure.

Exclude Data Items from Optimizations

When you want to select code generation optimizations such as **Signal storage reuse**, you can preserve individual data items from the optimizations. The generated code then allocates addressable memory for the items.

Select the optimizations that you previously cleared.

```
set_param('rtwdemo_configrpinterface','OptimizeBlockIOStorage','on')
set_param('rtwdemo_configrpinterface','LocalBlockOutputs','on')
set_param('rtwdemo_configrpinterface','DefaultParameterBehavior','Inlined')
```

1. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

2. In the Code Mappings editor, inspect the **Signals/States** tab.

3. In the model, select the output signal of the Gain block.

4. Pause on the ellipsis that appears above or below the signal line to open the action bar. Click the **Add Signal** button. In the Code Mappings editor, the **Signals** node expands and lists the signal that you added.

```
cm = coder.mapping.api.get('rtwdemo_configrpinterface');
gain_ports = get_param('rtwdemo_configrpinterface/Gain','PortHandles');
gain_outPort = gain_ports.Outport;
addSignal(cm,gain_outPort);
```

5. Apply the default storage class for Outports to the signal by setting **Storage Class** to `Model default`.

```
setSignal(cm,gain_outPort,'StorageClass','Model default','Identifier','Gain');
```

6. In the model, select the Gain block.

7. In the Property Inspector, set the gain value to the model workspace parameter object `K1`.

8. In the Code Mappings editor, on the **Parameters** tab, expand **Model Parameters**.

9. Select parameter `K1`. Apply a storage class other than `Auto` by setting **Storage class**. For example, use the storage class `Model default`. By using `Model default`, `K1` acquires the default storage class specified for model parameters, `Default`, and appears as a field of the global parameters structure.

```
set_param('rtwdemo_configrpinterface/Gain','Gain','K1');
setModelParameter(cm,'K1','StorageClass','Model default');
```

10. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                       Action                        Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.563s
```

11. In the code generation report, view the file `rtwdemo_configrpinterface.h`. The structure that contains signal data now defines only one field, `Gain`, which represents the test-pointed output of the Gain block.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.h');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    'B_rtwdemo_configrpinterface_T;',1,1)
```

```
/* Block signals (default storage) */
typedef struct {
  real_T Gain;                          /* '<Root>/Gain' */
} B_rtwdemo_configrpinterface_T;
```

The structure that contains block parameter data defines one field, `K1`, which represents the parameter object `K1`.

```
rtwdemodbtype(file,'/* Parameters (default storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (default storage) */
```

```
struct P_rtwdemo_configrpinterface_T_ {
  int8_T K1;                            /* Variable: K1
                                         * Referenced by: '<Root>/Gain'
                                         */
};
```

Access Data Through Generated Interfaces

You can configure the generated code to contain extra code and files so that you can access model data through standardized interfaces. For example, use the C API to log signal data and tune parameters during execution.

Copy this custom source code into a file named `ex_myHandCode.c` in your current folder.

```
#include "ex_myHandHdr.h"

#define paramIdx 0 /* Index of the target parameter,
determined by inspecting the array of structures generated by the C API. */
#define sigIdx 0 /* Index of the target signal,
determined by inspecting the array of structures generated by the C API. */

void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the parameter value only at
       the beginning of simulation and at the 5-second mark. */
    if (*tPtr == 0 || *tPtr == 5) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_ModelParameters *params;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        params = rtwCAPI_GetModelParameters(mmi);
        addrIdx = rtwCAPI_GetModelParameterAddrIdx(params,paramIdx);
        dTypeIdx =  rtwCAPI_GetModelParameterDataTypeIdx(params,paramIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'int8'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                /* At the 5-second mark, increment the parameter value by 1. */
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %f\n", *dataAddress);
                break;
            }
```

```
            case SS_INT8: {
                int8_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %i\n", *dataAddress);
                break;
            }
        }
    }
}

void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the signal value only when
       the simulation time is an integer value. */
    if (*tPtr-(int_T)*tPtr == 0) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_Signals *sigs;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        sigs = rtwCAPI_GetSignals(mmi);
        addrIdx = rtwCAPI_GetSignalAddrIdx(sigs,sigIdx);
        dTypeIdx =  rtwCAPI_GetSignalDataTypeIdx(sigs,sigIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'single'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }

            case SS_SINGLE: {
                real32_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }
        }
    }
}
```

2. Copy this custom header code into a file named `ex_myHandHdr.h` in your current folder.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
/* Include rtw_modelmap.h for definitions of C API macros. */
#include "rtw_modelmap.h"
#include "builtin_typeid_types.h"
#include "rtwtypes.h"
void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
```

These files use the C API to access signal and parameter data in the code that you generate from the example model.

3. Set model configuration parameters **Include headers** and **Source files** to `#include "ex_myHandHdr.h"` and `ex_myHandCode.c`, respectively.

```
set_param('rtwdemo_configrpinterface','CustomHeaderCode','#include "ex_myHandHdr.h"')
set_param('rtwdemo_configrpinterface','CustomSource','ex_myHandCode.c')
```

4. Select model configuration parameter **MAT-file Logging**. The generated executable runs only until the simulation stop time (which you set in the model configuration parameters).

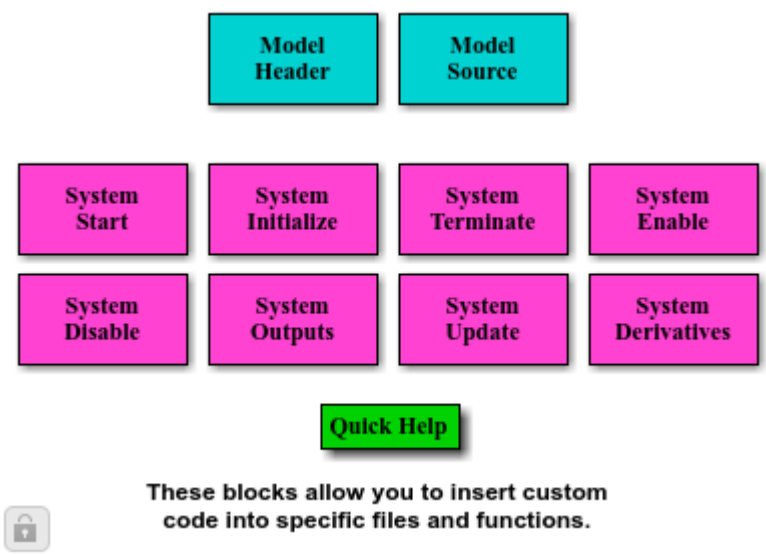
```
set_param('rtwdemo_configrpinterface','MatFileLogging','on')
```

5. Select C API model configuration parameters **parameters**, **signals**, **states**, and **root-level I/O**.

```
set_param('rtwdemo_configrpinterface','RTWCAPIParams','on')
set_param('rtwdemo_configrpinterface','RTWCAPISignals','on')
set_param('rtwdemo_configrpinterface','RTWCAPIStates','on')
set_param('rtwdemo_configrpinterface','RTWCAPIRootIO','on')
```

6. Load the Custom Code block library.

```
custcode
```



7. Add a System Outputs block to the model.

```
add_block('custcode/System Outputs','rtwdemo_configrpinterface/System Outputs')
```

8. In the System Outputs block dialog box, set **System Outputs Function Execution Code** to this code:

```
{
rtwdemo_configrpinterface_U.input2++;
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);
tuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}
```

9. In the block dialog box, set **System Outputs Function Exit Code** to this code:

```
{
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);
logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}
```

Alternatively, to configure the System Outputs block, at the command prompt, use these commands:

```
temp.TLCFile = 'custcode';
temp.Location = 'System Outputs Function';
temp.Middle = sprintf(['{\nrtwdemo_configrpinterface_U.input2++;'...
    '\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);'...
    '\ntuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));\n}']);
temp.Bottom = sprintf(['{\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_configrpinterface_M).mmi);'...
    '\nlogFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));\n}']);
set_param('rtwdemo_configrpinterface/System Outputs','RTWdata',temp)
```

10. Generate code.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of build procedure for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                      Action                        Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.705s
```

11. In the code generation report, view the interface file `rtwdemo_configrpinterface_capi.c`. This file initializes the arrays of structures that you can use to interact with data items through the C API. For example, in the array of structures `rtBlockSignals`, the first structure (index 0) describes the test-pointed output signal of the Gain block in the model.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface_capi.c');
rtwdemodbtype(file,'/* Block output signal information */',...
    '/* Individual block tuning',1,0)
```

```
/* Block output signal information */
```

```
static const rtwCAPI_Signals rtBlockSignals[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 0, 0, TARGET_STRING("rtwdemo_configrpinterface/Gain"),
    TARGET_STRING(""), 0, 0, 0, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

The fields of the structure, such as `addrMapIndex`, indicate indices into other arrays of structures, such as `rtDataAddrMap`, that describe the characteristics of the signal. These characteristics include the address of the signal data (a pointer to the data), the numeric data type, and the dimensions of the signal.

12. In the file `rtwdemo_configrpinterface.c`, view the code algorithm in the model `step` function. The algorithm first executes the code that you specified in the System Outputs block.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* user code (Output function Body) */',...
    '/* Logic: ''<Root>/LogOp'' incorporates:',1,0)
```

```
  /* user code (Output function Body) */

  /* System '<Root>' */
  {
    rtwdemo_configrpinterface_U.input2++;
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
      (rtwdemo_configrpinterface_M).mmi);
    tuneFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
  }

  /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Constant: '<Root>/Constant2'
   *  Inport: '<Root>/In1'
   *  Logic: '<Root>/LogOp'
   *  RelationalOperator: '<Root>/RelOp1'
   *  RelationalOperator: '<Root>/RelOp2'
   */
  mode = ((rtwdemo_configrpinterface_U.In1 > 10.0) ||
          (rtwdemo_configrpinterface_U.In1 < -10.0));

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In2'
   *  Lookup_n-D: '<Root>/Table1'
   */
  rtwdemo_configrpinterface_B.Gain = (real_T)rtwdemo_configrpinterface_P.K1 *
    look1_binlc(rtwdemo_configrpinterface_U.input2, rtCP_Table1_bp01Data,
                rtCP_Table1_tableData, 10U);

  /* Switch: '<Root>/Switch' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   */
  if (mode) {
```

```
  /* Outport: '<Root>/Out1' */
  output = rtwdemo_configrpinterface_B.Gain;
} else {
  /* Outport: '<Root>/Out1' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  output = rtwdemo_configrpinterface_DW.X;
}

/* End of Switch: '<Root>/Switch' */

/* Lookup_n-D: '<Root>/Table2' incorporates:
 *  Inport: '<Root>/In3'
 *  Inport: '<Root>/In4'
 */
rtwdemo_configrpinterface_DW.X = look2_binlc(rtwdemo_configrpinterface_U.In3,
  rtwdemo_configrpinterface_U.In4, rtCP_Table2_bp01Data, rtCP_Table2_bp02Data,
  rtCP_Table2_tableData, rtCP_Table2_maxIndex, 3U);

/* user code (Output function Trailer) */

/* System '<Root>' */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
    (rtwdemo_configrpinterface_M).mmi);
  logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
}

/* Matfile logging */
rt_UpdateTXYLogVars(rtwdemo_configrpinterface_M->rtwLogInfo,
                    (&rtwdemo_configrpinterface_M->Timing.taskTime0));

/* signal main to stop simulation */
{                                   /* Sample time: [1.0s, 0.0s] */
  if ((rtmGetTFinal(rtwdemo_configrpinterface_M)!=-1) &&
      !((rtmGetTFinal(rtwdemo_configrpinterface_M)-
         rtwdemo_configrpinterface_M->Timing.taskTime0) >
        rtwdemo_configrpinterface_M->Timing.taskTime0 * (DBL_EPSILON))) {
    rtmSetErrorStatus(rtwdemo_configrpinterface_M, "Simulation finished");
  }
}

/* Update absolute time for base rate */
/* The "clockTick0" counts the number of times the code of this task has
 * been executed. The absolute time is the multiplication of "clockTick0"
 * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
 * overflow during the application lifespan selected.
 * Timer of this task consists of two 32 bit unsigned integers.
 * The two integers represent the low bits Timing.clockTick0 and the high bits
 * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
 */
if (!(++rtwdemo_configrpinterface_M->Timing.clockTick0)) {
  ++rtwdemo_configrpinterface_M->Timing.clockTickH0;
}

rtwdemo_configrpinterface_M->Timing.taskTime0 =
  rtwdemo_configrpinterface_M->Timing.clockTick0 *
  rtwdemo_configrpinterface_M->Timing.stepSize0 +
```

```
    rtwdemo_configrpinterface_M->Timing.clockTickH0 *
    rtwdemo_configrpinterface_M->Timing.stepSize0 * 4294967296.0;
}

/* Model initialize function */
void rtwdemo_configrpinterface_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)rtwdemo_configrpinterface_M, 0,
                sizeof(RT_MODEL_rtwdemo_configrpinte_T));
  rtmSetTFinal(rtwdemo_configrpinterface_M, 10.0);
  rtwdemo_configrpinterface_M->Timing.stepSize0 = 1.0;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    rtwdemo_configrpinterface_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogT(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogX(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogXFinal(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(rtwdemo_configrpinterface_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(rtwdemo_configrpinterface_M->rtwLogInfo, 0);
    rtliSetLogMaxRows(rtwdemo_configrpinterface_M->rtwLogInfo, 1000);
    rtliSetLogDecimation(rtwdemo_configrpinterface_M->rtwLogInfo, 1);
    rtliSetLogY(rtwdemo_configrpinterface_M->rtwLogInfo, "");
    rtliSetLogYSignalInfo(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
    rtliSetLogYSignalPtrs(rtwdemo_configrpinterface_M->rtwLogInfo, (NULL));
  }

  /* block I/O */
  (void) memset(((void *) &rtwdemo_configrpinterface_B), 0,
                sizeof(B_rtwdemo_configrpinterface_T));

  /* Storage classes */
  output = 0.0;

  /* states (dwork) */
  (void) memset((void *)&rtwdemo_configrpinterface_DW, 0,
                sizeof(DW_rtwdemo_configrpinterface_T));

  /* exported global states */
  mode = false;

  /* external inputs */
  (void)memset(&rtwdemo_configrpinterface_U, 0, sizeof
               (ExtU_rtwdemo_configrpinterfac_T));
```

```
  /* Initialize DataMapInfo substructure containing ModelMap for C API */
  rtwdemo_configrpinterface_InitializeDataMapInfo();

  /* Matfile logging */
  rt_StartDataLoggingWithStartTime(rtwdemo_configrpinterface_M->rtwLogInfo, 0.0,
    rtmGetTFinal(rtwdemo_configrpinterface_M),
    rtwdemo_configrpinterface_M->Timing.stepSize0, (&rtmGetErrorStatus
    (rtwdemo_configrpinterface_M)));

  /* Start for DataStoreMemory: '<Root>/Data Store Memory' */
  mode = false;

  /* InitializeConditions for Lookup_n-D: '<Root>/Table2' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  rtwdemo_configrpinterface_DW.X = 0.0;
}

/* Model terminate function */
void rtwdemo_configrpinterface_terminate(void)
{
  /* (no terminate code required) */
}
```

This code first perturbs the input signal `input2` by incrementing the value of the signal each time the `step` function executes. The code then uses the built-in macro `rtmGetDataMapInfo` to extract model mapping information from the model data structure `rtwdemo_configrpinterface_M`. The pointer `MMI` points to the extracted mapping information, which allows the functions `tuneFcn` and `logFcn` to access the information contained in the arrays of structures that the C API file `rtwdemo_configrpinterface_capi.c` defines.

13. View the function `tuneFcn` in the file `ex_myHandCode.c`. This function uses the C API (through the model mapping information `mmi`) and a pointer to the simulation time to print the value of the parameter `K1` at specific times during code execution. When the simulation time reaches 5 seconds, the function changes the parameter value in memory. By using a `switch case` block, the function can access the parameter data whether the data type is `int8` or `double`.

14. View the code algorithm in the model `step` function again. Near the end of the function, the algorithm executes the code that you specified in the System Outputs block. This code calls the function `logFcn`.

```
rtwdemodbtype(file,'/* user code (Output function Trailer) */',...
    '/* Matfile logging */',1,0)
```

```
  /* user code (Output function Trailer) */

  /* System '<Root>' */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo
      (rtwdemo_configrpinterface_M).mmi);
    logFcn(MMI, rtmGetTPtr(rtwdemo_configrpinterface_M));
  }
```

15. View the function `logFcn` in the file `ex_myHandCode.c`. The function uses the C API to print the value of the test-pointed signal. The function can access the signal data whether the data type is `single` or `double`.

16. At the command prompt, run the generated executable `rtwdemo_configrpinterface.exe`.

```
system('rtwdemo_configrpinterface')
```

The parameter and signal values appear in the Command Window output.

For more information about data interfaces, including the C API, see “Generated Code Interfacing”.

See Also

`Simulink.Parameter` | `Simulink.Signal` | `Simulink.Signal` | System Outputs

Related Examples

- “Data Structures in the Generated Code” on page 36-31
- “Preserve Variables in Generated Code” on page 36-2
- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Exchange Data Between Generated and External Code Using C API” on page 62-2

Data Structures in the Generated Code

The code generator organizes the data used by the model in data structures. For each model, the code generator stores data, such as signals and states, in global structures that are also referred to as standard data structures. Use these data structures to access the data in the generated code.

Real-time model (`rtModel`) data structure is a top level data structure that contains instance-specific data, which is used by the model for reusable functions. The code generator uses this data structure for only multi-instance code generation.

Standard Data Structures

In a model, root-level Inport and Outport blocks represent the primary inputs and outputs of the block algorithm. By default, the code generator aggregates these blocks into standard structures that store input and output data.

Signal lines, block parameters, states, and other model data appear in the generated code as fields of standard structures. For general information, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49.

The table shows some common data structures in the generated code. The name of the structure type and, if applicable, structure variable in the code that you generate from a model depends on the model settings. Regardless of the settings, the name of the structure type in the code contains the short name from the **Short Name of Structure Type** column in the table.

Data Structures Generated for a Model

Short Name of Structure Type	Data Represented in the Structure
`ExtU`	Data from root Inport blocks
`ExtY`	Data from root Outport blocks
`B`	Block output signals
`ConstB`	Block outputs that have constant values
`P`	Block parameters
`ConstP`	Constant parameters
`DW`	Discrete block states
`XDis`	Status of enabled subsystems
`InstP`	Parameter arguments
`SharedDSM`	Shared local data stores

You can exclude data from appearing in these structures by using:

- Direct application of storage classes. For example, you can use storage classes to represent signals, tunable parameters, and states as individual global variables. For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.
- Configuration parameters, such as those on the **Optimization** pane in the Configuration Parameters dialog box. You can adjust these configuration parameters to control the default representation of data. For more information, see “Model Configuration Parameters: Code Generation Optimization”.

Real-Time Model Data Structure

To access the fields of an S-function's simulation data structure (`SimStruct`), the Simulink software provides a set of functions. The code generator replaces `SimStruct` with an optimized real-time model (`rtModel`) data structure.

Code generation by using the GRT and ERT system target files uses the real-time model data structure `(rtModel)`. This structure encapsulates model-specific information in a much more compact form than the simulation structure `SimStruct`. Depending on the model, the model-specific information includes data such as internal data for reusable functions and error state.

When you set **Code interface packaging** to `Reusable function`, the code generator packages model data (such as block I/O, Dwork, and parameters) into the real-time model data structure. The code generator passes the real-time model structure by reference to generated model entry-point functions. If you set **Pass root-level I/O as** to `Part of model data structure`, the code generator packages root-level model input and output into the real-time model data structure.

Efficient capabilities of the generated code depend on generation of `rtModel` rather than `SimStruct`, including:

- Integer absolute and elapsed timing services
- Independent timers for asynchronous tasks
- Generation of improved C API code for signal, state, and parameter monitoring
- Pruning the data structure to minimize its size (ERT-derived system target files only)

The real-time model data structure is defined in the `main.c` or `main.cpp` file that is generated when you build your model. For ERT models, if you clear the model configuration parameter **Generate an example main program**, `rtmodel.h` file is generated. The `rtmodel.h` file contains `#include` directives required by the `rt_main.c` or `rt_cppclass_main.cpp` static main program module. The `rt_main.c` or `rt_cppclass_main.cpp` includes `rtmodel.h` to access model-specific data structures and entry points because the static main program module is not created at code generation time.

The code generator provides a set of macros for accessing the `rtModel` data structure. These macros are analogous to the `ssSetxxx` and `ssGetxxx` macros that S-functions use to access `SimStruct` data, including noninlined S-functions compiled by the code generator.

Use the set of macros `rtmGetxxx` and `rtmSetxxx` to access the real-time model data structure. These macros are used in the generated code as well as in the `main.c` or `main.cpp` module of your external code. If you are customizing `main.c` or `main.cpp` (either a static or a generated file), use `rtmGetxxx` and `rtmSetxxx` instead of the `ssSetxxx` and `ssGetxxx` macros.

Usage of `rtmGetxxx` and `rtmSetxxx` macros is the same as for the `ssSetxxx` and `ssGetxxx` versions, except that you replace `SimStruct S` by real-time model data structure `rtM`. This table lists `rtmGetxxx` and `rtmSetxxx` macros that are used in `rt_main.c` and `rt_main.cpp`.

Macros for Accessing the Real-Time Model Data Structure

rtm Macro Syntax	Description
`rtmGetdX(rtm)`	Get the derivatives of block continuous states
`rtmGetOffsetTimePtr(RT_MDL rtM)`	Return the pointer to vector that stores sample time offsets of the model associated with rtM
`rtmGetNumSampleTimes(RT_MDL rtM)`	Get the number of sample times of a block
`rtmGetPerTaskSampleHitsPtr(RT_MDL)`	Return a pointer to `NumSampleTime` × `NumSampleTime` matrix
`rtmGetRTWExtModeInfo(RT_MDL rtM)`	Return an external mode information data structure of the model (used by code generator only for external mode)
`rtmGetRTWLogInfo(RT_MDL)`	Return a data structure used by code generator logging (used by code generator only)
`rtmGetRTWRTModelMethodsInfo(RT_MDL)`	Return a data structure of real-time model methods information (used by code generator only)
`rtmGetRTWSolverInfo(RT_MDL)`	Return data structure containing model solver information (used by code generator only)
`rtmGetSampleHitPtr(RT_MDL)`	Return a pointer to Sample Hit flag vector
`rtmGetSampleTime(RT_MDL rtM, int TID)`	Get task sample time
`rtmGetSampleTimePtr(RT_MDL rtM)`	Get pointer to a task sample time
`rtmGetSampleTimeTaskIDPtr(RT_MDL rtM)`	Get pointer to a task ID
`rtmGetSimTimeStep(RT_MDL)`	Return simulation step type ID (`MINOR_TIME_STEP`, `MAJOR_TIME_STEP`)
`rtmGetStepSize(RT_MDL)`	Return the fundamental step size of the model
`rtmGetT(RT_MDL,t)`	Get the current simulation time
`rtmSetT(RT_MDL,t)`	Set the time of the next sample hit
`rtmGetTaskTime(RT_MDL,tid)`	Get the current time for the current task
`rtmGetTFinal(RT_MDL)`	Get the simulation stop time
`rtmSetTFinal(RT_MDL,finalT)`	Set the simulation stop time
`rtmGetTimingData(RT_MDL)`	Return a data structure used by timing engine of the model (used by code generator only)
`rtmGetTPtr(RT_MDL)`	Return a pointer to the current time
`rtmGetTStart(RT_MDL)`	Get the simulation start time
`rtmIsContinuousTask(rtm)`	Determine whether a task is continuous
`rtmIsMajorTimeStep(rtm)`	Determine whether the simulation is in a major step
`rtmIsSampleHit(RT_MDL,tid)`	Determine whether the sample time is hit
`rtmGetErrorStatus(rtm)`	Get the current error status
`rtmSetErrorStatus(rtm,val)`	Set the current error status
`rtmGetErrorStatusPointer(rtm)`	Return a pointer to the current error status
`rtmGetStopRequested(rtm)`	Return whether a stop is requested

rtm Macro Syntax	Description
`rtmGetBlockIO(rtm)`	Get the block I/O data structure
`rtmSetBlockIO(rtm,val)`	Set the block I/O data structure
`rtmGetContStates(rtm)`	Get the continuous states data structure
`rtmSetContStates(rtm,val)`	Set the continuous states data structure
`rtmGetDefaultParam(rtm)`	Get the default parameters data structure
`rtmSetDefaultParam(rtm,val)`	Set the default parameters data structure
`rtmGetPrevZCSigState(rtm)`	Get the previous zero-crossing signal state data structure
`rtmSetPrevZCSigState(rtm,val)`	Set the previous zero-crossing signal state data structure
`rtmGetRootDWork(rtm)`	Get the DWork data structure
`rtmSetRootDWork(rtm,val)`	Set the DWork data structure
`rtmGetU(rtm)`	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmSetU(rtm,val)`	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmGetY(rtm)`	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
`rtmSetY(rtm,val)`	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

For more information, see “SimStruct Macros and Functions Listed by Usage”.

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment” on page 36-35
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Access Signal, State, and Parameter Data During Execution” on page 36-17
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

How Generated Code Exchanges Data with an Environment

To use the code that you generate from a model, you call the generated entry-point functions such as `step` and `initialize`. The environment in which you call these functions must provide input signal data and, depending on your application, scheduling information. The generated algorithm then calculates output data that the calling environment can use. The environment and the algorithm can exchange this data through global variables or through formal parameters (arguments).

The set of input and output data and the exchange mechanisms constitute the interfaces of the entry-point functions. When you understand the default interfaces and how to control them, you can:

- Write code that calls the generated code.
- Generate reusable (reentrant) code that you can call multiple times in a single application.
- Integrate the generated code with other, external code in your application.

In a model, root-level Inport and Outport blocks represent the primary inputs and outputs of the block algorithm. By default, the code generator aggregates these blocks into standard structures that store input and output data.

For information about how the generated code stores internal data, which does not participate in the model interface, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49.

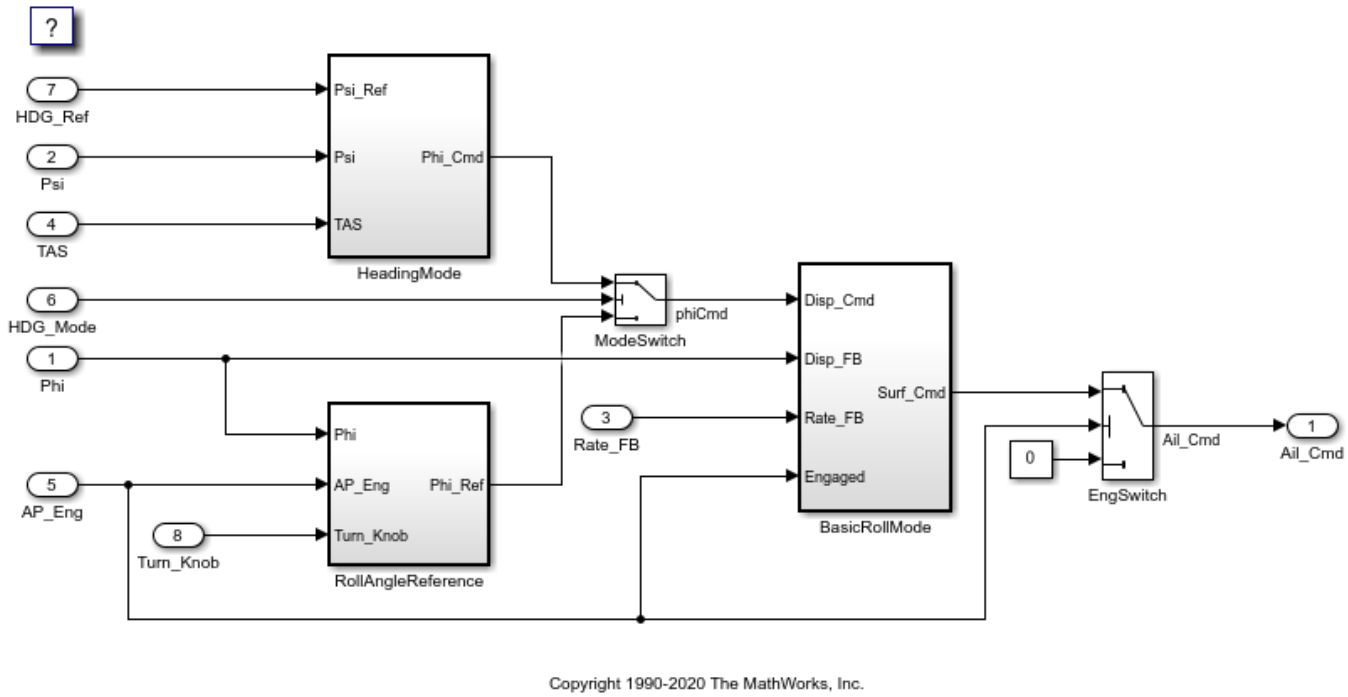
Data Interfaces in the Generated Code

This example shows how the generated code exchanges data with an environment.

Explore Example Model

Open the example model `RollAxisAutopilot`.

```
open_system('RollAxisAutopilot')
```



At the root level, the model has a number of Inport blocks and one Outport block.

In the model, set **Configuration Parameters > Code Generation > System target file** to `grt.tlc`.

```
set_param('RollAxisAutopilot','SystemTargetFile','grt.tlc')
```

Inspect the setting for **Configuration Parameters > Code Generation > Interface > Code interface packaging**. The setting `Nonreusable function` means that the generated code is not reusable (not reentrant).

For this example, to generate simpler code, clear the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Mat-file logging**.

```
set_param('RollAxisAutopilot','MatFileLogging','off')
```

Generate Nonreusable Code

Generate code from the model.

```
slbuild('RollAxisAutopilot')
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
```

```
RollAxisAutopilot  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.135s
```

Inspect the file `RollAxisAutopilot.h`. The file defines a structure type that represents input data and a type that represents output data.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.h');
rtwdemodbtype(file,...
    '/* External inputs (root inport signals with default storage) */',...
    '} ExtY_RollAxisAutopilot_T;',1,1)
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  real32_T Phi;                        /* '<Root>/Phi' */
  real32_T Psi;                        /* '<Root>/Psi' */
  real32_T Rate_FB;                    /* '<Root>/Rate_FB' */
  real32_T TAS;                        /* '<Root>/TAS' */
  boolean_T AP_Eng;                    /* '<Root>/AP_Eng' */
  boolean_T HDG_Mode;                  /* '<Root>/HDG_Mode' */
  real32_T HDG_Ref;                    /* '<Root>/HDG_Ref' */
  real32_T Turn_Knob;                  /* '<Root>/Turn_Knob' */
} ExtU_RollAxisAutopilot_T;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real32_T Ail_Cmd;                    /* '<Root>/Ail_Cmd' */
} ExtY_RollAxisAutopilot_T;
```

Each field corresponds to an Inport or Outport block at the root level of the model. The name of each field derives from the name of the block.

The file also defines a structure type, the *real-time model data structure*, whose single field stores a run-time indication of whether the generated code has encountered an error during execution.

```
rtwdemodbtype(file,'/* Real-time Model Data Structure */',...
    '/* Block states (default storage) */',1,0)
```

```
/* Real-time Model Data Structure */
struct tag_RTM_RollAxisAutopilot_T {
  const char_T *errorStatus;
};
```

When you modify model configuration parameters to suit your application, this structure can also contain other data that pertain to the entire model, such as scheduling information.

For the structure type that represents the real-time model data structure, the file `RollAxisAutopilot_types.h` creates an alias (`typedef`) that the generated code later uses to allocate memory for the structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot_types.h');
rtwdemodbtype(file,'/* Forward declaration for rtModel */',...
    'RT_MODEL_RollAxisAutopilot_T;',1,1)
```

```
/* Forward declaration for rtModel */
typedef struct tag_RTM_RollAxisAutopilot_T RT_MODEL_RollAxisAutopilot_T;
```

Using these structure types, the file `RollAxisAutopilot.c` defines (allocates memory for) global structure variables that store input and output data for the generated algorithm. The file also defines variables that represent the real-time model data structure and a pointer to the structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,...
    '/* External inputs (root inport signals with default storage) */',...
    '= &RollAxisAutopilot_M_;',1,1)
```

```
/* External inputs (root inport signals with default storage) */
ExtU_RollAxisAutopilot_T RollAxisAutopilot_U;

/* External outputs (root outports fed by signals with default storage) */
ExtY_RollAxisAutopilot_T RollAxisAutopilot_Y;

/* Real-time model */
static RT_MODEL_RollAxisAutopilot_T RollAxisAutopilot_M_;
RT_MODEL_RollAxisAutopilot_T *const RollAxisAutopilot_M = &RollAxisAutopilot_M_;
```

The file `RollAxisAutopilot.h` declares these structure variables. Your external code can include (`#include`) this file, whose general name is *`model`*`.h` where *`model`* is the name of the model, to access data that participate in the model interface.

In `RollAxisAutopilot.c`, the model `step` function, which represents the primary model algorithm, uses a `void void` interface (without arguments).

```
rtwdemodbtype(file,...
    '/* Model step function */','void RollAxisAutopilot_step(void)',1,1)
```

```
/* Model step function */
void RollAxisAutopilot_step(void)
```

In the function definition, the algorithm reads input data and writes output data by directly accessing the global structure variables. For example, near the end of the algorithm, the code for the `EngSwitch` block reads the value of the `AP_Eng` field (which represents an input) and, based on that value, conditionally writes a constant value to the `Ail_Cmd` field (which represents an output).

```
rtwdemodbtype(file,'if (RollAxisAutopilot_U.AP_Eng) {','      }',1,1)
```

```
  if (RollAxisAutopilot_U.AP_Eng) {
    /* Outputs for Atomic SubSystem: '<Root>/BasicRollMode' */
    /* Saturate: '<S1>/CmdLimit' */
    if (RollAxisAutopilot_Y.Ail_Cmd > 15.0F) {
      /* Sum: '<S1>/Sum2' incorporates:
       *  Outport: '<Root>/Ail_Cmd'
       */
      RollAxisAutopilot_Y.Ail_Cmd = 15.0F;
    } else if (RollAxisAutopilot_Y.Ail_Cmd < -15.0F) {
      /* Sum: '<S1>/Sum2' incorporates:
       *  Outport: '<Root>/Ail_Cmd'
       */
```

```
      RollAxisAutopilot_Y.Ail_Cmd = -15.0F;
    }

    /* End of Saturate: '<S1>/CmdLimit' */
    /* End of Outputs for SubSystem: '<Root>/BasicRollMode' */
  } else {
    /* Sum: '<S1>/Sum2' incorporates:
     *  Constant: '<Root>/Zero'
     *  Outport: '<Root>/Ail_Cmd'
     */
    RollAxisAutopilot_Y.Ail_Cmd = 0.0F;
  }

  /* End of Switch: '<Root>/EngSwitch' */
}

/* Model initialize function */
void RollAxisAutopilot_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(RollAxisAutopilot_M, (NULL));

  /* states (dwork) */
  (void) memset((void *)&RollAxisAutopilot_DW, 0,
                sizeof(DW_RollAxisAutopilot_T));

  /* external inputs */
  (void)memset(&RollAxisAutopilot_U, 0, sizeof(ExtU_RollAxisAutopilot_T));

  /* external outputs */
  RollAxisAutopilot_Y.Ail_Cmd = 0.0F;

  /* SystemInitialize for Atomic SubSystem: '<Root>/RollAngleReference' */
  /* InitializeConditions for UnitDelay: '<S4>/FixPt Unit Delay1' */
  RollAxisAutopilot_DW.FixPtUnitDelay1_DSTATE = 0.0F;

  /* End of SystemInitialize for SubSystem: '<Root>/RollAngleReference' */

  /* SystemInitialize for Atomic SubSystem: '<Root>/BasicRollMode' */
  /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_DW.Integrator_DSTATE = 0.0F;
  RollAxisAutopilot_DW.Integrator_PrevResetState = 0;

  /* End of SystemInitialize for SubSystem: '<Root>/BasicRollMode' */
}

/* Model terminate function */
void RollAxisAutopilot_terminate(void)
{
  /* (no terminate code required) */
}
```

Due to the `void void` interface and the direct data access, the function is not reentrant. If you call the function multiple times in an application, each call reads and writes input and output data to the

same global structure variables, resulting in data corruption and unintentional interaction between the calls.

The model initialization function, `RollAxisAutopilot_initialize`, initializes input and output data to zero. The function also initializes the error status. The function directly accesses the global variables, which means the function is not reentrant.

```
rtwdemodbtype(file,'/* Model initialize function */',...
    'rtmSetErrorStatus(RollAxisAutopilot_M, (NULL));',1,1)
```

```
/* Model initialize function */
void RollAxisAutopilot_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(RollAxisAutopilot_M, (NULL));
```

```
rtwdemodbtype(file,'  /* external inputs */',...
    '/* SystemInitialize for Atomic SubSystem:',1,0)
```

```
  /* external inputs */
  (void)memset(&RollAxisAutopilot_U, 0, sizeof(ExtU_RollAxisAutopilot_T));

  /* external outputs */
  RollAxisAutopilot_Y.Ail_Cmd = 0.0F;
```

Generate Reusable Code

You can configure the generated code as reentrant, which means that you can call the entry-point functions multiple times in an application. With this configuration, instead of directly accessing global variables, the entry-point functions exchange input, output, and other model data through formal parameters (pointer arguments). With these pointer arguments, each call can read inputs and write outputs to a set of separate global variables, preventing unintentional interaction between the calls.

In the model, set **Configuration Parameters > Code Generation > Interface > Code interface packaging** to `Reusable function`.

```
set_param('RollAxisAutopilot','CodeInterfacePackaging','Reusable function')
```

Generate code from the model.

```
slbuild('RollAxisAutopilot')
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
RollAxisAutopilot  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.175s
```

Now, in `RollAxisAutopilot.h`, the real-time model data structure contains pointers to the error indication, the input data, the output data, and additional data in the form of a `DWork` substructure (which stores, for example, block states such as the state of a Discrete-Time Integrator block).

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.h');
rtwdemodbtype(file,'/* Real-time Model Data Structure */',...
    '/* External data declarations for dependent source files */',1,0)
```

```
/* Real-time Model Data Structure */
struct tag_RTM_RollAxisAutopilot_T {
  const char_T *errorStatus;
  ExtU_RollAxisAutopilot_T *inputs;
  ExtY_RollAxisAutopilot_T *outputs;
  DW_RollAxisAutopilot_T *dwork;
};
```

To call the generated code multiple times in an application, your code must allocate memory for a real-time model data structure per call. The file `RollAxisAutopilot.c` defines a specialized function that allocates memory for a new real-time model data structure and returns a pointer to the structure. The function also allocates memory for the substructures that the fields in the model data structure point to, such as the input and output structures.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,'/* Model data allocation function */',...
    'RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot(void)',1,1)
```

```
/* Model data allocation function */
RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot(void)
```

In `RollAxisAutopilot.c`, the model `step` function accepts an argument that represents the real-time model data structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,...
    '/* Model step function */','void RollAxisAutopilot_step',1,1)
```

```
/* Model step function */
void RollAxisAutopilot_step(RT_MODEL_RollAxisAutopilot_T *const
```

In the function definition, the algorithm first extracts each pointer from the real-time model data structure into a local variable.

```
rtwdemodbtype(file,...
    '*RollAxisAutopilot_DW =','RollAxisAutopilot_M->outputs;',1,1)
```

```
  DW_RollAxisAutopilot_T *RollAxisAutopilot_DW = RollAxisAutopilot_M->dwork;
  ExtU_RollAxisAutopilot_T *RollAxisAutopilot_U = (ExtU_RollAxisAutopilot_T *)
    RollAxisAutopilot_M->inputs;
  ExtY_RollAxisAutopilot_T *RollAxisAutopilot_Y = (ExtY_RollAxisAutopilot_T *)
    RollAxisAutopilot_M->outputs;
```

Then, to access the input and output data stored in global memory, the algorithm interacts with these local variables.

```
rtwdemodbtype(file,'  if (RollAxisAutopilot_U->AP_Eng) {',...
    '  /* End of Switch: ''<Root>/EngSwitch'' */',1,1)
```

```
  if (RollAxisAutopilot_U->AP_Eng) {
    /* Outputs for Atomic SubSystem: '<Root>/BasicRollMode' */
    /* Saturate: '<S1>/CmdLimit' */
    if (RollAxisAutopilot_Y->Ail_Cmd > 15.0F) {
      /* Sum: '<S1>/Sum2' incorporates:
       *  Outport: '<Root>/Ail_Cmd'
       */
      RollAxisAutopilot_Y->Ail_Cmd = 15.0F;
    } else if (RollAxisAutopilot_Y->Ail_Cmd < -15.0F) {
      /* Sum: '<S1>/Sum2' incorporates:
       *  Outport: '<Root>/Ail_Cmd'
       */
      RollAxisAutopilot_Y->Ail_Cmd = -15.0F;
    }

    /* End of Saturate: '<S1>/CmdLimit' */
    /* End of Outputs for SubSystem: '<Root>/BasicRollMode' */
  } else {
    /* Sum: '<S1>/Sum2' incorporates:
     *  Constant: '<Root>/Zero'
     *  Outport: '<Root>/Ail_Cmd'
     */
    RollAxisAutopilot_Y->Ail_Cmd = 0.0F;
  }
```

Similarly, the model initialization function accepts the real-time model data structure as an argument.

```
rtwdemodbtype(file,...
    '/* Model initialize function */','void RollAxisAutopilot_initialize',1,1)
```

```
/* Model initialize function */
void RollAxisAutopilot_initialize(RT_MODEL_RollAxisAutopilot_T *const
```

Because each call that you make to an entry-point function interacts with a separate real-time model data structure, you avoid unintentional interaction between the calls.

Configure Data Interface

To control the characteristics of the data interface in the generated code, see “Control Data and Function Interface in Generated Code” on page 36-44.

See Also

Related Examples

- “Data Structures in the Generated Code” on page 36-31
- “Choose an External Code Integration Workflow” on page 58-2
- “Analyze Generated Data Code Interface” on page 54-13

- “Control Data and Function Interface in Generated Code” on page 36-44
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

Control Data and Function Interface in Generated Code

To use the code that you generate from a model, you call generated entry-point functions such as `step` and `initialize`. The calling environment and the generated functions exchange input and output data through global variables or through formal parameters (arguments). This data and the exchange mechanisms constitute the interfaces of the entry-point functions. For information about the default interfaces for reentrant and nonreentrant models in the generated code, see “How Generated Code Exchanges Data with an Environment” on page 36-35.

By controlling the interfaces that appear in the generated code, you can:

- Minimize the modifications that you must make to existing code.
- Generate stable interfaces that do not change or minimally change when you make changes to the model.
- Generate code that exchanges data more efficiently (for example, by using pointers and pass-by-reference arguments for nonscalar data).

Control Type Names, Field Names, and Variable Names of Standard I/O Structures (Embedded Coder)

By default, for nonreentrant code, Inport blocks at the root level of the model appear in the generated code as fields of a global structure variable. Similarly, Outport blocks appear in a different structure. For reentrant code, depending on your setting for model configuration parameter **Pass root-level I/O**, the code generator can also package input and output data into standard structures.

With Embedded Coder, you can control these names. See “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2.

Control Names of Generated Entry-Point Functions (Embedded Coder)

To use the generated code, you write code that calls generated entry-point functions. For example, entry-point functions include *`model`*`_step`, *`model`*`_initialize`, and top-level functions generated from an export-function model. To control the names of the model entry-point functions, use the Code Mappings editor (requires Embedded Coder) to apply a combination of these techniques:

- On the **Function Defaults** tab, specify default naming rules for categories of entry-point functions by applying function customization templates in the Code Mappings editor. With this technique, a naming rule applies to functions in a category. For more information, see “Configure Default Code Generation for Functions” on page 33-20.
- On the **Functions** tab, specify names for individual entry-point functions by either directly editing the **Function Name** column or through a configuration dialog box opened from the **Function Preview** column. Names that you specify override default naming rules specified by function customization templates. For more information, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

Control Data Interface for Nonreentrant Code

When you set the model configuration parameter **Code interface packaging** to `Nonreusable function` (the default), generated entry-point functions are not reentrant. Typically, the functions exchange data with the calling environment through direct access to global variables.

Configure Inport or Outport Block as Separate Global Variable

To remove the block from the standard I/O structures by creating a separate global variable, apply a storage class, such as `ExportedGlobal` or `ExportToFile`, to the signal that the block represents. You can configure a default storage class for categories of data elements, such as Inport blocks. As you add such blocks to the model, they acquire the storage class that you specify. You can use the Code Mappings editor also to configure individual blocks. You might do this if a model has just a few data elements of a specific category or override the default configuration settings.

For an example, see “Design Data Interface by Configuring Inport and Outport Blocks”. For general information about configuring data for code generation, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Configure Generated Code to Read or Write to Global Variables Defined by External Code

If your calling code already defines a global variable that you want the generated code to use as input data or use to store output data, you can reuse the variable by preventing the code generator from duplicating the definition. Apply a storage class to the corresponding Inport or Outport block in the model. Choose a storage class that specifies an imported data scope, such as `ImportedExtern` or `ImportFromFile`. For information about applying storage classes, see “C Code Generation Configuration for Model Interface Elements” on page 33-2 and “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.

Package Multiple Inputs or Outputs into Custom Structure

You can configure a single Inport or Outport block to appear in the generated code as a custom structure that contains multiple input or output signals. You can also configure the block to appear as a substructure of the default I/O structures or as a separate structure variable.

Configure the block as a nonvirtual bus by using a `Simulink.Bus` object as the data type of the block. If your external code defines the structure type, consider using the `Simulink.importExternalCTypes` function to generate the bus object.

- To generate the bus signal as a substructure in the standard I/O structures, leave the block storage class at the default setting, `Auto`. If you have Embedded Coder, in the Code Mappings editor, on the **Data Defaults** tab, set the storage class for categories **Inports** and **Outports** to `Default`.
- To generate the bus signal as a separate global structure variable, apply a storage class such as `ExportedGlobal` or `ExportToFile`.

For more information about grouping signals into custom structures in the generated code, see “Organize Data into Structures in Generated Code” on page 36-147.

Configure Inport or Outport Block as Function Call (Embedded Coder)

If your external code defines a function that returns input data for the generated code or accepts output data that the generated code produces, you can configure an Inport or Outport block so that the generated code calls the function instead of accessing a global variable. Apply the Embedded Coder storage class `GetSet`. For more information, see “Access Data Through Functions with Storage Class GetSet” on page 40-25.

Pass Inputs and Outputs Through Function Arguments (Embedded Coder)

With Embedded Coder, you can optionally configure the model step (execution) function to access root-level input and output through arguments instead of directly reading and writing to global

variables. Completely control the argument characteristics such as name, order, and passing mechanism (by reference or by value). This level of configuration can help to integrate generated code with your external code.

To pass inputs and outputs through arguments, in the Configure C Step Function interface dialog box, select **Configure arguments for Step function prototype**. Each Inport and Outport block at the root level of the model appears in the code as an argument of the execution function. For more information, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

Configure Referenced Model Inputs and Outputs as Global Variables (void-void)

By default, for a nonreentrant referenced model, the generated code passes root-level input and output through function arguments. A nonreentrant referenced model is one in which you set model configuration parameter **Total number of instances allowed per top model** to `One`.

To pass this data through global variables instead (for a `void-void` interface), in the referenced model, apply storage classes such as `ExportedGlobal` and `ExportToFile` to root-level Inport and Outport blocks.

- To apply a default storage class to such blocks, use the Code Mapping Editor. With this technique, as you add such blocks to the model, the blocks acquire the default storage class, preserving the `void-void` interface. For more information, see “Configure Default Code Generation for Data” on page 33-16.
- To override the default storage class, apply storage classes directly to individual blocks by using the Code Mapping editor. See “C Code Generation Configuration for Model Interface Elements” on page 33-2.
- For an example that requires Embedded Coder, see “Establish Data Ownership in a Model Hierarchy” on page 37-10.

Control Data Interface for Reentrant Code

When you set **Code interface packaging** to `Reusable function`, generated entry-point functions are reentrant. The functions exchange data with the calling environment through formal parameters (arguments). By default, each root-level Inport and Outport block appears in the generated code as a separate argument instead of a field of the standard I/O structures.

- To reduce the number of arguments, see “Reduce Number of Arguments by Using Structures”.
- To control the data types of the arguments, see “Control Data Types of Arguments”.

Prevent Unintended Changes to the Interface

Some changes that you make to a model change the entry-point function interfaces in the generated code. For example, if you change the name of the model, the names of the functions can change. If you configure the model code to exchange data through arguments, when you add or remove Inport or Outport blocks or change the names of the blocks, the corresponding arguments can change.

For easier maintenance of your calling code, prevent changes to the interfaces of the entry-point functions.

- If you exchange input and output data through arguments, configure the generated code to package Inport and Outport blocks into structures instead of allowing each block to appear as a

separate argument (the default). Then, when you add or remove Inport or Outport blocks, or change their properties such as name and data type, the fields of the structures change, but the function interfaces do not change. See “Reduce Number of Arguments by Using Structures” on page 36-47.

- Set the data types of Inport and Outport blocks explicitly instead of using an inherited data type setting (which these blocks use by default). Inherited data type settings can cause the blocks to use different data types depending on the data types of upstream and downstream signals. For more information about configuring data types, see “Control Data Types of Signals”.
- With Embedded Coder, specify function names that do not depend on the model name. If you specify a naming rule with a function customization template, do not use the token `$R` in the rule. See “Control Names of Generated Entry-Point Functions (Embedded Coder)”.

Reduce Number of Arguments by Using Structures

Reducing the number of arguments of a function can improve the readability of the code and reduce consumption of stack memory. To create a structure argument that can pass multiple pieces of data at one time, use these techniques:

- Manually combine multiple Inport or Outport blocks so that they appear in the generated code as fields of a structure or substructure of a standard data structure. The generated entry-point function or functions accept the address of the structure as a separate argument or as a substructure (field) of the standard I/O structures.

  Replace the Inport or Outport blocks with a single block, and configure the new block as a nonvirtual bus by using a `Simulink.Bus` object as the data type of the block. If your external code defines the structure type, consider using the `Simulink.importExternalCTypes` function to generate the bus object. See “Organize Data into Structures in Generated Code” on page 36-147 and `Simulink.importExternalCTypes`.
- When you generate reentrant code with Embedded Coder, configure Inport and Outport blocks to appear in aggregated structures by default. Set model configuration parameter **Pass root-level I/O as** to a setting other than `Individual arguments`.

  - To package Inport and Outport blocks into the real-time model data structure, select `Part of model data structure`. The code generator aggregates the blocks into the default I/O structures to which the real-time model data structure points. The generated entry-point function or functions accept the real-time model data structure as a single argument. If you choose this setting, the function has the smallest number of arguments.
  - To aggregate Inport blocks into a structure and Outport blocks into a different structure, select `Structure reference`. The generated entry-point function or functions accept the address of each structure as an argument. Another argument accepts the real-time model data structure. If you choose this setting, the inputs and outputs are more identifiable in the function interfaces.

    With this setting, if you remove the root-level Inport blocks or the root-level Outport blocks, the function signature can change. Similarly, the signature can change if you add a root-level Inport or Outport block to a model that does not include such blocks.
  - For more control over the characteristics of the structures, set **Pass root-level I/O as** to `Part of model data structure`. Then, set the default storage class for Inport and Outport blocks to a structured storage class that you create. With this technique:

- You can create a single structure for the blocks or create two separate structures.
- You can control the names of the structure types.
- The structures appear as substructures of the real-time model data structure. You must set **Pass root-level I/O as** to `Part of model data structure`.

Create the storage class by using the Embedded Coder Dictionary (see "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2). Apply the storage class by using the Code Mappings editor (see "Configure Default Code Generation for Data" on page 33-16).

For more information, see Pass root-level I/O as.

Control Data Types of Arguments

You can configure the generated entry-point functions to exchange data through arguments. For a scalar or array argument, to control the name of the primitive data type, use a `Simulink.AliasType` object to either set the data type of the corresponding block or to configure data type replacements for the entire model. These techniques require Embedded Coder. For more information, see "Manage Replacement of Simulink Data Types in Generated Code" on page 38-2.

Promote Data Item to the Interface

By default, the code generator assumes that Inport and Outport blocks at the root level of the model constitute the data interface of the model. You can promote an arbitrary signal, block parameter, or block state to the interface so that other systems and components can access it. See "Promote Internal Data to the Interface" on page 36-66.

See Also

Related Examples

- "Design Data Interface by Configuring Inport and Outport Blocks" on page 36-169
- "Define Interfaces of Model Components"
- "Analyze Generated Data Code Interface" on page 54-13
- "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114
- "How Generated Code Exchanges Data with an Environment" on page 36-35
- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 58-61
- "Configure Generated Code According to Interface Control Document Specifications" on page 40-54
- "C Code Generation Configuration for Model Interface Elements" on page 33-2

How Generated Code Stores Internal Signal, State, and Parameter Data

To calculate outputs from inputs, the generated code stores some internal data in global memory. A signal that does not connect to a root-level input or output (Inport or Outport block) is internal data.

Internal data can also include:

- A block state such as the state of a Unit Delay block. The algorithm must retain the state value between execution cycles, so the generated code typically stores states in global memory (for example, as a global variable or a field of a global structure variable).
- A block parameter, such as the **Gain** parameter of a Gain block, whose value the code generator cannot inline in the code. For example, the code generator cannot inline the value of a nonscalar parameter.
- The status indicator of a conditionally executed subsystem such as an enabled subsystem.

For more efficient code, you can configure optimizations such as **Configuration Parameters > Default parameter behavior** and **Configuration Parameters > Signal storage reuse** that attempt to eliminate storage for internal data. However, the optimizations cannot eliminate storage for some data, which consume memory in the generated code.

When you understand the default format in which the generated code stores internal data, you can:

- Make signals accessible and parameters tunable by default. You can then interact with and monitor the code during execution.
- Generate efficient production code by eliminating storage for internal data and, depending on your hardware and build toolchain, controlling the placement in memory of data that the optimizations cannot eliminate.
- Promote pieces of internal data to the model interface so that other components and systems can access that data.

For information about how the generated code exchanges data with a calling environment through interfaces, see “How Generated Code Exchanges Data with an Environment” on page 36-35.

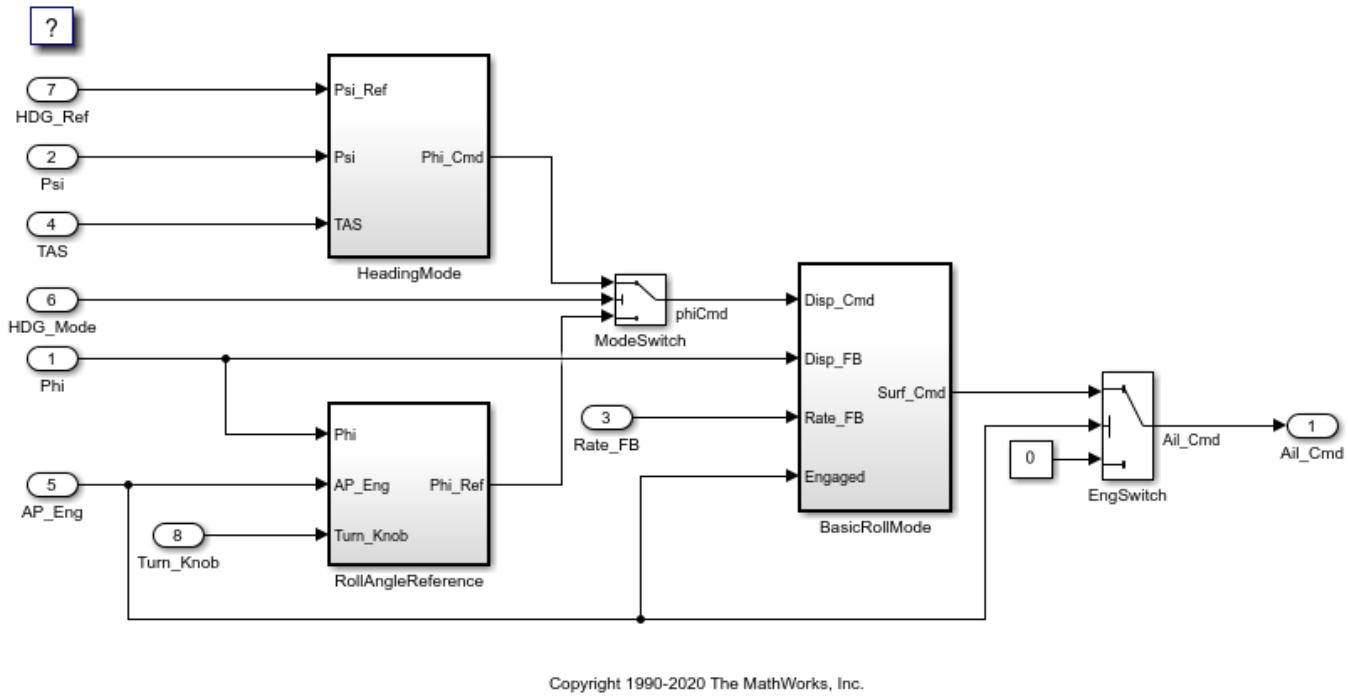
Internal Data in the Generated Code

This example shows how the generated code stores internal data such as block states.

Explore Example Model

Open the example model `RollAxisAutopilot`.

```
open_system('RollAxisAutopilot')
```



Copyright 1990-2020 The MathWorks, Inc.

The model contains internal signals that do not connect to root-level Inport or Outport blocks. Some of the signals have a name, such as the `phiCmd` signal.

The model also contains some blocks that maintain state data. For example, in the `BasicRollMode` subsystem, a Discrete-Time Integrator block labeled `Integrator` maintains a state.

In the model, set **Configuration Parameters > Code Generation > System target file** to `grt.tlc`.

```
set_param('RollAxisAutopilot','SystemTargetFile','grt.tlc')
```

Inspect the setting for **Configuration Parameters > Code Generation > Interface > Code interface packaging**. The setting `Nonreusable function` means that the generated code is not reusable (reentrant).

For this example, generate simpler code by clearing **Configuration Parameters > Code Generation > Interface > Advanced parameters > Mat-file logging**.

```
set_param('RollAxisAutopilot','MatFileLogging','off')
```

Generate Nonreusable Code

Set these configuration parameters:

- Set **Default parameter behavior** to `Tunable`.
- Clear **Signal storage reuse**.

```
set_param('RollAxisAutopilot','DefaultParameterBehavior','Tunable',...
    'OptimizeBlockIOStorage','off')
```

Generate code from the model.

```
slbuild('RollAxisAutopilot')

### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================================
RollAxisAutopilot  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 33.39s
```

The file `RollAxisAutopilot.h` defines several structure types that represent internal data. For example, the block input and output structure defines one field for each internal signal in the model. Each field name derives from the name of the block that generates the signal or, if you specify a name for the signal, from the name of the signal.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.h');
rtwdemodbtype(file,...
    '/* Block signals (default storage) */','} B_RollAxisAutopilot_T;',1,1)
```

```
/* Block signals (default storage) */
typedef struct {
  real32_T phiCmd;                      /* '<Root>/ModeSwitch' */
  real32_T hdgError;                    /* '<S2>/Sum' */
  real32_T DispGain;                    /* '<S2>/DispGain' */
  real32_T Product;                     /* '<S2>/Product' */
  real32_T Abs;                         /* '<S3>/Abs' */
  real32_T FixPtUnitDelay1;             /* '<S4>/FixPt Unit Delay1' */
  real32_T Xnew;                        /* '<S4>/Enable' */
  real32_T TKSwitch;                    /* '<S3>/TKSwitch' */
  real32_T RefSwitch;                   /* '<S3>/RefSwitch' */
  real32_T Integrator;                  /* '<S1>/Integrator' */
  real32_T DispLimit;                   /* '<S1>/DispLimit' */
  real32_T Sum;                         /* '<S1>/Sum' */
  real32_T DispGain_f;                  /* '<S1>/DispGain' */
  real32_T RateLimit;                   /* '<S1>/RateLimit' */
  real32_T Sum1;                        /* '<S1>/Sum1' */
  real32_T RateGain;                    /* '<S1>/RateGain' */
  real32_T Sum2;                        /* '<S1>/Sum2' */
  real32_T CmdLimit;                    /* '<S1>/CmdLimit' */
  real32_T IntGain;                     /* '<S1>/IntGain' */
  boolean_T NotEngaged;                 /* '<S3>/NotEngaged' */
  boolean_T TKThreshold;                /* '<S3>/TKThreshold' */
  boolean_T RefThreshold2;              /* '<S3>/RefThreshold2' */
  boolean_T RefThreshold1;              /* '<S3>/RefThreshold1' */
  boolean_T Or;                         /* '<S3>/Or' */
  boolean_T NotEngaged_e;               /* '<S1>/NotEngaged' */
} B_RollAxisAutopilot_T;
```

The file defines a structure type, the DWork structure, to represent block states such as the state of the Discrete-Time Integrator block.

```
rtwdemodbtype(file,...
    '/* Block states (default storage) for system','} DW_RollAxisAutopilot_T;',1,1)
```

```
/* Block states (default storage) for system '<Root>' */
typedef struct {
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S4>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} DW_RollAxisAutopilot_T;
```

The file defines a structure type to represent parameter data. Each tunable block parameter in the model, such as the **Gain** parameter of a Gain block, appears as a field of this structure. If a block parameter acquires its value from a MATLAB® variable or a `Simulink.Parameter` object, the variable or object appears as a field, not the block parameter.

The file also defines a structure type, the *real-time model data structure*, whose single field represents a run-time indication of whether the generated code has encountered an error during execution.

```
rtwdemodbtype(file,'/* Real-time Model Data Structure */',...
    '/* Block parameters (default storage) */',1,0)
```

```
/* Real-time Model Data Structure */
struct tag_RTM_RollAxisAutopilot_T {
  const char_T *errorStatus;
};
```

For the structure type that represents the real-time model data structure, the file `RollAxisAutopilot_types.h` creates an alias that the generated code later uses to allocate memory for the structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot_types.h');
rtwdemodbtype(file,'/* Forward declaration for rtModel */',...
    'RT_MODEL_RollAxisAutopilot_T;',1,1)
```

```
/* Forward declaration for rtModel */
typedef struct tag_RTM_RollAxisAutopilot_T RT_MODEL_RollAxisAutopilot_T;
```

Using these structure types, the file `RollAxisAutopilot.c` defines (allocates memory for) global structure variables that store internal data for the generated algorithm. The file also defines variables that represent the real-time model data structure and a pointer to the structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,'/* Block signals (default storage) */',...
    '= &RollAxisAutopilot_M_;',1,1)
```

```
/* Block signals (default storage) */
B_RollAxisAutopilot_T RollAxisAutopilot_B;

/* Block states (default storage) */
DW_RollAxisAutopilot_T RollAxisAutopilot_DW;

/* External inputs (root inport signals with default storage) */
ExtU_RollAxisAutopilot_T RollAxisAutopilot_U;
```

```
/* External outputs (root outports fed by signals with default storage) */
ExtY_RollAxisAutopilot_T RollAxisAutopilot_Y;

/* Real-time model */
static RT_MODEL_RollAxisAutopilot_T RollAxisAutopilot_M_;
RT_MODEL_RollAxisAutopilot_T *const RollAxisAutopilot_M = &RollAxisAutopilot_M_;
```

The model `step` function, which represents the primary model algorithm, uses a `void void` interface (with no arguments).

```
rtwdemodbtype(file,...
    '/* Model step function */','void RollAxisAutopilot_step(void)',1,1)
```

```
/* Model step function */
void RollAxisAutopilot_step(void)
```

In the function definition, the algorithm performs calculations and stores intermediate results in the signal and state structures by directly accessing the global variables. The algorithm also reads parameter data from the corresponding global variable. For example, in the `BasicRollMode` subsystem, the code generated for the `Integrator` block reads and writes signal, state, and parameter data from the structures.

```
rtwdemodbtype(file,'/* DiscreteIntegrator: ''<S1>/Integrator'' *',...
    '/* End of DiscreteIntegrator: ''<S1>/Integrator'' */',1,1)
```

```
  /* DiscreteIntegrator: '<S1>/Integrator' */
  if (RollAxisAutopilot_B.NotEngaged_e ||
      (RollAxisAutopilot_DW.Integrator_PrevResetState != 0)) {
    RollAxisAutopilot_DW.Integrator_DSTATE = RollAxisAutopilot_P.Integrator_IC;
  }

  if (RollAxisAutopilot_DW.Integrator_DSTATE >= RollAxisAutopilot_P.intLim) {
    RollAxisAutopilot_DW.Integrator_DSTATE = RollAxisAutopilot_P.intLim;
  } else if (RollAxisAutopilot_DW.Integrator_DSTATE <=
             RollAxisAutopilot_P.Integrator_LowerSat) {
    RollAxisAutopilot_DW.Integrator_DSTATE =
      RollAxisAutopilot_P.Integrator_LowerSat;
  }

  /* DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_B.Integrator = RollAxisAutopilot_DW.Integrator_DSTATE;

  /* Saturate: '<S1>/DispLimit' */
  u0 = RollAxisAutopilot_B.phiCmd;
  u1 = RollAxisAutopilot_P.DispLimit_LowerSat;
  u2 = RollAxisAutopilot_P.dispLim;
  if (u0 > u2) {
    /* Saturate: '<S1>/DispLimit' */
    RollAxisAutopilot_B.DispLimit = u2;
  } else if (u0 < u1) {
    /* Saturate: '<S1>/DispLimit' */
    RollAxisAutopilot_B.DispLimit = u1;
  } else {
    /* Saturate: '<S1>/DispLimit' */
    RollAxisAutopilot_B.DispLimit = u0;
```

```
}

/* End of Saturate: '<S1>/DispLimit' */

/* Sum: '<S1>/Sum' incorporates:
 *  Inport: '<Root>/Phi'
 */
RollAxisAutopilot_B.Sum = RollAxisAutopilot_B.DispLimit -
  RollAxisAutopilot_U.Phi;

/* Gain: '<S1>/DispGain' */
RollAxisAutopilot_B.DispGain_f = RollAxisAutopilot_P.dispGain *
  RollAxisAutopilot_B.Sum;

/* Saturate: '<S1>/RateLimit' */
u0 = RollAxisAutopilot_B.DispGain_f;
u1 = RollAxisAutopilot_P.RateLimit_LowerSat;
u2 = RollAxisAutopilot_P.rateLim;
if (u0 > u2) {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B.RateLimit = u2;
} else if (u0 < u1) {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B.RateLimit = u1;
} else {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B.RateLimit = u0;
}

/* End of Saturate: '<S1>/RateLimit' */

/* Sum: '<S1>/Sum1' incorporates:
 *  Inport: '<Root>/Rate_FB'
 */
RollAxisAutopilot_B.Sum1 = RollAxisAutopilot_B.RateLimit -
  RollAxisAutopilot_U.Rate_FB;

/* Gain: '<S1>/RateGain' */
RollAxisAutopilot_B.RateGain = RollAxisAutopilot_P.rateGain *
  RollAxisAutopilot_B.Sum1;

/* Sum: '<S1>/Sum2' */
RollAxisAutopilot_B.Sum2 = RollAxisAutopilot_B.Integrator +
  RollAxisAutopilot_B.RateGain;

/* Saturate: '<S1>/CmdLimit' */
u0 = RollAxisAutopilot_B.Sum2;
u1 = RollAxisAutopilot_P.CmdLimit_LowerSat;
u2 = RollAxisAutopilot_P.cmdLim;
if (u0 > u2) {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B.CmdLimit = u2;
} else if (u0 < u1) {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B.CmdLimit = u1;
} else {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B.CmdLimit = u0;
```

```
  }

  /* End of Saturate: '<S1>/CmdLimit' */

  /* Gain: '<S1>/IntGain' */
  RollAxisAutopilot_B.IntGain = RollAxisAutopilot_P.intGain *
    RollAxisAutopilot_B.Sum1;

  /* Update for DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_DW.Integrator_DSTATE +=
    RollAxisAutopilot_P.Integrator_gainval * RollAxisAutopilot_B.IntGain;
  if (RollAxisAutopilot_DW.Integrator_DSTATE >= RollAxisAutopilot_P.intLim) {
    RollAxisAutopilot_DW.Integrator_DSTATE = RollAxisAutopilot_P.intLim;
  } else if (RollAxisAutopilot_DW.Integrator_DSTATE <=
             RollAxisAutopilot_P.Integrator_LowerSat) {
    RollAxisAutopilot_DW.Integrator_DSTATE =
      RollAxisAutopilot_P.Integrator_LowerSat;
  }

  RollAxisAutopilot_DW.Integrator_PrevResetState = (int8_T)
    RollAxisAutopilot_B.NotEngaged_e;

  /* End of Update for DiscreteIntegrator: '<S1>/Integrator' */
  /* End of Outputs for SubSystem: '<Root>/BasicRollMode' */

  /* Switch: '<Root>/EngSwitch' incorporates:
   *  Inport: '<Root>/AP_Eng'
   */
  if (RollAxisAutopilot_U.AP_Eng) {
    /* Outport: '<Root>/Ail_Cmd' */
    RollAxisAutopilot_Y.Ail_Cmd = RollAxisAutopilot_B.CmdLimit;
  } else {
    /* Outport: '<Root>/Ail_Cmd' incorporates:
     *  Constant: '<Root>/Zero'
     */
    RollAxisAutopilot_Y.Ail_Cmd = RollAxisAutopilot_P.Zero_Value_c;
  }

  /* End of Switch: '<Root>/EngSwitch' */
}

/* Model initialize function */
void RollAxisAutopilot_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(RollAxisAutopilot_M, (NULL));

  /* block I/O */
  (void) memset(((void *) &RollAxisAutopilot_B), 0,
                sizeof(B_RollAxisAutopilot_T));

  /* states (dwork) */
  (void) memset((void *)&RollAxisAutopilot_DW, 0,
                sizeof(DW_RollAxisAutopilot_T));

  /* external inputs */
```

```
  (void)memset(&RollAxisAutopilot_U, 0, sizeof(ExtU_RollAxisAutopilot_T));

  /* external outputs */
  RollAxisAutopilot_Y.Ail_Cmd = 0.0F;

  /* SystemInitialize for Atomic SubSystem: '<Root>/RollAngleReference' */
  /* InitializeConditions for UnitDelay: '<S4>/FixPt Unit Delay1' */
  RollAxisAutopilot_DW.FixPtUnitDelay1_DSTATE =
    RollAxisAutopilot_P.LatchPhi_vinit;

  /* End of SystemInitialize for SubSystem: '<Root>/RollAngleReference' */

  /* SystemInitialize for Atomic SubSystem: '<Root>/BasicRollMode' */
  /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_DW.Integrator_DSTATE = RollAxisAutopilot_P.Integrator_IC;
  RollAxisAutopilot_DW.Integrator_PrevResetState = 0;

  /* End of SystemInitialize for SubSystem: '<Root>/BasicRollMode' */
}

/* Model terminate function */
void RollAxisAutopilot_terminate(void)
{
  /* (no terminate code required) */
}
```

Due to the `void void` interface and the direct data access, the function is not reentrant. If you call the function multiple times in an application, each call writes data to the global structure variables and the subsequent call can read that data, resulting in unintentional interference between the calls.

The model initialization function `RollAxisAutopilot_initialize` initializes all of the internal data to zero. The function also initializes the error status by calling a specialized macro function. The initialization function directly accesses the global variables, which means that the function is not reentrant.

```
rtwdemodbtype(file,'/* Model initialize function */',...
    'sizeof(DW_RollAxisAutopilot_T));',1,1)
```

```
/* Model initialize function */
void RollAxisAutopilot_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(RollAxisAutopilot_M, (NULL));

  /* block I/O */
  (void) memset(((void *) &RollAxisAutopilot_B), 0,
                sizeof(B_RollAxisAutopilot_T));

  /* states (dwork) */
  (void) memset((void *)&RollAxisAutopilot_DW, 0,
                sizeof(DW_RollAxisAutopilot_T));
```

The function then initializes the block states in the DWork structure to the initial values that the block parameters in the model specify. Two of the three states in the model have tunable initial values, so the code initializes them by reading data from the parameters structure.

```
rtwdemodbtype(file,...
    '/* SystemInitialize for Atomic SubSystem: ''<Root>/RollAngleReference'' */',...
    '/* Model terminate function */',1,0)
```

```
  /* SystemInitialize for Atomic SubSystem: '<Root>/RollAngleReference' */
  /* InitializeConditions for UnitDelay: '<S4>/FixPt Unit Delay1' */
  RollAxisAutopilot_DW.FixPtUnitDelay1_DSTATE =
    RollAxisAutopilot_P.LatchPhi_vinit;

  /* End of SystemInitialize for SubSystem: '<Root>/RollAngleReference' */

  /* SystemInitialize for Atomic SubSystem: '<Root>/BasicRollMode' */
  /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_DW.Integrator_DSTATE = RollAxisAutopilot_P.Integrator_IC;
  RollAxisAutopilot_DW.Integrator_PrevResetState = 0;

  /* End of SystemInitialize for SubSystem: '<Root>/BasicRollMode' */
}
```

Generate Reusable Code

You can configure the generated code as reentrant, which means you can call the entry-point functions multiple times in an application. With this configuration, instead of directly accessing global variables, the entry-point functions accept internal data through formal parameters (pointer arguments). With these pointer arguments, each call can maintain internal data in a set of separate global variables, preventing unintentional interaction between the calls.

In the model, set **Configuration Parameters > Code Generation > Interface > Code interface packaging** to `Reusable function`.

```
set_param('RollAxisAutopilot','CodeInterfacePackaging','Reusable function')
```

Generate code from the model.

```
slbuild('RollAxisAutopilot')
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
RollAxisAutopilot  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 22.83s
```

Now, in `RollAxisAutopilot.h`, the real-time model data structure contains pointers to the error indication, the internal data, and primary input and output data in the form of `ExtU` and `ExtY` substructures (the fields of which represent Inport and Outport blocks at the root level of the model).

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.h');
rtwdemodbtype(file,'/* Real-time Model Data Structure */',...
    '/* External data declarations for dependent source files */',1,0)
```

```
/* Real-time Model Data Structure */
struct tag_RTM_RollAxisAutopilot_T {
  const char_T *errorStatus;
  B_RollAxisAutopilot_T *blockIO;
  ExtU_RollAxisAutopilot_T *inputs;
  ExtY_RollAxisAutopilot_T *outputs;
  DW_RollAxisAutopilot_T *dwork;
};

/* Block parameters (default storage) */
extern P_RollAxisAutopilot_T RollAxisAutopilot_P;
```

To call the generated code multiple times in an application, your code must allocate memory for a real-time model data structure per call. The file `RollAxisAutopilot.c` defines a specialized function that allocates memory for a new real-time model data structure and returns a pointer to the structure. The function also allocates memory for the substructures that the fields in the model data structure point to, such as the DWork structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,'/* Model data allocation function */',...
    'RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot(void)',1,1)
```

```
/* Model data allocation function */
RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot(void)
```

The model `step` function accepts an argument that represents the real-time model data structure.

```
rtwdemodbtype(file,'/* Model step function */','void RollAxisAutopilot_step',1,1)
```

```
/* Model step function */
void RollAxisAutopilot_step(RT_MODEL_RollAxisAutopilot_T *const
```

In the function definition, the algorithm first extracts each pointer from the real-time model data structure into a local variable.

```
rtwdemodbtype(file,'*RollAxisAutopilot_B =','RollAxisAutopilot_M->outputs;',1,1)
```

```
  B_RollAxisAutopilot_T *RollAxisAutopilot_B = RollAxisAutopilot_M->blockIO;
  DW_RollAxisAutopilot_T *RollAxisAutopilot_DW = RollAxisAutopilot_M->dwork;
  ExtU_RollAxisAutopilot_T *RollAxisAutopilot_U = (ExtU_RollAxisAutopilot_T *)
    RollAxisAutopilot_M->inputs;
  ExtY_RollAxisAutopilot_T *RollAxisAutopilot_Y = (ExtY_RollAxisAutopilot_T *)
    RollAxisAutopilot_M->outputs;
```

Then, to access the internal data stored in global memory, the algorithm interacts with these local variables.

```
rtwdemodbtype(file,'/* DiscreteIntegrator: ''<S1>/Integrator'' */',...
    '/* End of DiscreteIntegrator: ''<S1>/Integrator'' */',1,1)
```

```
  /* DiscreteIntegrator: '<S1>/Integrator' */
  if (RollAxisAutopilot_B->NotEngaged_e ||
      (RollAxisAutopilot_DW->Integrator_PrevResetState != 0)) {
    RollAxisAutopilot_DW->Integrator_DSTATE = RollAxisAutopilot_P.Integrator_IC;
```

```
}

if (RollAxisAutopilot_DW->Integrator_DSTATE >= RollAxisAutopilot_P.intLim) {
  RollAxisAutopilot_DW->Integrator_DSTATE = RollAxisAutopilot_P.intLim;
} else if (RollAxisAutopilot_DW->Integrator_DSTATE <=
           RollAxisAutopilot_P.Integrator_LowerSat) {
  RollAxisAutopilot_DW->Integrator_DSTATE =
    RollAxisAutopilot_P.Integrator_LowerSat;
}

/* DiscreteIntegrator: '<S1>/Integrator' */
RollAxisAutopilot_B->Integrator = RollAxisAutopilot_DW->Integrator_DSTATE;

/* Saturate: '<S1>/DispLimit' */
u0 = RollAxisAutopilot_B->phiCmd;
u1 = RollAxisAutopilot_P.DispLimit_LowerSat;
u2 = RollAxisAutopilot_P.dispLim;
if (u0 > u2) {
  /* Saturate: '<S1>/DispLimit' */
  RollAxisAutopilot_B->DispLimit = u2;
} else if (u0 < u1) {
  /* Saturate: '<S1>/DispLimit' */
  RollAxisAutopilot_B->DispLimit = u1;
} else {
  /* Saturate: '<S1>/DispLimit' */
  RollAxisAutopilot_B->DispLimit = u0;
}

/* End of Saturate: '<S1>/DispLimit' */

/* Sum: '<S1>/Sum' */
RollAxisAutopilot_B->Sum = RollAxisAutopilot_B->DispLimit -
  RollAxisAutopilot_U->Phi;

/* Gain: '<S1>/DispGain' */
RollAxisAutopilot_B->DispGain_f = RollAxisAutopilot_P.dispGain *
  RollAxisAutopilot_B->Sum;

/* Saturate: '<S1>/RateLimit' */
u0 = RollAxisAutopilot_B->DispGain_f;
u1 = RollAxisAutopilot_P.RateLimit_LowerSat;
u2 = RollAxisAutopilot_P.rateLim;
if (u0 > u2) {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B->RateLimit = u2;
} else if (u0 < u1) {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B->RateLimit = u1;
} else {
  /* Saturate: '<S1>/RateLimit' */
  RollAxisAutopilot_B->RateLimit = u0;
}

/* End of Saturate: '<S1>/RateLimit' */

/* Sum: '<S1>/Sum1' */
RollAxisAutopilot_B->Sum1 = RollAxisAutopilot_B->RateLimit -
  RollAxisAutopilot_U->Rate_FB;
```

```
/* Gain: '<S1>/RateGain' */
RollAxisAutopilot_B->RateGain = RollAxisAutopilot_P.rateGain *
  RollAxisAutopilot_B->Sum1;

/* Sum: '<S1>/Sum2' */
RollAxisAutopilot_B->Sum2 = RollAxisAutopilot_B->Integrator +
  RollAxisAutopilot_B->RateGain;

/* Saturate: '<S1>/CmdLimit' */
u0 = RollAxisAutopilot_B->Sum2;
u1 = RollAxisAutopilot_P.CmdLimit_LowerSat;
u2 = RollAxisAutopilot_P.cmdLim;
if (u0 > u2) {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B->CmdLimit = u2;
} else if (u0 < u1) {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B->CmdLimit = u1;
} else {
  /* Saturate: '<S1>/CmdLimit' */
  RollAxisAutopilot_B->CmdLimit = u0;
}

/* End of Saturate: '<S1>/CmdLimit' */

/* Gain: '<S1>/IntGain' */
RollAxisAutopilot_B->IntGain = RollAxisAutopilot_P.intGain *
  RollAxisAutopilot_B->Sum1;

/* Update for DiscreteIntegrator: '<S1>/Integrator' */
RollAxisAutopilot_DW->Integrator_DSTATE +=
  RollAxisAutopilot_P.Integrator_gainval * RollAxisAutopilot_B->IntGain;
if (RollAxisAutopilot_DW->Integrator_DSTATE >= RollAxisAutopilot_P.intLim) {
  RollAxisAutopilot_DW->Integrator_DSTATE = RollAxisAutopilot_P.intLim;
} else if (RollAxisAutopilot_DW->Integrator_DSTATE <=
           RollAxisAutopilot_P.Integrator_LowerSat) {
  RollAxisAutopilot_DW->Integrator_DSTATE =
    RollAxisAutopilot_P.Integrator_LowerSat;
}

RollAxisAutopilot_DW->Integrator_PrevResetState = (int8_T)
  RollAxisAutopilot_B->NotEngaged_e;

/* End of Update for DiscreteIntegrator: '<S1>/Integrator' */
/* End of Outputs for SubSystem: '<Root>/BasicRollMode' */

/* Switch: '<Root>/EngSwitch' */
if (RollAxisAutopilot_U->AP_Eng) {
  /* Outport: '<Root>/Ail_Cmd' */
  RollAxisAutopilot_Y->Ail_Cmd = RollAxisAutopilot_B->CmdLimit;
} else {
  /* Outport: '<Root>/Ail_Cmd' incorporates:
   *  Constant: '<Root>/Zero'
   */
  RollAxisAutopilot_Y->Ail_Cmd = RollAxisAutopilot_P.Zero_Value_c;
}
```

```
  /* End of Switch: '<Root>/EngSwitch' */
}

/* Model initialize function */
void RollAxisAutopilot_initialize(RT_MODEL_RollAxisAutopilot_T *const
  RollAxisAutopilot_M)
{
  DW_RollAxisAutopilot_T *RollAxisAutopilot_DW = RollAxisAutopilot_M->dwork;

  /* SystemInitialize for Atomic SubSystem: '<Root>/RollAngleReference' */
  /* InitializeConditions for UnitDelay: '<S4>/FixPt Unit Delay1' */
  RollAxisAutopilot_DW->FixPtUnitDelay1_DSTATE =
    RollAxisAutopilot_P.LatchPhi_vinit;

  /* End of SystemInitialize for SubSystem: '<Root>/RollAngleReference' */

  /* SystemInitialize for Atomic SubSystem: '<Root>/BasicRollMode' */
  /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
  RollAxisAutopilot_DW->Integrator_DSTATE = RollAxisAutopilot_P.Integrator_IC;
  RollAxisAutopilot_DW->Integrator_PrevResetState = 0;

  /* End of SystemInitialize for SubSystem: '<Root>/BasicRollMode' */
}

/* Model terminate function */
void RollAxisAutopilot_terminate(RT_MODEL_RollAxisAutopilot_T
  * RollAxisAutopilot_M)
{
  /* model code */
  rt_FREE(RollAxisAutopilot_M->blockIO);
  rt_FREE(RollAxisAutopilot_M->inputs);
  rt_FREE(RollAxisAutopilot_M->outputs);
  rt_FREE(RollAxisAutopilot_M->dwork);
  rt_FREE(RollAxisAutopilot_M);
}

/* Model data allocation function */
RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot(void)
{
  RT_MODEL_RollAxisAutopilot_T *RollAxisAutopilot_M;
  RollAxisAutopilot_M = (RT_MODEL_RollAxisAutopilot_T *) malloc(sizeof
    (RT_MODEL_RollAxisAutopilot_T));
  if (RollAxisAutopilot_M == (NULL)) {
    return (NULL);
  }

  (void) memset((char *)RollAxisAutopilot_M, 0,
                sizeof(RT_MODEL_RollAxisAutopilot_T));

  /* block I/O */
  {
    B_RollAxisAutopilot_T *b = (B_RollAxisAutopilot_T *) malloc(sizeof
      (B_RollAxisAutopilot_T));
    rt_VALIDATE_MEMORY(RollAxisAutopilot_M,b);
    RollAxisAutopilot_M->blockIO = (b);
  }

  /* states (dwork) */
```

```
  {
    DW_RollAxisAutopilot_T *dwork = (DW_RollAxisAutopilot_T *) malloc(sizeof
      (DW_RollAxisAutopilot_T));
    rt_VALIDATE_MEMORY(RollAxisAutopilot_M,dwork);
    RollAxisAutopilot_M->dwork = (dwork);
  }

  /* external inputs */
  {
    ExtU_RollAxisAutopilot_T *RollAxisAutopilot_U = (ExtU_RollAxisAutopilot_T *)
      malloc(sizeof(ExtU_RollAxisAutopilot_T));
    rt_VALIDATE_MEMORY(RollAxisAutopilot_M,RollAxisAutopilot_U);
    RollAxisAutopilot_M->inputs = (((ExtU_RollAxisAutopilot_T *)
      RollAxisAutopilot_U));
  }

  /* external outputs */
  {
    ExtY_RollAxisAutopilot_T *RollAxisAutopilot_Y = (ExtY_RollAxisAutopilot_T *)
      malloc(sizeof(ExtY_RollAxisAutopilot_T));
    rt_VALIDATE_MEMORY(RollAxisAutopilot_M,RollAxisAutopilot_Y);
    RollAxisAutopilot_M->outputs = (RollAxisAutopilot_Y);
  }

  {
    B_RollAxisAutopilot_T *RollAxisAutopilot_B = RollAxisAutopilot_M->blockIO;
    DW_RollAxisAutopilot_T *RollAxisAutopilot_DW = RollAxisAutopilot_M->dwork;
    ExtU_RollAxisAutopilot_T *RollAxisAutopilot_U = (ExtU_RollAxisAutopilot_T *)
      RollAxisAutopilot_M->inputs;
    ExtY_RollAxisAutopilot_T *RollAxisAutopilot_Y = (ExtY_RollAxisAutopilot_T *)
      RollAxisAutopilot_M->outputs;

    /* block I/O */
    (void) memset(((void *) RollAxisAutopilot_B), 0,
                  sizeof(B_RollAxisAutopilot_T));

    /* states (dwork) */
    (void) memset((void *)RollAxisAutopilot_DW, 0,
                  sizeof(DW_RollAxisAutopilot_T));

    /* external inputs */
    (void)memset(RollAxisAutopilot_U, 0, sizeof(ExtU_RollAxisAutopilot_T));

    /* external outputs */
    RollAxisAutopilot_Y->Ail_Cmd = 0.0F;
  }

  return RollAxisAutopilot_M;
}
```

Similarly, the model initialization function accepts the real-time model data structure as an argument.

```
rtwdemodbtype(file,...
    '/* Model initialize function */','void RollAxisAutopilot_initialize',1,1)
```

```
/* Model initialize function */
void RollAxisAutopilot_initialize(RT_MODEL_RollAxisAutopilot_T *const
```

Because each call that you make to an entry-point function interacts with a separate real-time model data structure, you avoid unintentional interaction between the calls.

Eliminate Internal Data with Code Generation Optimizations

For more efficient code that consumes less memory, select the optimizations, such as **Default parameter behavior**, that you cleared earlier.

```
set_param('RollAxisAutopilot','DefaultParameterBehavior','Inlined',...
    'OptimizeBlockIOStorage','on',...
    'LocalBlockOutputs','on')
```

In this example, for simpler code, set **Code interface packaging** to `Nonreusable function`.

```
set_param('RollAxisAutopilot','CodeInterfacePackaging','Nonreusable function')
```

Generate code from the model.

```
slbuild('RollAxisAutopilot')
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
RollAxisAutopilot  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 18.309s
```

Now, `RollAxisAutopilot.h` does not define a structure for block inputs and outputs. For all of the internal signals in the model, the optimizations either eliminated storage or created local function variables instead of global structure fields.

The optimizations were not able to eliminate storage for the three block states, so the file continues to define the DWork structure type.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.h');
rtwdemodbtype(file,...
    '/* Block states (default storage) for system','} DW_RollAxisAutopilot_T;',1,1)
```

```
/* Block states (default storage) for system '<Root>' */
typedef struct {
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S4>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} DW_RollAxisAutopilot_T;
```

The code generated for the Discrete-Time Integrator block now stores state and output data only in the DWork structure.

```
file = fullfile('RollAxisAutopilot_grt_rtw','RollAxisAutopilot.c');
rtwdemodbtype(file,'/* Update for DiscreteIntegrator: ''<S1>/Integrator''',...
    '/* End of Update for DiscreteIntegrator: ''<S1>/Integrator'' */',1,1)
```

```
  /* Update for DiscreteIntegrator: '<S1>/Integrator' incorporates:
   *  Gain: '<S1>/IntGain'
   */
  RollAxisAutopilot_DW.Integrator_DSTATE += 0.5F * rtb_TKSwitch * 0.025F;
  if (RollAxisAutopilot_DW.Integrator_DSTATE >= 5.0F) {
    RollAxisAutopilot_DW.Integrator_DSTATE = 5.0F;
  } else if (RollAxisAutopilot_DW.Integrator_DSTATE <= -5.0F) {
    RollAxisAutopilot_DW.Integrator_DSTATE = -5.0F;
  }

  RollAxisAutopilot_DW.Integrator_PrevResetState = (int8_T)tmp;
```

The optimizations also eliminated storage for the block parameters in the model. For example, in the Discrete-Time Integrator block, the **Upper saturation limit** and **Lower saturation limit** parameters are set to `intLim` and `-intLim`. `intLim` is a `Simulink.Parameter` object that stores the value `5`. In the code generated for the Discrete-Time Integrator, these block parameters and `intLim` appear as inlined literal numbers `5.0F` and `-5.0F`.

If a model contains a parameter that the code generator cannot inline directly (for example, an array parameter), the code defines a structure type that represents the data. This *constant parameters* structure uses the `const` storage type qualifier, so some build toolchains can optimize the assembly code further.

The declaration of the structure, `ConstParam__model_`, is in *`model`*`.h`.

```
/* Constant parameters (auto storage) */
typedef struct {
   /* Expression: [1 2 3 4 5 6 7]
    * Referenced by: '<Root>/Constant'
    */
   real_T Constant_Value[7];

   /* Expression: [7 6 5 4 3 2 1]
    * Referenced by: '<Root>/Gain'
    */
   real_T Gain_Gain[7];
 } ConstParam_model;
```

The declaration of the constant parameters, *`model`*`_ConstP`, is in *`model`*`_data.c` by default.

```
/* Constant parameters (auto storage) */
const ConstParam_model model_ConstP = {
   /* Expression: [1 2 3 4 5 6 7]
    * Referenced by: '<Root>/Constant'
    */
   { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 },

   /* Expression: [7 6 5 4 3 2 1]
    * Referenced by: '<Root>/Gain'
    */
   { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 }
};
```

*model*_ConstP is passed as an argument to referenced models.

Local Variables in the Generated Code

When you select the optimization **Configuration Parameters > Enable local block outputs**, the code generator attempts to yield more efficient code by representing internal signals as local variables instead of fields of a global structure. If the memory consumed by local variables risks exceeding the stack space available on your target hardware, consider indicating the maximum stack size by setting **Configuration Parameters > Maximum stack size (bytes)**. For more information, see Maximum stack size (bytes).

Appearance of Test Points in the Generated Code

A *test point* is a signal that is stored in a unique memory location. For information about including test points in your model, see “Configure Signals as Test Points”.

When you generate code for models that include test points, the build process allocates a separate memory buffer for each test point. By default, test points are stored as members of a standard data structure such as *model*_B.

If you have Embedded Coder:

- You can control the default representation of test points by specifying code generation settings for the **Internal data** category of data in the Code Mapping Editor (see “Configure Default Code Generation for Data” on page 33-16).
- You can specify that the build process ignore test points in the model, allowing optimal buffer allocation, by using the Ignore test point signals parameter. Ignoring test points facilitates transitioning from prototyping to deployment and avoids accidental degradation of generated code due to workflow artifacts. See Ignore test point signals.

Virtual buses do not appear in generated code, even when associated with a test point. To display a bus in generated code, use a nonvirtual bus or a virtual bus converted to a nonvirtual bus by a Signal Conversion block.

Appearance of Workspace Variables in the Generated Code

Workspace variables are variables that you use to specify block parameter values in a model. Workspace variables include numeric MATLAB variables and `Simulink.Parameter` objects that you store in a workspace, such as the base workspace, or in a data dictionary.

When you set **Default parameter behavior** to `Tunable`, by default, workspace variables appear in the generated code as tunable fields of the global parameters structure. If you use such a variable to specify multiple block parameter values, the variable appears as a single field of the global parameters structure. The code does not create multiple fields to represent the block parameters. Therefore, tuning the field value during code execution changes the mathematical behavior of the model in the same way as tuning the value of the MATLAB variable or parameter object during simulation.

If you have Embedded Coder, you can control the default representation of workspace variables by specifying code generation settings for categories of parameter data in the Code Mapping Editor (see “Configure Default Code Generation for Data” on page 33-16).

- The **Model parameters** category applies to variables that you store in a model workspace.
- The **External parameters** category applies to variables that you store in the base workspace or a data dictionary.

Promote Internal Data to the Interface

By default, the code generator assumes that other systems and components in your application do not need to access internal data. For example, internal data are subject to optimizations that can eliminate them from the generated code. For prototyping and testing purposes, you can access internal data by clearing the optimizations or by configuring test points and applying storage classes (see "Preserve Variables in Generated Code" on page 36-2). For optimized production code, configure individual data items to appear in the generated code as part of the model interface.

Data That You Can Promote

Depending on the reentrancy of the generated code, that is, the setting that you choose for **Code interface packaging**, you can configure each data item in a model to participate in the interface by appearing in the code as one of these entities:

- A global symbol, such as a global variable or a call to a specialized function
- A formal parameter (argument) of the generated entry-point functions

The table shows the mechanisms that each category of data can use to participate in the interface.

Category of Data	Appear as Global Symbol	Appear as Argument of Entry-Point Function
Root-level Inport or Outport block	Only for a nonreentrant model.	Yes.
Signal connecting two blocks	Only for a nonreentrant model.	Only for a reentrant model and only as a field of a structure. Alternatively, connect the signal to a root-level Outport block.
Block state	Only for a nonreentrant model.	Only for a reentrant model and only as a field of a structure.
Data store such as a Data Store Memory block	Yes.	Only for a reentrant model and only as a field of a structure.
Block parameter or parameter object such as `Simulink.Parameter`	Yes.	Only as a field of a structure.

Single-Instance Algorithm

For a single-instance algorithm (you set **Code interface packaging** to `Nonreusable function`), apply storage classes directly to individual data items by using the Model Data Editor or the Property Inspector. With a directly applied storage class, a data item appears in the code as a global symbol such as a global variable. The storage class also prevents optimizations from eliminating storage for the data item.

You can apply storage classes to signals, block states, and block parameters. (For block parameters, you apply storage classes indirectly through parameter objects such as `Simulink.Parameter`).

However, for a signal, consider connecting the signal to an Outport block at the root level of the model. Then, optionally, you can apply a storage class to the block. In the block diagram, the Outport block shows that the signal represents a system output.

For more information about storage classes, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Reentrant Algorithm

For a reentrant algorithm (you set **Code interface packaging** to `Reusable function`), use different techniques to configure data items to appear in the code as formal parameters (arguments) of the generated entry-point functions.

- For an internal signal, directly apply the storage class `Model default` (see “C Code Generation Configuration for Model Interface Elements” on page 33-2). If you have Embedded Coder, in the Code Mapping Editor, for the **Internal data** category, set the default storage class to `Default` or to a structured storage class that you define in an Embedded Coder Dictionary (see “Create Code Definitions for Use in the Code Mappings Editor” on page 32-2). Alternatively, configure the signal as a test point (see “Appearance of Test Points in the Generated Code” on page 36-65). By default, the signal appears as a field of one of the standard data structures (see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49). If you do not want the signal to appear in production code, use a test point so that you can later select the model configuration parameter **Ignore test point signals**.

  Alternatively, connect the signal to an Outport block at the root level of the model. Connecting the signal to a root-level Outport block prevents optimizations from eliminating the signal from the code. To help with signal routing in a large model, use Goto and From blocks.
- For a block parameter, create a parameter object such as `Simulink.Parameter` and directly apply a storage class other than `Auto` to the object. The storage class prevents optimizations from inlining the parameter value in the code.

  With the storage class, the parameter is shared between instances of the model, which are calls to the entry-point functions. The functions access the parameter data directly, as a global symbol, not an argument. To enable each instance of the model to use a different value for the parameter, configure the parameter as a model argument. For more information, see “Specify Instance-Specific Parameter Values for Reusable Referenced Model”.

  For information about applying storage classes, see “C Code Generation Configuration for Model Interface Elements”.

Control Default Representation of Internal Data (Embedded Coder)

By default, the code generator aggregates internal data that optimizations cannot eliminate, such as most state data, into standard structures such as the DWork structure. With Embedded Coder, you can control how the generated code stores this data.

Control Placement of Data in Memory by Inserting Pragmas

Use the Code Mapping Editor to specify a default memory section for each category of data such as states and signals (**Internal data**). In the generated code, your custom pragmas or other decorations surround the data definitions and declarations.

You can also partition the structures according to atomic subsystems in your model so that you can specify different default memory sections for the data of subroutines and other algorithmic subcomponents.

For more information, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Control Names of Types, Fields, and Global Variables for Standard Data Structures

You can control some characteristics of the standard data structures. For more information, see “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2.

For additional control over structure characteristics, such as placement in generated code files, create your own structured storage class by using the Embedded Coder Dictionary. Then, apply the storage class to categories of data by using the Code Mapping Editor. The storage class removes the data from the standard structures, creating other structures that you can more finely control. For more information about applying default storage classes to categories of data, see “Configure Default Code Generation for Data” on page 33-16. For more information about creating a storage class, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

Organize Data into Structures According to Subcomponents

- In the standard data structures, to create substructures that contain the data for a single-instance (nonreentrant) subroutine or subcomponent, use an atomic subsystem to encapsulate the corresponding blocks. In the subsystem parameters, set **Function packaging** to `Reusable function`. For more information, see “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2.

  Alternatively, encapsulate the blocks in a model and use a Model block. In the referenced model, set **Configuration Parameters** > **Model Referencing** > **Total number of instances allowed per top model** to `Multiple`. For more information, see “Generate Code for Model Reference Hierarchy” on page 6-2.
- To create separate, standalone structures that contain the data for a multi-instance (reentrant) subroutine or subcomponent, in the model, use an atomic subsystem to encapsulate the corresponding blocks. In the subsystem parameters, set **Function packaging** to `Nonreusable function` and select **Function with separate data**. For more information, see “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2.

  Alternatively, encapsulate the blocks in a model and use a Model block. In the referenced model, choose one of these techniques:

  - Set **Configuration Parameters** > **Model Referencing** > **Total number of instances allowed per top model** to `One`. For more information, see “Generate Code for Model Reference Hierarchy” on page 6-2.
  - Set **Total number of instances allowed per top model** to `Multiple` and create a structured storage class by using an Embedded Coder Dictionary. Then, apply the storage class to categories of data by using the Code Mapping Editor. To create a storage class, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2. To apply the storage class to categories of data, see “Configure Default Code Generation for Data” on page 33-16.

Organize Signal and Parameter Data into Meaningful, Custom Structures and Substructures

To organize arbitrary signals and parameters into custom structures and substructures, create nonvirtual bus signals and parameter structures. Optionally, to prevent optimizations from eliminating the data from the code, set the storage class of a bus signal or parameter structure to a value other than `Auto` (the default setting).

As you add blocks to the model, you must explicitly place each new signal and parameter into a bus or a structure.

For more information, see “Organize Data into Structures in Generated Code” on page 36-147.

Create Separate Global Variables Instead of Structure Fields

To make a category of internal data appear in the generated code as separate, unstructured global variables instead of fields of a standard data structure, apply an unstructured storage class to the data category by using the Code Mapping Editor. For example, apply the storage class `ExportedGlobal`. However, if you generate multi-instance, reentrant code by setting the configuration parameter **Code interface packaging** to a value other than `Nonreusable function`, you cannot use this technique for some categories of data (see “Storage Classes and Reentrant, Multi-Instance Models and Components”).

To apply default storage classes to categories of data by using the Code Mapping Editor, see “Configure Default Code Generation for Data” on page 33-16. To choose a storage class, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

See Also

Related Examples

- “Data Structures in the Generated Code” on page 36-31
- “How Generated Code Exchanges Data with an Environment” on page 36-35
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Choose Data Configuration Approach

When you generate code from a model, the code includes design data such as signals, states, and parameters, which your application code can read from and write to. When deciding how you want this data represented in the generated code, consider:

- What types of data elements do I want to use in my model?
- Where do I want to store the data?
- What are my code generation objectives?
- How can I configure the data to meet my code generation objectives?

The answers to these questions depend on your model architecture (standalone model, standalone hierarchy of referenced models, or a system of components), your application type (reentrant or single-instance), and whether you are integrating your code with existing external code.

Data Element Categories

The possible data elements in your model fall under the data element types listed in this table.

Data Element Type	Description
Inports	Root-level input ports of a model, such as Inport and Bus Element In blocks.
Outports	Root-level output ports of a model, such as Outport and Bus Element Out blocks.
Model-scoped parameters	• Model parameter arguments — Variables in the model workspace that you configure as model arguments. These parameters are exposed at the model block to enable the parent model to provide different values for each model instance. • Model parameters — Parameters that are defined in the model workspace. Unlike model arguments, these parameters are shared by each instance of the model.
External parameters	Parameters that you define as objects in the base workspace or in a data dictionary. Multiple models in an application can use these parameters.

Data Element Type	Description
Data stores	• Internal data (local data stores) — Data stores that are accessible from anywhere in a subsystem hierarchy that is at or below the level at which you define the data store. Defined graphically in a model by including a Data Store Memory block or by creating a signal object in the model workspace. • Global data stores — Data stores that are defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. • Shared local data stores — Data Store Memory blocks that have the block parameter **Share across model instances** set. These data stores are accessible only in the system where the Data Store Memory block is located. The data store value is shared across instances of the model.
Internal data	Local data, such as internal signals and block states. Includes local data stores.
Constants	Constant parameters and constant-value block output in a model.

The code configuration options available for your data are dependent on the category of data that you are configuring. Where you store the data determines if you configure it by using a data object or by using a code mapping that is saved with the model.

Where to Store Data Elements Used by Your Model

Where you store data for your model can depend on the type of model architecture that you are using and your modeling goals:

- MATLAB base workspace — Used for the temporary storage of data while experimenting with modeling techniques.
- Model workspace — Used to permanently store data that is local to a single model. Because the data is stored with the model, using this workspace can improve model portability. The model workspace is predominantly used to store parameters, arguments, and numeric variables. These elements are not typically stored in the model workspace:
  - Signal objects — You cannot configure a signal object in the model workspace for code generation. You do not need to define a signal object to configure a signal for code generation.
  - Data type objects.
- Simulink data dictionary — Used to permanently store global data, data types, and data that is shared between models.

For more information, see “Determine Where to Store Variables and Objects for Simulink Models”.

When to Configure a Data Element for Code Generation

Whether you customize the generation of your data for code generation depends on your objectives for the generated code. If you choose not to configure customizations, the code generator determines whether to eliminate or change the representation of the data in generated code for optimization purposes.

There are many reasons you might choose to configure a data item. Configuring data prevents optimizations that might eliminate the data item from the generated code and enables you to configure how the generated code stores and defines the data item. For example, if you configure a data item, you can:

- Tune parameters.
- Monitor signals and states during execution.
- Configure data placement in memory.
- Exchange data with external code.
- Improve readability and traceability of generated code.

Configure a Data Element for Code Generation

To configure your data for code generation, apply a storage class. A storage class is a code generation setting that enables you to control the appearance and placement of a data element in the generated code and prevents optimizations that might eliminate storage for that data element. You can use storage classes available with Simulink Coder and Embedded Coder. If you have special requirements that are not met by these storage classes and you are generating code for an ERT-based target, you can define and use a new storage class. For more information on storage classes and storage class properties, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

When to Configure Data by Using a Data Object

For most model-owned data, you can map a storage class directly to a data item by using the Code Mappings editor or code mapping API. You must configure some data that your model uses for code generation by using a data object, including:

- Parameters — Includes parameters in the base workspace, model workspace, or data dictionary. Configure parameters by using a `Simulink.Parameter` object.
- Global data stores — Configure by using a signal object.

Before configuring numeric variables for code generation, you must convert them to `Simulink.Parameter` objects.

Configure Model-Owned Data by Using Code Mapping

Model-owned data consists of outport blocks, signal lines, block states, data stores, and parameter objects in the model workspace. For this data, you can map a storage class to a data element in the coder app by using the Code Mappings editor or by using the code mappings API. For convenience, you can also configure external data objects in the Code Mappings editor, though for external data objects, the storage class is stored on the data object and not in the model. To see these objects, in the editor, click the **Refresh** link to the right of the category name.

To configure model-owned data, you first configure default code generation settings for categories of data elements. Using a default mapping saves time and reduces the risk of introducing errors in the code, especially when your model contains a significant number of elements in a particular category.

- In the **C Code** tab, select **Code Interface > Default Code Mappings**.
- On the **Data Defaults** tab, select a data category, then select a default storage class for that category. If you do not specify a storage class for a category, the `Default` storage class applies. In

this case, the code generator places the code for the category of data elements in standard structures. Other storage class options include:

- Relevant built-in storage classes, such as `ExportedGlobal`.
- Relevant storage classes in an available package, such as `ImportFromFile` (requires Embedded Coder).
- Storage classes defined in an Embedded Coder Dictionary (requires Embedded Coder). For more information on available storage classes, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

- In the Property Inspector, configure the properties for that storage class. (For example, header and definition files).

For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

For certain categories of data, including inports, outports, data stores, and block states, the default mapping is likely the only configuration you need to perform. But for data elements such as parameters and internal signals, you might need to specify an individual mapping to prevent the data from being optimized away. To apply a storage class to an individual data element:

- In the Code Mappings editor, find the data element on its corresponding category tab (for example, to configure an individual parameter, select the **Parameters** tab).
- Select a storage class for the data element. To apply the default configuration, select the storage class `Model default`.
- In the Property Inspector, configure the properties for that storage class. (For example, header and definition files).

For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Configure External Data

Configure external data objects for code generation by using the Code Mappings editor or the Model Explorer. Select a data object in the **Contents** pane. Then, in the **Dialog** pane, on the **Code Generation** tab, select a storage class and set properties for that storage class. For convenience, you can also configure external data objects in the Code Mappings editor. To view and configure these objects, in the editor, click the **Refresh** link to the right of the category name.

You can configure external data objects programmatically. For example, to configure a `Simulink.Parameter` object for code generation by applying the storage class `ExportedGlobal`, you can use code similar to:

```
P = Simulink.Parameter;
P.Value = 5;
P.DataType = 'int32';
P.CoderInfo.StorageClass = 'ExportedGlobal';
```

Load Storage Class Packages into Embedded Coder Dictionary

To apply a storage class from a storage class package other than Simulink (such as a package that you create):

1. Open the **Embedded Coder** app.

2. Open the Embedded Coder Dictionary. On the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.
3. In the Embedded Coder Dictionary, click **Storage Class**.
4. Below the **Storage Classes** table, click **Manage Packages**.
5. In the Manage Packages dialog box, select a package. If the package of interest is not in the list, click **Refresh**. Then, select your package.
6. Click **Load**.
7. Close the Embedded Coder Dictionary.
8. Use the Code Mappings editor or code mappings API to configure the model's interface elements.

For more information, see **Embedded Coder Dictionary**, **Code Mappings Editor – C**, and `coder.mapping.api.CodeMapping`.

To apply the storage class by using a data object (required parameters), instead of creating a `Simulink.Signal` or `Simulink.Parameter` object, create a `myPackage.Signal` or `myPackage.Parameter` object. To create data objects from your package, see “Create Data Objects from Another Data Class Package”. For an example that shows how to create and apply your own storage class, see “Create and Apply a Storage Class” on page 40-14.

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

More About

- “Choose Storage Class for Controlling Data Representation in Generated Code”
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15
- “C Code Generation Configuration for Model Interface Elements”

Choose Storage Class for Controlling Data Representation in Generated Code

A storage class is a code generation setting that you apply to data, such as parameters, signals, and states. During code configuration, use a storage class to control the appearance and placement of a data element in the generated code and to prevent optimizations from eliminating storage for that data element.

For model data, you can apply a storage class directly to a model data element by using the Code Mappings editor or code mappings API. Using either the editor or the API, apply a default storage class for categories of data, and then override that setting, as needed, for individual data elements.

For external data, you apply a storage class to a data object by using the Model Explorer or programmatically by using functions such as `get_param` and `set_param`.

Once you have specified a storage class for an element or category of elements, you can set properties for that storage class, such as header files, definition files, and memory sections.

Storage Class Properties

The property settings for a storage class define how and where the code generator represents data in the generated code. The properties that the user of a storage class can configure vary depending on the storage class. For most individual elements, the user can configure an identifier for naming the data element in the code. To identify properties that the user can configure for predefined storage classes, see the following information on specific storage classes.

When you create a storage class by using the Custom Storage Class Designer, you can choose whether the user of the storage class can specify additional settings. See “Allow Users of Storage Class to Specify Property Value” on page 40-18.

Default Storage Class

By default, the storage class for an individual data element is `Auto`. By using this default value, you are choosing to have the code generator handle storage class selection and application. In this case, the data element is subject to code generation optimizations, which can eliminate the element from the code or change the representation of the element. If optimizations do not eliminate the data element, the element appears as a field of a standard data structure in the generated code.

When using the Code Mappings editor or the code mappings API, a data element configured with storage class `Auto` that is not eliminated by optimizations acquires the default code generation settings for the corresponding data category. If you do not specify a code configuration for a category of data elements, the storage class is `Default`. When a data element category is using this storage class, data elements in that category appear as a field of a standard data structure in the generated code.

Built-in and Predefined Storage Classes

To prevent optimizations from eliminating storage for a data element, you can choose a storage class for the element based on your code generation requirements. Available storage classes include built-in storage classes, predefined storage classes in the `Simulink` package, and might also include other project-specific storage classes defined in an Embedded Coder Dictionary. If you have special

requirements that are not met by the listed storage classes and you are generating code for an ERT-based target, you can define and use a new storage class. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2 and “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.

For Simulink Coder, you can choose from these built-in and predefined storage classes.

Requirements	Storage Class
Enable optimizations, potentially generating more efficient code.	“Auto” on page 36-77 (Individual data elements)
For data elements that cannot be optimized away, represent data as a field of a standard data structure.	“Default” on page 36-77 (Default mapping)
Prevent optimizations from eliminating storage for a data element.	“Model Default” on page 36-77 (Individual mapping)
Access data from a standalone global variable. Generated code contains the variable declaration and definition.	“ExportedGlobal” on page 36-78
Access data from a standalone global variable. Generated code contains the variable declaration. Your external code provides the definition.	“ImportedExtern, ImportedExternPointer” on page 36-78

If you have Embedded Coder, you can choose from these additional predefined storage classes available in the `Simulink` package.

Requirements	Storage Class
Prevent optimizations from eliminating storage for a data element.	“Dictionary Default” on page 36-77 (Default mapping)
Generate standalone variables.	• “ExportToFile” on page 36-80 • “ImportFromFile” on page 36-82 • “FileScope” on page 36-81 • “Localizable” on page 36-82 • “Reusable” on page 36-83 • “Const, Volatile, and ConstVolatile” on page 36-79
Generate variables as fields of a structure.	• “Struct” on page 36-83 • “Bitfield” on page 36-79
Generate variables that have a value that is fixed during code compilation.	• “Define, ImportedDefine” on page 36-80 • “CompilerFlag” on page 36-79
Generate variables that you access through a function call.	“GetSet” on page 36-81
Generate variables for single-instance data and generate structures for multi-instance data.	“MultiInstance” on page 36-83

These storage classes are examples of storage classes that you can add to an Embedded Coder Dictionary. These examples are defined and made available when you prepare a model for code generation by using the Quick Start tool.

Requirements	Storage Class
Generate global structures that contain parameter data and signal or state data.	“ParamStruct, SignalStruct” on page 36-84

Note Signal objects defined in the base workspace or a data dictionary with a non-auto storage class are not supported for models that are configured with an ERT-based system target file and service code interface.

Auto

When you select this storage class, the code generator handles how to represent the data in the generated code and can enable optimizations to operate on the data element, potentially generating more efficient code.

`Auto` is the default storage class setting for each data element in a model. The data element is subject to code generation optimizations, which can eliminate the element from the code or change the representation of the element. For information about these optimizations, such as those on the **Configuration Parameters** > **Code Generation** > **Optimization** pane, see “How Generated Code Stores Internal Signal, State, and Parameter Data”.

Optimizations cannot eliminate some data, such as most block states, from the code. The remaining data acquires a default storage class that you specify on the **Data Defaults** tab of the Code Mappings editor (see “Configure Default Code Generation for Data” on page 33-16). If a data element cannot be eliminated, the name of the element in the code is based on naming rules that you specify by using model configuration parameters. See “Identifier Format Control” on page 55-17—requires Embedded Coder.

Default

On the **Data Defaults** tab of the Code Mappings editor, `Default` is the default storage class setting for each data element category. If you leave the storage class setting for a category at this value, data elements that are not subject to code generation optimizations appear as a field of a standard data structure. See “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49.

Model Default

When you configure an individual data element for code generation, use the `Model Default` storage class to prevent optimizations from eliminating storage for a data element. See “How Generated Code Stores Internal Signal, State, and Parameter Data”. With this storage class setting, the data element acquires the default storage class that you specify for the corresponding data category on the **Data Defaults** tab of the Code Mappings editor. The name of the data element in the code is the same as the name in the model.

Dictionary Default

In Embedded Coder, if you link a model to a data dictionary, which includes a coder dictionary that configures default code definitions for categories of data, you can use the Code Mappings editor to apply the dictionary defaults. On the **Data Defaults** tab, select a category, then set the storage class

to `Dictionary Default`. For more information on configuring a default code mapping in a coder dictionary, see “Configure Default Code Mapping in a Shared Dictionary” on page 32-16.

ExportedGlobal

Use this built-in storage class to generate a global variable definition and declaration. By default, the name of the variable is the name of the data element. The code declares the variable in the generated file *`model`*`.h`, which you can include (`#include`) in your external code. The code generator does not optimize this global variable from the generated code.

When you use this storage class, you can also configure this property.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.

For an example that uses this storage class, see “C Code Generation Configuration for Model Interface Elements”.

ImportedExtern, ImportedExternPointer

To make a data element in a model represent a global variable that your external code defines, use the built-in storage class `ImportedExtern`. The generated algorithmic code uses the variable without defining it.

Generate code that reads from and writes to a global variable defined by your external code. The generated code declares the variable in the generated file *`model`*`_private.h` so that the model entry-point functions can read and write to the variable.

For example, you can apply the storage class `ImportedExtern` to a signal line, block state, or parameter object. For imported data:

- The generated code does not initialize parameter data. Your code must initialize imported parameter data.
- The generated initialization functions dynamically initialize some signal and state data. Unlike data that the generated code allocates, the code does not initialize imported signal or state data to a stored value of zero. Instead, the code immediately initializes the data to the real-world nonzero value that you specify in Simulink.

Note The code generator initializes zero values that you specify in an Initialize Function block, even if those values have imported scope.

When your external code defines a data element and provides a pointer for accessing that data, use `ImportedExternPointer`. The generated code reads from and writes to that pointer. The generated code declares the variable in the generated file *`model`*`_private.h` and reads and writes to the data by dereferencing the pointer.

When you use these storage classes, you can also configure this property.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.

Bitfield

Use this storage class to generate a structure that stores Boolean data in named bit fields.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`StructName`	Name of the structure for the data element in the generated code.

For an example that uses this storage class, see “Bitfields” on page 26-74.

CompilerFlag

Use this storage class to support preprocessor conditionals defined by using a compiler flag or option.

If you build the generated code by using Embedded Coder, to specify the compiler option, you can use the model configuration parameter **Configuration Parameters > Code Generation > Custom Code > Code information > Defines**. See Code Generation Pane: Custom Code: Additional Build Information: Defines.

When you use this storage class, you can also configure this property.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.

For an example that uses this storage class, see “Generate Preprocessor Conditionals for Variant Systems” on page 27-32.

Const, Volatile, and ConstVolatile

Use these storage classes to generate a global variable definition and declaration with the `const`, `volatile`, or `const` and `volatile` type qualifiers.

When you use these storage classes, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`HeaderFile`	Source header file that contains declarations for global data read by the data element and external code.

Property	Description
`DefinitionFile`	Source definition file that contains definitions for global data read by the data element and external code.
`Owner`	Owner of global data that is defined with the code generated for one of multiple components that use that data.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

For an example that uses this storage class, see “Type Qualifiers” on page 26-12.

Define, ImportedDefine

Use the `Define` storage class to generate a macro (`#define` directive) such as `#define myParam 5`.

Use `ImportedDefine` to generate code that uses a macro (`#define` directive) defined in a header file in your external code. For an example, see “Macro Definitions (#define)” on page 26-56.

When you use these storage classes, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`HeaderFile`	Source header file that contains declarations for global data that is read by the data element and external code.

For an example that uses this storage class, see “Macro Definitions (#define)” on page 26-56.

ExportToFile

Generate a global variable definition and declaration to an external file. You can specify the name and placement of the files that define and declare the variable. See “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`HeaderFile`	Source header file where the code generator places the declarations for global data read by the data element and external code.

Property	Description
`DefinitionFile`	Source definition file where the code generator places the definitions for global data read by the data element and external code. The code generator honors this setting only for single-instance models. The setting is ignored for reusable multi-instance models because the data definition is handled at the parent level.
`Owner`	A component in the model hierarchy where the code generator places a global data definition instead of placing it in the top component of the hierarchy. Requires that you set the model configuration parameter **Use owner from data object for data definition placement**.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

For an example that uses this storage class, see “Definition, Initialization, and Declaration of Parameter Data” on page 26-5.

FileScope

Use this storage class to generate a global variable definition and declaration that has the `static` type qualifier. In the generated code, the scope of the variable is limited to the current file, which is typically *`model`*`.c`.

In a model reference hierarchy, if a referenced model uses a parameter object (such as `Simulink.Parameter`) that you create in the base workspace or a data dictionary, you cannot apply `FileScope` to the object. As a workaround, move the parameter object into the model workspace of the referenced model. Then, you can use `FileScope`.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

GetSet

Use this storage class to generate code that interacts with data by calling custom accessor functions. Your external code defines the data and provides the function definitions.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`HeaderFile`	Source header file that contains declarations for the `get` and `set` functions.
`GetFunction`	Data element that appears in the generated code as a call to a specified `get` function.
`SetFunction`	Data element that appears in the generated code as a call to a specified `set` function.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

For an example that uses this storage class, see "Access Data Through Functions with Storage Class GetSet" on page 40-25.

ImportFromFile

Use this storage class to generate code that reads from and writes to a global variable defined by your external code. `ImportFromFile` is similar to `ExportToFile`, but the generated code does not define the variable.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`HeaderFile`	Source header file that contains declarations for global data that is read by the data element and external code. To avoid linker errors, you must add include guards to the beginning of the header file.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

For an example that uses this storage class, see "Integrate External Application Code with Code Generated from PID Controller" on page 40-45.

Localizable

For signals, if possible, minimize the use of global storage by using local variables.

- If the variable is used in a single function, the variable is local to that function.
- If the variable is used in a single file, the variable is local to that file.
- If the variable is used in more than one function or file, the variable is a global.

Generating local variables prevents the code generator from removing the variables from the generated code.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`PreserveDimensions`	When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array.

For an example that uses this storage class, see “Generate Local Variables with Localizable Storage Class” on page 40-62.

Reusable

Use this storage class to reuse the same variable for multiple independent signals in a model. The code generator stores intermediate calculations of a data path (a series of connected blocks) in a single, reused global variable.

When you use this storage class, you can also configure this property.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.

For an example that uses this storage class, see “Specify Buffer Reuse for Signals in a Path” on page 78-17.

Struct

Use this storage class to generate a global structure with a name that you can specify.

When you use this storage class, you can also configure these properties.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.
`StructName`	Name of the structure for the data element in the generated code.

For an example that uses this storage class, see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67 and “Structures of Signals” on page 26-67.

MultiInstance

To generate unstructured variables for single-instance data and structures for multi-instance data, use this storage class. When you apply this storage class to a data item, the Embedded Coder Dictionary determines if it is a single-instance storage class or a multi-instance storage class. The dictionary determines the type of class by the type of data and by the context of the model within the model reference hierarchy. You can duplicate this storage class to create a copy that you can edit.

When you apply this storage class to an individual data element, you can also configure the **Identifier** property. The **Identifier** property is an identifier string that the code generator uses to name the data element in the generated code.

For an example that uses this storage class, see “Flexible Storage Class for Different Model Hierarchy Contexts” on page 32-34.

ParamStruct, SignalStruct

Use these storage classes to generate global structures that contain parameter and signal or state data, respectively. In a hierarchy of components (referenced models or atomic subsystems), you can use these storage classes to create a corresponding hierarchy of structures. These storage classes appear in the Code Mappings editor only after preparing a model for code generation by using the Quick Start tool.

These storage classes are available only for model-owned data.

When you use these storage classes, you can also configure this property.

Property	Description
`Identifier`	Identifier string that the code generator uses to name the data element in the generated code.

Storage Class Limitations

- When you use storage classes in the Code Mappings editor (Embedded Coder) or code mappings API, some limitations apply. See “Limitations” on page 33-11.
- Data objects cannot use an Embedded Coder storage class and a multiword data type.
- For Embedded Coder storage classes in models that use referenced models:
  - If you apply a grouped storage class, such as `Struct` or `Bitfield`, to multiple data items, you must set the storage class **Data scope** property to `Imported` and you must provide the data declaration in an external header file. Grouped storage classes use a single variable in the generated code to represent multiple data objects.
  - If a parameter object exists in the base workspace or a data dictionary, and a referenced model uses the object, you cannot apply the storage class `FileScope`. To apply this storage class to the parameter object, move the object into the model workspace of the referenced model.
- You cannot apply the storage class `FileScope` to data items used by a data exchange interface (C API, external mode, or ASAP2) or MAT-file logging. File-scoped data is not externally accessible.
- You cannot apply the storage class `FileScope` to data that is used in multiple files.

See Also

Related Examples

- “C Code Generation Configuration for Model Interface Elements”
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

- “Control Data and Function Interface in Generated Code”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Use Enumerated Data in Generated Code

In this section...

"Enumerated Data Types" on page 36-86
"Specify Integer Data Type for Enumeration" on page 36-86
"Customize Enumerated Data Type" on page 36-87
"Control Enumerated Type Implementation in Generated Code" on page 36-91
"Type Casting for Enumerations" on page 36-92
"Enumerated Type Limitations" on page 36-92

Enumerated Data Types

Enumerated data is data that is restricted to a finite set of values. An *enumerated data type* is a MATLAB class that defines a set of *enumerated values*. Each enumerated value consists of an *enumerated name* and an *underlying integer* which the software uses internally and in generated code. The following is a MATLAB class definition for an enumerated data type named `BasicColors`, which is used in the examples in this section.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
end
```

For basic information about enumerated data types and their use in Simulink models, see "Use Enumerated Data in Simulink Models". For information about enumerated data types in Stateflow charts, see "Define Enumerated Data Types" (Stateflow).

Specify Integer Data Type for Enumeration

When you specify a data type for your enumeration, you can:

- Control the size of enumerated data types in the generated code by specifying a superclass.
- Reduce RAM/ROM usage.
- Improve code portability.
- Improve integration with legacy code.

You can specify these integer data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `Simulink.IntEnumType`. Specify values in the range of the signed integer for your hardware platform.

Use a Class Definition in a MATLAB File

To specify an integer data type size, derive your enumeration class from the integer data type.

```
classdef Colors < int8
   enumeration
     Red(0)
     Green(1)
     Blue(2)
   end
end
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green ((Colors)1)
#define Blue     ((Colors)2)
```

Use the Function Simulink.defineIntEnumType

To specify an integer data type size, specify the name-value pair `StorageType` as the integer data type.

```
Simulink.defineIntEnumType('Colors',{'Red','Green','Blue'},...
[0;1;2],'StorageType','int8')
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green ((Colors)1)
#define Blue     ((Colors)2)
```

Customize Enumerated Data Type

When you generate code from a model that uses enumerated data, you can implement these static methods to customize the behavior of the type during simulation and in generated code:

- `getDefaultValue` — Specifies the default value of the enumerated data type.
- `getDescription` — Specifies a description of the enumerated data type.
- `getHeaderFile` — Specifies a header file where the type is defined for generated code.
- `getDataScope` — Specifies whether generated code exports or imports the enumerated data type definition to or from a separate header file.
- `addClassNameToEnumNames` — Specifies whether the class name becomes a prefix in generated code.

The first of these methods, `getDefaultValue`, is relevant to both simulation and code generation, and is described in “Specify a Default Enumerated Value”. The other methods are relevant only to code generation. To customize the behavior of an enumerated type, include a version of the method in the `methods(Static)` section of the enumeration class definition. If you do not want to customize the type, omit the `methods(Static)` section. The table summarizes the methods and the data to supply for each one.

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value
`getDefaultValue`	Specifies the default enumeration member for the class.	First member specified in the enumeration definition	A character vector containing the name of an enumeration member in the class (see “Instantiate Enumerations”).
`getDescription`	Specifies a description of the enumeration class.	`''`	A character vector containing the description of the type.
`getHeaderFile`	Specifies the name of a header file. The method `getDataScope` determines the significance of the file.	`''`	A character vector containing the name of the header file that defines the enumerated type. By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify the custom return value as `'<myTypes.h>'`.
`getDataScope`	Specifies whether generated code exports or imports the definition of the enumerated data type. Use the method `getHeaderFile` to specify the generated or included header file that defines the type.	`'Auto'`	One of: `'Auto'`, `'Exported'`, or `'Imported'`.
`addClassNameToEnumNames`	Specifies whether to prefix the class name in generated code.	`false`	`true` or `false`.

Specify a Description

To specify a description for an enumerated data type, include this method in the `methods(Static)` section of the enumeration class:

```
function retVal = getDescription()
% GETDESCRIPTION  Optional description of the data type.
  retVal = 'description';
end
```

Substitute a MATLAB character vector for *`description`*. The generated code that defines the enumerated type includes the specified description.

Import Type Definition in Generated Code

To prevent generated code from defining an enumerated data type, which allows you to provide the definition in an external file, include these methods in the `methods(Static)` section of the enumeration class:

```
function retVal = getHeaderFile()
  % GETHEADERFILE Specifies the file that defines this type in generated code.
  % The method getDataScope determines the significance of the specified file.
  retVal = 'imported_enum_type.h';
end

function retVal = getDataScope()
  % GETDATASCOPE Specifies whether generated code imports or exports this type.
  % Return one of:
  % 'Auto':     define type in model_types.h, or import if header file specified
  % 'Exported': define type in a generated header file
  % 'Imported': import type definition from specified header file
  % If you do not define this method, DataScope is 'Auto' by default.
  retVal = 'Imported';
end
```

Instead of defining the type in *`model`*`_types.h`, which is the default behavior, generated code imports the definition from the specified header file using a `#include` statement like:

```
#include "imported_enum_type.h"
```

Generating code does not create the imported header file. You must provide the header file, using the file name specified by the method `getHeaderFile`, that defines the enumerated data type.

To create a Simulink enumeration that corresponds to your existing C-code enumeration, use the `Simulink.importExternalCTypes` function.

Export Type Definition in Generated Code

To generate a separate header file that defines an enumerated data type, include these methods in the `methods(Static)` section of the enumeration class:

```
function retVal = getDataScope()
  % GETDATASCOPE Specifies whether generated code imports or exports this type.
  % Return one of:
  % 'Auto':     define type in model_types.h, or import if header file specified
  % 'Exported': define type in a generated header file
  % 'Imported': import type definition from specified header file
  % If you do not define this method, DataScope is 'Auto' by default.
  retVal = 'Exported';
end

function retVal = getHeaderFile()
  % GETHEADERFILE Specifies the file that defines this type in generated code.
  % The method getDataScope determines the significance of the specified file.
  retVal = 'exported_enum_type.h';
end
```

Generated code exports the enumerated type definition to the generated header file `exported_enum_type.h`.

Add Prefixes To Class Names

By default, enumerated values in generated code have the same names that they have in the enumeration class definition. Alternatively, your code can prefix every enumerated value in an enumeration class with the name of the class. You can use this technique to prevent identifier conflicts or to improve the readability of the code. To specify class name prefixing, include this method in the `methods(Static)` section of an enumeration class:

```
    function retVal = addClassNameToEnumNames()
      % ADDCLASSNAMETOENUMNAMES Specifies whether to add the class name
      % as a prefix to enumeration member names in generated code.
      % Return true or false.
      % If you do not define this method, no prefix is added.
      retVal = true;
    end
```

Specify the return value as `true` to enable class name prefixing or as `false` to suppress prefixing. If you specify `true`, each enumerated value in the class appears in generated code as *`EnumTypeName_EnumName`*. For the example enumeration class `BasicColors` in "Enumerated Data Types" on page 36-86, the data type definition in generated code might look like this:

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef enum {
  BasicColors_Red = 0,            /* Default value */
  BasicColors_Yellow = 1,
  BasicColors_Blue = 2,
} BasicColors;

#endif
```

The enumeration class name `BasicColors` appears as a prefix for each of the enumerated names.

Control Use of Duplicate Enumeration Member Names

When you import the enumeration data from a header file, you can control the use of duplicate enumeration member names during code generation. Duplicate enumeration member names improve the readability of the code. Use the model configuration parameter **Duplicate enumeration member names** to allow duplicate enumeration member names in different enumeration types during code generation or to generate an error or a warning message. You can use duplicate enumeration member names only if two enumerations have the same `StorageType` and have these specifications:

- `DataScope` set to `'Imported'`
- `StorageType` set to `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, or `'uint32'`
- `Value` is the same

For example:

```
typedef int32_T enum {
  Red = 0,
  Yellow = 1,
  Blue = 2,
}A;

typedef int32_T enum {
  Black = 0,
  Yellow = 1,
  White = 2,
}B;
```

You can have a `Yellow` enumeration member in enumeration `A` and `B` without prefixing the member name with a class name to improve the readability of your code.

Control Enumerated Type Implementation in Generated Code

Suppose that you define an enumerated type `BasicColors`. You can specify that the generated code implement the type definition by using:

- An `enum` block. The native integer type of your hardware is the underlying integer type for the enumeration members.
- A `typedef` statement and a series of `#define` macros. The `typedef` statement bases the enumerated type name on a specific integer data type, such as `int8`. The macros associate the enumeration members with the underlying integer values.

Implement Enumerated Type by Using enum Block

To implement the type definition by using an `enum` block:

- In Simulink, define the enumerated type by using a `classdef` block in a script file. Derive the enumeration from the type `Simulink.IntEnumType`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Do not specify the property `StorageType`.

When you generate code, the type definition appears in an `enum` block.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef enum {
  Red = 0,            /* Default value */
  Yellow,
  Blue,
} BasicColors;

#endif
```

Implement Enumerated Type Using a Specific Integer Type

To implement the type definition using a `typedef` statement and `#define` macros:

- In Simulink, define the enumerated type using a `classdef` block in a script file. Derive the enumeration from a specific integer type such as `int8`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Specify the property `StorageType` using a specific integer type such as `int8`.

When you generate code, the type definition appears as a `typedef` statement and a series of `#define` macros.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef int8_T BasicColors;

#define Red ((BasicColors)0)            /* Default value */
#define Yellow ((BasicColors)1)
#define Blue ((BasicColors)2)

#endif
```

By default, the generated file *model*_types.h contains enumerated type definitions.

Type Casting for Enumerations

Safe Casting

A Simulink Data Type Conversion block accepts a signal of integer type. The block converts the input to one of the underlying values of an enumerated type.

If the input value does not match an underlying value of an enumerated type value, Simulink inserts a safe cast to replace the input value with the enumerated type default value.

Enable and Disable Safe Casting

You can enable or disable safe casting for enumerations during code generation for a Simulink Data Type Conversion block or a Stateflow block.

To control safe casting, enable or disable the **Saturate on integer overflow** block parameter. The parameter works as follows:

- **Enabled:** Simulink replaces a nonmatching input value with the default value of the enumerated values during simulation. The software generates a safe cast function during code generation.
- **Disabled:** For a nonmatching input value, Simulink generates an error during simulation. The software omits the safe cast function during code generation. In this case, the code is more efficient. However, the code may be more vulnerable to run-time errors.

Safe Cast Function in Generated Code

This example shows how the safe cast function `int32_T ET08_safe_cast_to_BasicColors` for the enumeration `BasicColors` appears in generated code when generated for 32-bit hardware.

```
static int32_T ET08_safe_cast_to_BasicColors(int32_T input)
{
    int32_T output;
    /* Initialize output value to default value for BasicColors (Red) */
    output = 0;
    if ((input >= 0) && (input <= 2)) {
    /* Set output value to input value if it is a member of BasicColors */
        output = input;
    }
    return output;
}
```

Through this function, the enumerated type's default value is used if the input value does not match one of underlying values of the enumerated type's values.

If the block's **Saturate on integer overflow** parameter is disabled, this function does not appear in generated code.

Enumerated Type Limitations

- Generated code does not support logging enumerated data.
- For `uint32`-based enumerations, enumeration values must be less than or equal to `intmax('int32')`.

See Also

`enumeration` | `Simulink.defineIntEnumType` | `Simulink.data.getEnumTypeInfo`

Related Examples

- “Use Enumerated Data in Simulink Models”
- “Simulink Enumerations”
- “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30

Data Stores in Generated Code

In this section...

"About Data Stores" on page 36-94
"Generate Code for Data Store Memory Blocks" on page 36-94
"Storage Classes for Data Store Memory Blocks" on page 36-95
"Data Store Buffering in Generated Code" on page 36-96
"Data Stores Shared by Instances of a Reusable Model" on page 36-99
"Structures in Generated Code Using Data Stores" on page 36-99

About Data Stores

A data store contains data that is accessible in a model hierarchy at or below the level in which the data store is defined. Data stores can allow subsystems and referenced models to share data without having to use I/O ports to pass the data from level to level. See "Data Stores with Data Store Memory Blocks" for information about data stores in Simulink. This section provides additional information about data store code generation.

Generate Code for Data Store Memory Blocks

To control the code generated for a Data Store Memory block, apply a storage class to the data store. You can associate a Data Store Memory block with a signal object that you store in a workspace or data dictionary, and control code generation for the block by applying the storage class to the object:

1. On the **Modeling** tab, click **Model Data Editor**.
2. In the Model Data Editor, select the **Data Stores** tab.
3. Begin editing the name of the target Data Store Memory block by clicking the corresponding row in the **Name** column.
4. Next to the name, click the button ⋮ and select **Create and Resolve**.
5. In the Create New Data dialog box, set **Value** to `Simulink.Signal`. Optionally, use the **Location** drop-down list to choose a workspace for storing the resulting `Simulink.Signal` object.
6. Click **Create**. The `Simulink.Signal` object, which has the same name as the data store, appears in the target workspace. Simulink selects the block parameter **Data store name must resolve to Simulink signal object**.

   When the property dialog box for the object opens, click **OK**.
7. In the **Simulink Coder** or **Embedded Coder** app, open the Code Mappings editor. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.
8. On the **Data Stores** tab, apply the target storage class.

Note When a Data Store Memory block is associated with a signal object, the mapping between the **Data store name** and the signal object name must be one-to-one. If two or more identically named entities map to the same signal object, the name conflict is flagged as an error at code generation time. See "Resolve Conflicts in Configuration of Signal Objects for Code Generation" for more information.

Storage Classes for Data Store Memory Blocks

You can control how Data Store Memory blocks in your model are stored and represented in the generated code by assigning storage classes and type qualifiers. You do this in almost exactly the same way you assign storage classes and type qualifiers for block states.

Data Store Memory blocks, like block states, have `Auto` storage class by default, and their memory is stored within the `DWork` vector. The symbolic name of the storage location is based on the data store name.

You can generate code from multiple Data Store Memory blocks that have the same data store name, subject to the following restriction: *at most one* of the identically named blocks can have a storage class other than `Auto`. An error is reported if this condition is not met.

For blocks with `Auto` storage class, the code generator produces a unique symbolic name for each block to avoid name clashes. For Data Store Memory blocks with storage classes other than `Auto`, the generated code uses the data store name as the symbol.

In the following model, a Data Store Write block writes to memory declared by the Data Store Memory block `myData`:

To control the storage declaration for a Data Store Memory block, in the coder app, use the Code Mappings editor. On the **Data Stores** tab, select a **Storage Class** for the block.

Data Store Memory blocks are nonvirtual because code is generated for their initialization in `.c` and `.cpp` files and their declarations in header files. The following table shows how the code generated for the Data Store Memory block in the preceding model differs for different storage classes. The table gives the variable declarations and `MdlOutputs` code generated for the `myData` block.

Storage Class	Declaration	Code

`Auto` or `Model default` (when Code Mapping Editor specifies `Default` storage class)	In *`model`*`.h`
	`typedef struct`
	`D_Work_tag`
	`{`
	`  real_T myData;`
	`}`
	`D_Work;`
	
	In *`model`*`.c` or *`model`*`.cpp`
	`/* Block states (auto storage) */`
	`D_Work` *`model`*`_DWork;`
		`model``_DWork.myData =`
		`  rtb_SineWave;`

Storage Class	Declaration	Code

`ExportedGlobal`	In *`model`*`.c` or *`model`*`.cpp`
	`/* Exported block states */`
	`real_T myData;`
	In *`model`*`.h`
	`extern real_T myData;`
		`myData = rtb_SineWave;`

`ImportedExtern`	In *`model`*`_private.h` `extern real_T myData;`	`myData = rtb_SineWave;`
`ImportedExternPointer`	In *`model`*`_private.h` `extern real_T *myData;`	`(*myData) = rtb_SineWave;`

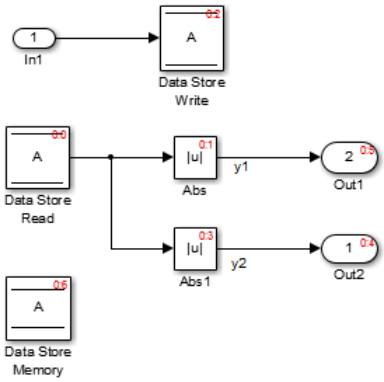
For information about applying storage classes, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

For ERT models, you can preserve dimensions of multidimensional arrays in data stores. For more information, see “Preserve Dimensions of Multidimensional Arrays in Generated Code” on page 47-2.

Data Store Buffering in Generated Code

A Data Store Read block is a nonvirtual block that copies the value of the data store to its output buffer when it executes. Since the value is buffered, downstream blocks connected to the output of the data store read utilize the same value, even if a Data Store Write block updates the data store in between execution of two of the downstream blocks.

The next figure shows a model that uses blocks whose priorities have been modified to achieve a particular order of execution:



```
/* local block i/o variables */
real_T rtb_DataStoreRead;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A;
```

Buffer the value of A

```
/* Abs: '<Root>/Abs1' incorporates:
 *  DataStoreRead: '<Root>/Data Store Read'
 */
y1 = fabs(A);
```

Use A (whose value equals the buffered value at this point

```
/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 *  Inport: '<Root>/In1'
 */
A = u1;
```

Update the value of A

```
/* Abs: '<Root>/Abs' */
y2 = fabs(rtb_DataStoreRead);
```

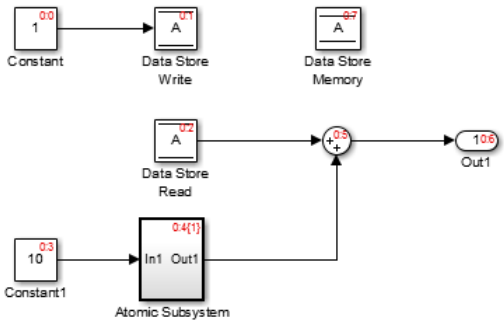
Consistently use the same buffered value as before the update to A

The following execution order applies:

1. The block Data Store Read buffers the current value of the data store A at its output.
2. The block Abs1 uses the buffered output of Data Store Read.
3. The block Data Store Write updates the data store.
4. The block Abs uses the buffered output of Data Store Read.

Because the output of Data Store Read is a buffer, both Abs and Abs1 use the same value: the value of the data store at the time that Data Store Read executes.

The next figure shows another example:



```
real_T rtb_DataStoreRead;

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 *  Constant: '<Root>/Constant'
 */
A = DoBufferDSMRead2_P.Constant_Value;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A;

/* Outputs for atomic SubSystem: '<Root>/Atomic Subsystem' */
DoBufferDSMRead_AtomicSubsystem();

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

/* Outport: '<Root>/Out1' incorporates:
 *  Sum: '<Root>/Sum'
 */
DoBufferDSMRead2_Y.Out1 = rtb_DataStoreRead + DoBufferDSMRead2_B.Abs;
```

Buffer the value of A

We don't do a global analysis to detect if this function writes to A

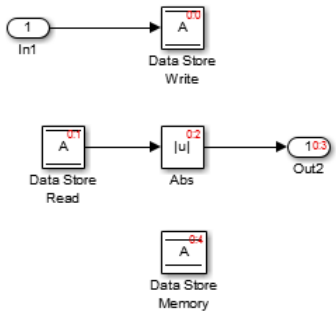
Use the buffered value of A

In this example, the following execution order applies:

1. The block Data Store Read buffers the current value of the data store A at its output.
2. Atomic Subsystem executes.
3. The Sum block adds the output of Atomic Subsystem to the output of Data Store Read.

Simulink assumes that Atomic Subsystem might update the data store, so Simulink buffers the data store. Atomic Subsystem executes after Data Store Read buffers its output, and the buffer provides a way for the Sum block to use the value of the data store as it was when Data Store Read executed.

In some cases, the code generator determines that it can optimize away the output buffer for a Data Store Read block, and the generated code refers to the data store directly, rather than a buffered value of it. The next figure shows an example:




```
/* Model step function */
void DONTbufferDSMRead_step(void)
{
  /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
   *  Inport: '<Root>/In1'
   */
  A = u1;

  /* Abs: '<Root>/Abs' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   */
  y2 = fabs(A);
}
```

In the generated code, the argument of the `fabs()` function is the data store `A` rather than a buffered value.

Data Stores Shared by Instances of a Reusable Model

You can use a data store to share a piece of data between the instances of a reusable referenced model (see "Share Data Among Referenced Model Instances") or a model that you configure to generate reentrant code (by setting the configuration parameter **Code interface packaging** to `Reusable function`). If you implement the data store as a Data Store Memory block and select the **Share across model instances** parameter:

- By default, the data store appears in the generated code as a separate global symbol.
- If you have Embedded Coder, to restrict access such that only the code generated from the model can use the data store, configure the data store to appear as `static` by applying the storage class `FileScope`. For more information about `FileScope` and other storage classes, see "Choose Storage Class for Controlling Data Representation in Generated Code" on page 36-75.

Structures in Generated Code Using Data Stores

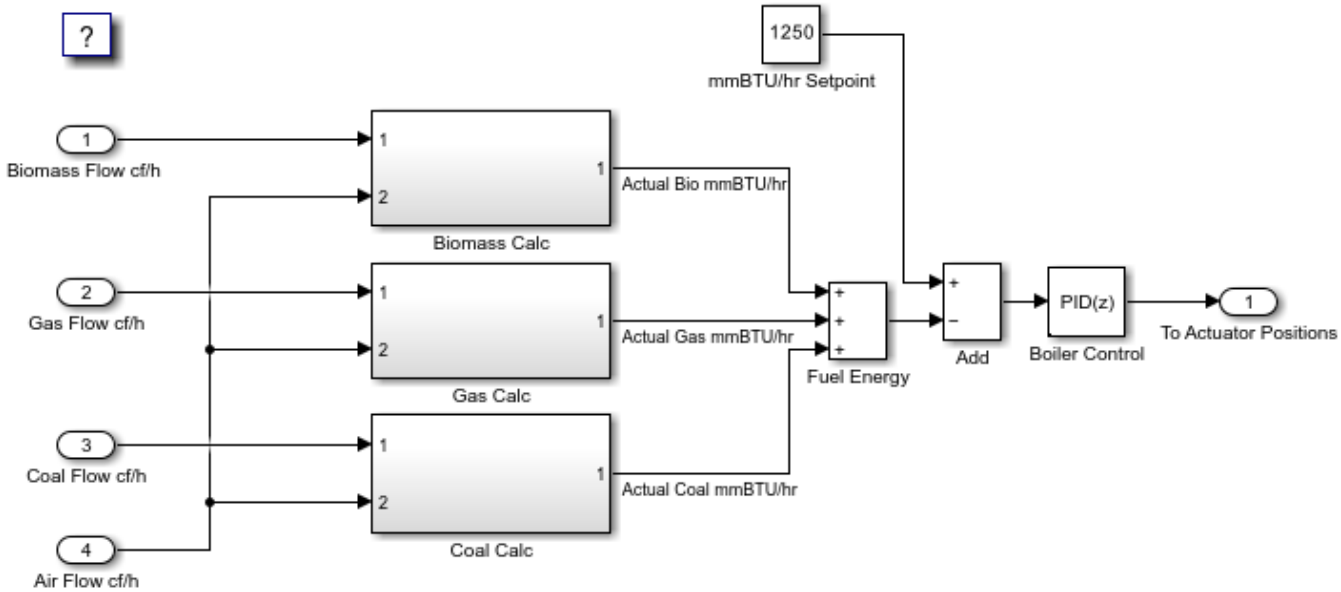
If you use more than one data store to provide global access to multiple signals in generated code, you can combine the signals into a single structure variable by using one data store. This combination of signal data can help you integrate the code generated from a model with other existing code that requires the data in a structure format.

This example shows how to store several model signals in a structure in generated code using a single data store. To store multiple signals in a data store, you configure the data store to accept a composite signal, such as a nonvirtual bus signal or an array of nonvirtual bus signals.

Explore Example Model

Open the example model `BusStructInCode`.

```
open_system('BusStructInCode')
```



Copyright 2023 The MathWorks, Inc.

The model contains three subsystems that perform calculations on the inputs from the top level of the model. In each subsystem, a Data Store Memory block stores an intermediate calculated signal.

Generate code with the model. In the code generation report, view the file `BusStructInCode.c`. The code defines a global variable for each data store.

```
real_T BioBTURate;
real_T CoalBTURate;
real_T GasBTURate;
```

Suppose that you want to integrate code generated from the example model with other existing code. Suppose also that the existing code requires access to data from the three data stores in a single structure variable. You can use a data store to assemble the target data in a structure in generated code.

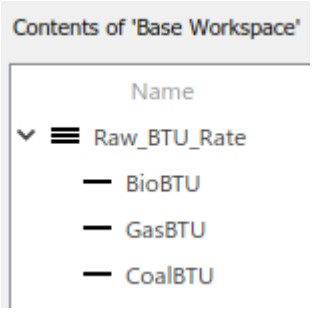
Configure Data Store

Configure a data store to contain multiple signals by creating a bus type to use as the data type of the data store. Define the bus type using the same hierarchy of elements as the structure that you want to appear in generated code.

1. Open the Type Editor tool.

   ```
   typeeditor
   ```

2. Define a new bus type `Raw_BTU_Rate` with one element for each of the three target signals. Name the elements `BioBTU`, `GasBTU`, and `CoalBTU`.

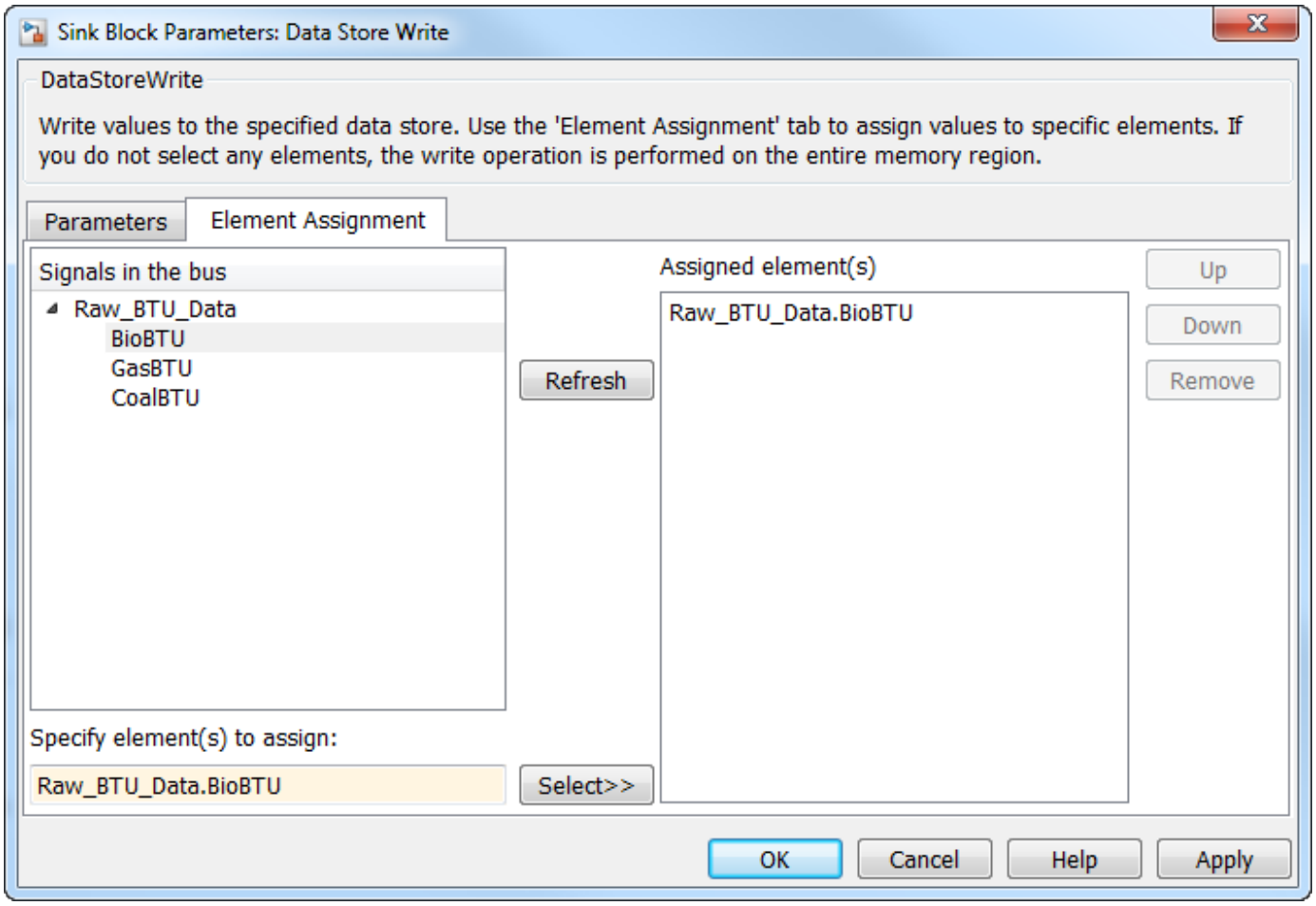


3. At the top level of the example model, add a Data Store Memory block.
4. On the **Modeling** tab, click **Model Data Editor**.
5. In the Model Data Editor, inspect the **Data Stores** tab.
6. For the new Data Store Memory block, use the **Name** column to set the data store name to `Raw_BTU_Data`.
7. Use the **Data Type** column to set the data type of the data store to `Bus: Raw_BTU_Rate`.
8. In the Code Mappings editor, on the **Data Stores** tab, apply the storage class `ExportedGlobal` to `Raw_BTU_Data`.

Write to Data Store Elements

To write to a specific element of a data store, use a Data Store Write block. On the **Element Assignment** tab in the dialog box, you can specify to write to a single element, a collection of elements, or the entire contents of a data store.

1. Open the **Biomass Calc** subsystem.
2. Delete the Data Store Memory block `BioBTURate`.
3. In the block dialog box for the Data Store Write block, set **Data store name** to `Raw_BTU_Data`.
4. On the **Element Assignment** tab, under **Signals in the bus**, expand the contents of the data store `Raw_BTU_Data`. Click the element `BioBTU`, and then click **Select**. Click **OK**.



5 Modify the **Gas Calc** and **Coal Calc** subsystems similarly.

- Delete the Data Store Memory block in each subsystem.
- In each Data Store Write block dialog box, set **Data store name** to `Raw_BTU_Data`.
- In the **Gas Calc** subsystem, use the Data Store Write block to write to the data store element `GasBTU`. In the **Coal Calc** subsystem, write to the element `CoalBTU`.

Generate Code with Data Store Structure

1 Generate code for the example model.

2 In the code generation report, view the file `BusStructInCode.h`. The code defines a structure that corresponds to the bus type `Raw_BTU_Rate`.

```
typedef struct {
  real_T BioBTU;
  real_T GasBTU;
  real_T CoalBTU;
} Raw_BTU_Rate;
```

3 View the file `BusStructInCode.c`. The code represents the data store with a global variable `Raw_BTU_Data` of the structure type `Raw_BTU_Rate`. In the model step function, the code assigns the data from the calculated signals to the fields of the global variable `Raw_BTU_Data`.

See Also

Related Examples

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Structures in Generated Code Using Data Stores” on page 36-99
- “When to Use a Data Store”
- “Generate Code That Dereferences Data from a Literal Memory Address” on page 71-15

Specify Single-Precision Data Type for Embedded Application

When you want code that uses only single precision, such as when you are targeting a single-precision processor, you can use model configuration parameters and block parameters to prevent the introduction of `double` in the model.

To design and validate a single-precision model, see “Validate a Floating-Point Embedded Model”. If you have Fixed-Point Designer, you can use the Single Precision Converter app (see **Single Precision Converter**).

Use single Data Type as Default for Underspecified Types

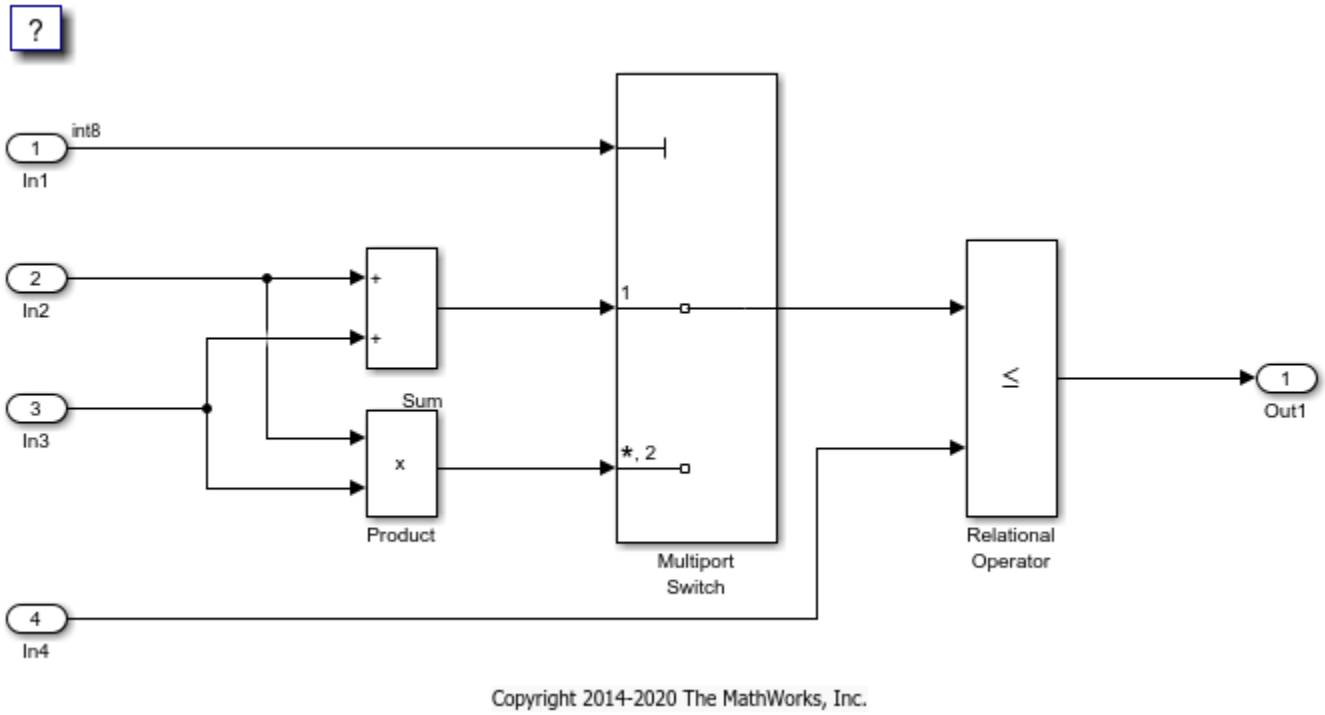
This example shows how to avoid introducing a double-precision data type in code generated for a single-precision hardware target.

If you specify an inherited data type for signals, but data type propagation rules cannot determine data types for the signals, the signal data types default to `double`. You can use a model configuration parameter to specify the default data type as `single`.

Explore Example Model

Open the example model rtwdemo_underspecified_datatype and configure it to show the generated names of blocks.

```
model = 'rtwdemo_underspecified_datatype';
load_system(model)
set_param(model,'HideAutomaticNames','off')
open_system(model);
```



The root inports `In2`, `In3`, and `In4` specify `Inherit: Auto` for the **Data type** block parameter. The downstream blocks also use inherited data types.

Generate Code with `double` as Default Data Type

The model starts with the configuration parameter **System target file** set to `ert.tlc`, which requires Embedded Coder®. Set **System target file** to `grt.tlc` instead.

```
set_param(model,'SystemTargetFile','grt.tlc')
```

Generate code from the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_underspecified_datatype
### Successful completion of build procedure for: rtwdemo_underspecified_datatype

Build Summary

Top model targets built:

Model                            Action                        Rebuild Reason
========================================================================================
rtwdemo_underspecified_datatype  Code generated and compiled.  Code generation information file c

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.896s
```

In the code generation report, view the file `rtwdemo_underspecified_datatype.h`. The code uses the `double` data type to define the variables `In2`, `In3`, and `In4` because the Inport data types are underspecified in the model.

```
cfile = fullfile('rtwdemo_underspecified_datatype_grt_rtw',...
    'rtwdemo_underspecified_datatype.h');
rtwdemodbtype(cfile,...
    '/* External inputs (root inport signals with default storage) */',...
    '/* External outputs (root outports fed by signals with default storage) */',...
    1, 0);
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  int8_T In1;                          /* '<Root>/In1' */
  real_T In2;                          /* '<Root>/In2' */
  real_T In3;                          /* '<Root>/In3' */
  real_T In4;                          /* '<Root>/In4' */
} ExtU_rtwdemo_underspecified_d_T;
```

Generate Code with `single` as Default Data Type

Open the Configuration Parameters dialog box. On the **Math and Data Types** pane, select `single` in the **Default for underspecified data type** drop-down list.

Alternatively, enable the optimization at the command prompt.

```
set_param(model, 'DefaultUnderspecifiedDataType', 'single');
```

Generate code from the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_underspecified_datatype
### Successful completion of build procedure for: rtwdemo_underspecified_datatype

Build Summary

Top model targets built:

Model                            Action                        Rebuild Reason
========================================================================================
rtwdemo_underspecified_datatype  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.621s
```

In the code generation report, view the file `rtwdemo_underspecified_datatype.h`. The code uses the `single` data type to define the variables `In2`, `In3`, and `In4`.

```
rtwdemodbtype(cfile,...
    '/* External inputs (root inport signals with default storage) */',...
    '/* External outputs (root outports fed by signals with default storage) */',...
    1, 0);
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  int8_T In1;                          /* '<Root>/In1' */
  real32_T In2;                        /* '<Root>/In2' */
  real32_T In3;                        /* '<Root>/In3' */
  real32_T In4;                        /* '<Root>/In4' */
} ExtU_rtwdemo_underspecified_d_T;
```

See Also

Related Examples

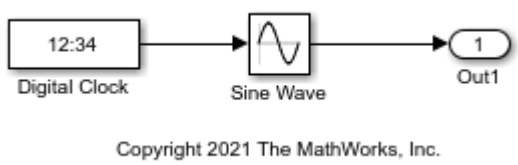
- “Default for underspecified data type”
- “Subnormal Number Execution Speed” on page 76-16
- Language standard
- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “About Data Types in Simulink”

Tune Phase Parameter of Sine Wave Block During Code Execution

Under certain conditions, you cannot configure the **Phase** parameter of a Sine Wave block to appear in the generated code as a tunable global variable (for more information, see the block reference page). This example shows how to generate code so that you can tune the phase during execution.

Create the model `ex_phase_tunable` by using a Digital Clock block.

```
open_system('ex_phase_tunable')
```



Set **Default parameter behavior** to `Tunable` so that the parameters of the Sine Wave block appear in the generated code as tunable fields of the global parameter structure.

```
set_param('ex_phase_tunable','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
slbuild('ex_phase_tunable')
```

```
### Starting build procedure for: ex_phase_tunable
### Successful completion of code generation for: ex_phase_tunable

Build Summary

Top model targets built:

Model              Action           Rebuild Reason
========================================================================================
ex_phase_tunable  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.09s
```

In the code generation report, view the file `ex_phase_tunable.c`. The code algorithm in the model `step` function calculates the Sine Wave block output. The parameters of the block, including **Phase**, appear in the code as tunable structure fields.

```
file = fullfile('ex_phase_tunable_grt_rtw','ex_phase_tunable.c');
rtwdemodbtype(file,'/* Outport: ''<Root>/Out1'' incorporates:',...
    'ex_phase_tunable_P.SineWave_Bias;',1,1)
```

```
  /* Outport: '<Root>/Out1' incorporates:
   *  DigitalClock: '<Root>/Digital Clock'
   *  Sin: '<Root>/Sine Wave'
   */
  ex_phase_tunable_Y.Out1 = sin(ex_phase_tunable_P.SineWave_Freq *
```

```
(((ex_phase_tunable_M->Timing.clockTick1+
   ex_phase_tunable_M->Timing.clockTickH1* 4294967296.0)) * 1.0) +
ex_phase_tunable_P.SineWave_Phase) * ex_phase_tunable_P.SineWave_Amp +
ex_phase_tunable_P.SineWave_Bias;
```

During code execution, you can assign new values to the structure field that corresponds to the **Phase** parameter.

See Also

Related Examples

- “Preserve Variables in Generated Code” on page 36-2

Switch Between Output Waveforms During Code Execution for Waveform Generator Block

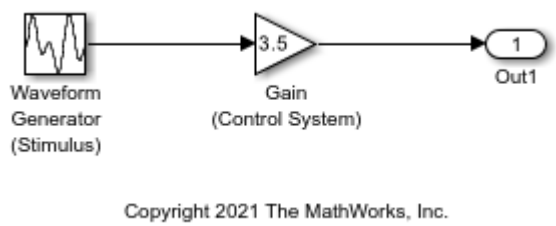
This example shows how to generate code that enables you to switch between stimulus waveforms during code execution.

For a Waveform Generator block, you cannot make the parameters of a waveform, such as amplitude and phase shift, tunable in the generated code. Instead, you can generate code that enables you to choose an active waveform from a set of waveform variants that you specify in the block. During execution of the code, you activate a variant by adjusting the value of a global structure field.

You must set the model configuration parameter **Default parameter behavior** to `Tunable`. Then, by default, block parameters in the model appear tunable in the generated code. These parameters can consume large amounts of memory for a large model.

Create the model `ex_switch_waveform`.

```
open_system('ex_switch_waveform')
```



In the Waveform Generator block, configure this waveform:

```
square(amp,10,0,dutyCycle)
```

In the base workspace, create the variables `amp` and `dutyCycle`.

```
amp = 2.71;
dutyCycle = 50;
```

This waveform represents the baseline stimulus that you want the application to use at the start of execution.

Suppose that, during execution, you want to observe the effects of changing the waveform frequency from 10 to 15 and the phase shift from 0 to 0.5. In the Waveform Generator block, add these waveform variants:

```
square(amp,10,0.5,dutyCycle)
square(amp,15,0,dutyCycle)
square(amp,15,0.5,dutyCycle)
```

Set the model configuration parameter **Default parameter behavior** to `Tunable`.

```
set_param('ex_switch_waveform','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
slbuild('ex_switch_waveform')
```

```
### Starting build procedure for: ex_switch_waveform
### Successful completion of code generation for: ex_switch_waveform

Build Summary

Top model targets built:

Model               Action           Rebuild Reason
========================================================================================
ex_switch_waveform  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.635s
```

The generated file `ex_switch_waveform.h` defines the standard structure type that stores tunable parameter data for the model. The structure contains a field whose value represents the active waveform.

```
file = fullfile('ex_switch_waveform_grt_rtw','ex_switch_waveform.h');
rtwdemodbtype(file,'/* Parameters (default storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (default storage) */
struct P_ex_switch_waveform_T_ {
  real_T WaveformGeneratorStimulus_Selec;
                              /* Mask Parameter: WaveformGeneratorStimulus_Selec
                               * Referenced by: '<S1>/Switch'
                               */
  real_T GainControlSystem_Gain;       /* Expression: 3.5
                                        * Referenced by: '<Root>/Gain (Control System)'
                                        */
};
```

The file `ex_switch_waveform_data.c` defines a global structure variable and initializes the field value to `1`. This value represents the baseline waveform.

```
file = fullfile('ex_switch_waveform_grt_rtw','ex_switch_waveform_data.c');
rtwdemodbtype(file,'/* Block parameters (default storage) */','};',1,1)
```

```
/* Block parameters (default storage) */
P_ex_switch_waveform_T ex_switch_waveform_P = {
  /* Mask Parameter: WaveformGeneratorStimulus_Selec
   * Referenced by: '<S1>/Switch'
   */
  1.0,

  /* Expression: 3.5
   * Referenced by: '<Root>/Gain (Control System)'
   */
  3.5
};
```

The file `ex_switch_waveform.c` defines the model execution function. The function uses a `switch` statement to determine the value of the active waveform, and then calculates the value of the root-level Outport block, `Out1`.

```
file = fullfile('ex_switch_waveform_grt_rtw','ex_switch_waveform.c');
rtwdemodbtype(file,'switch ((int32_T)',...
    'ex_switch_waveform_Y.Out1 =',1,1)
```

```
  switch ((int32_T)ex_switch_waveform_P.WaveformGeneratorStimulus_Selec) {
   case 1:
    temp = temp_tmp - floor(temp_tmp) <= 0.5 ? 2.71 : -2.71;
    break;

   case 2:
    temp = (temp_tmp - 0.79577471545947676) - floor(temp_tmp -
      0.79577471545947676) <= 0.5 ? 2.71 : -2.71;
    break;

   case 3:
    temp = temp - floor(temp) <= 0.5 ? 2.71 : -2.71;
    break;

   default:
    temp = (temp - 1.1936620731892151) - floor(temp - 1.1936620731892151) <= 0.5
      ? 2.71 : -2.71;
    break;
  }

  /* End of MultiPortSwitch: '<S1>/MultiportSwitch' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain (Control System)'
   */
  ex_switch_waveform_Y.Out1 = ex_switch_waveform_P.GainControlSystem_Gain * temp;
```

During code execution, to change the active waveform, adjust the value of the structure field in the global parameters structure.

See Also

Related Examples

- “Preserve Variables in Generated Code” on page 36-2

Create Tunable Calibration Parameter in the Generated Code

A calibration parameter is a value stored in global memory that an algorithm reads for use in calculations but does not write to. Calibration parameters are tunable because you can change the stored value during algorithm execution. You create calibration parameters so that you can:

- Determine an optimal parameter value by tuning the parameter and monitoring signal values during execution.
- Efficiently adapt an algorithm to different execution conditions by overwriting the parameter value stored in memory. For example, you can use the same control algorithm for multiple vehicles of different masses by storing different parameter values in each vehicle's engine control unit.

In Simulink, create a `Simulink.Parameter` object to represent a calibration parameter. You use the parameter object to set block parameter values, such as the **Gain** parameter of a Gain block. To control the representation of the parameter object in the generated code, you apply a storage class to the object.

To make block parameters accessible in the generated code by default, for example for rapid prototyping, set **Default parameter behavior** (see Default parameter behavior) to `Tunable`. For more information, see "Preserve Variables in Generated Code" on page 36-2.

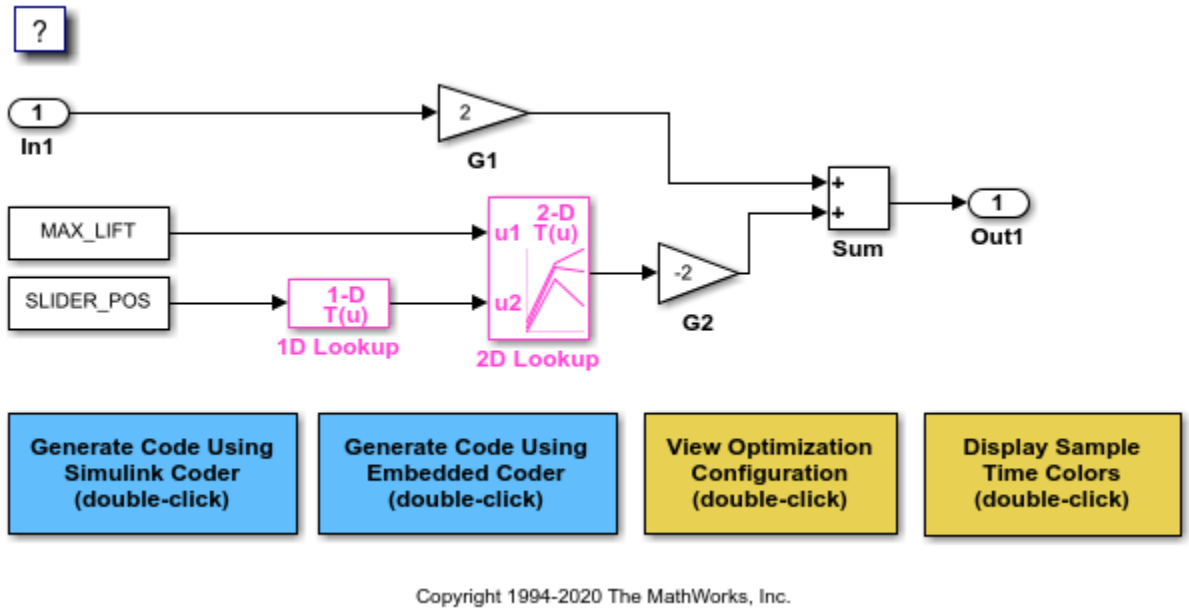
Represent Block Parameter as Tunable Global Variable

This example shows how to create tunable parameter data by representing block parameters as global variables in the generated code.

Configure Block Parameter by Using Parameter Object

Open the example model `rtwdemo_paraminline` and configure it to show the generated names of blocks.

```
load_system('rtwdemo_paraminline')
set_param('rtwdemo_paraminline','HideAutomaticNames','off')
open_system('rtwdemo_paraminline')
```



On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, inspect the **Parameters** tab.

In the model, click the `G1` Gain block. The Model Data Editor highlights the row that corresponds to the **Gain** parameter of the block.

In the Model Data Editor **Value** column, change the gain value from `2` to `myGainParam`.

Next to `myGainParam`, click the action button (with three vertical dots) and select **Create**.

In the Create New Data block dialog box, set **Value** to `Simulink.Parameter(2)`. Click **Create**. A `Simulink.Parameter` object `myGainParam` stores the parameter value, `2`, in the model workspace.

In the myGainParam dialog box, on the **Code Generation** tab, click **Configure in Coder App**.

In the Code Mappings editor, set **Storage Class** of `myGainParam` to `ExportedGlobal`. This storage class causes the parameter object to appear in the generated code as a tunable global variable.

Alternatively, to create the parameter object and configure the model, use these commands at the command prompt:

```
set_param('rtwdemo_paraminline/G1','Gain','myGainParam')
mws = get_param('rtwdemo_paraminline', 'modelworkspace');
mws.assignin('myGainParam',Simulink.Parameter(2));
cm = coder.mapping.utils.create('rtwdemo_paraminline');
setModelParameter(cm,'myGainParam','StorageClass','ExportedGlobal');
```

Use the Model Data Editor to create a parameter object, `myOtherGain`, for the `G2` Gain block. Apply the storage class `ExportedGlobal`.

Alternatively, use these commands at the command prompt:

```
set_param('rtwdemo_paraminline/G2','Gain','myOtherGain')
mws.assignin('myOtherGain',Simulink.Parameter(-2));
setModelParameter(cm,'myOtherGain','StorageClass','ExportedGlobal');
```

Generate and Inspect Code

Generate code from the model.

```
slbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for: rtwdemo_paraminline
### Successful completion of build procedure for: rtwdemo_paraminline

Build Summary

Top model targets built:

Model                Action                       Rebuild Reason
========================================================================================
rtwdemo_paraminline  Code generated and compiled  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.502s
```

The generated file `rtwdemo_paraminline.h` contains `extern` declarations of the global variables `myGainParam` and `myOtherGain`. You can include (`#include`) this header file so that your code can read and write the value of the variable during execution.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.h');
rtwdemodbtype(file,...
    'extern real_T myGainParam;','Referenced by: ''<Root>/G2''',1,1)
```

```
extern real_T myGainParam;             /* Variable: myGainParam
                                        * Referenced by: '<Root>/G1'
                                        */
extern real_T myOtherGain;             /* Variable: myOtherGain
                                        * Referenced by: '<Root>/G2'
```

The file `rtwdemo_paraminline.c` allocates memory for and initializes `myGainParam` and `myOtherGain`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,...
    '/* Exported block parameters */','Referenced by: ''<Root>/G2''',1,1)
```

```
/* Exported block parameters */
real_T myGainParam = 2.0;              /* Variable: myGainParam
                                        * Referenced by: '<Root>/G1'
                                        */
real_T myOtherGain = -2.0;             /* Variable: myOtherGain
                                        * Referenced by: '<Root>/G2'
```

The generated code algorithm in the model `step` function uses `myGainParam` and `myOtherGain` for calculations.

```
rtwdemodbtype(file,...
    '/* Model step function */','/* Model initialize function */',1,0)
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/G1'
   *  Gain: '<Root>/G2'
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = myGainParam * rtwdemo_paraminline_U.In1 +
    myOtherGain * -75.0;
}
```

Apply Storage Class When Block Parameter Refers to Numeric MATLAB Variable

If you use a numeric variable to set the value of a block parameter, you cannot apply a storage class to the variable. As a workaround, you can convert the variable to a parameter object, and then apply a storage class to the object. To convert the variable to a parameter object, choose one of these techniques:

- On the Model Data Editor **Parameters** tab, with **Change view** set to `Code`, find the row that corresponds to the variable. In the **Storage Class** column, from the drop-down list, select `Convert to parameter object`. The Model Data Editor converts the variable to a parameter object. Then, use the **Storage Class** column to apply a storage class to the object.

  You can also use this technique in the Model Explorer.
- Use the Data Object Wizard (see “Create Data Objects for a Model Using Data Object Wizard”). In the Wizard, select the **Parameters** check box. The Wizard converts variables to objects. Then, apply storage classes to the objects, for example, by using the Model Data Editor or the Model Explorer.

Create Storage Class That Represents Calibration Parameters (Embedded Coder)

This example shows how to create a storage class that yields a calibration parameter in the generated code. The storage class causes each parameter object (`Simulink.Parameter`) to appear as a global variable with special decorations such as keywords and pragmas.

In the generated code, the calibration parameters must appear as global variables defined in a file named `calPrms.c` and declared in `calPrms.h`. The variable definitions must look like these definitions:

```
#pragma SEC(CALPRM)

const volatile float param1 = 3.0F;
const volatile float param2 = 5.0F;
const volatile float param3 = 7.0F;

#pragma SEC()
```

The variables use the keywords `const` and `volatile`. The pragma `#pragma SEC(CALPRM)` controls the placement of the variables in memory. To implement the pragma, variable definitions must appear in a contiguous block of code.

Also, the generated code must include an ASAP2 (`a2l`) description of each parameter.

Create Package for Storing Storage Class and Memory Section Definitions

Now, create a package in your current folder by copying the example package `+SimulinkDemos`. The package stores the definitions of `Parameter` and `Signal` classes that you later use to apply the storage class to data elements in models. Later, the package also stores the definitions of the storage class and an associated memory section.

1. Set your current MATLAB folder to a writable location.
2. Copy the `+SimulinkDemos` package folder into your current folder. Name the copy `+myPackage`.

   ```
   copyfile(fullfile(matlabroot,...
       'toolbox','simulink','simdemos','dataclasses','+SimulinkDemos'),...
       '+myPackage','f')
   ```
3. Navigate inside the `+myPackage` folder to the file `Signal.m` to edit the definition of the `Signal` class.
4. Uncomment the `methods` section that defines the method `setupCoderInfo`. In the call to the function `useLocalCustomStorageClasses`, replace `'packageName'` with `'myPackage'`. When you finish, the section looks like this:

   ```
     methods
       function setupCoderInfo(h)
         % Use custom storage classes from this package
         useLocalCustomStorageClasses(h, 'myPackage');
       end
     end % methods
   ```
5. Save and close the file.
6. Navigate inside the `+myPackage` folder to the file `Parameter.m` to edit the definition of the `Parameter` class. Uncomment the `methods` section that defines the method `setupCoderInfo` and replace `'packageName'` with `'myPackage'`.
7. Save and close the file.

Create Storage Class and Memory Section

1. Set your current folder to the folder that contains the package folder `+myPackage`.
2. Open the Custom Storage Class Designer.

   ```
   cscdesigner('myPackage')
   ```
3. In the Custom Storage Class Designer, on the **Memory Sections** tab, click **New**.
4. For the new memory section, set properties according to the table.

Property	Value
Name	`CalMem`
Statements Surround	`Group of variables`
Pre Statement	`#pragma SEC(CALPRM)`
Post Statement	`#pragma SEC()`

Property	Value
Is const	Select
Is volatile	Select

5 Click **Apply**.
6 On the **Custom Storage Class** tab, click **New**.
7 For the new storage class, set properties according to the table.

Property	Value
Name	`CalParam`
For signals	Clear
Data scope	`Exported`
Header File	`calPrms.h`
Definition File	`calPrms.c`
Memory Section	`CalMem`

8 Click **OK**. In response to the message about saving changes, click **Yes**.

Set Default Parameter Object to myPackage.Parameter

To make applying the storage class easier, use the Model Explorer to change the default parameter object from `Simulink.Parameter` to `myPackage.Parameter`.

1 At the command prompt, open the Model Explorer.

```
daexplr
```

2 In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.
3 In the Model Explorer toolbar, click the arrow next to the **Add Simulink Parameter** button. In the drop-down list, select **Customize class lists**.
4 In the **Customize class lists** dialog box, under **Parameter classes**, select the check box next to **myPackage.Parameter**. Click **OK**.
5 In the Model Explorer toolbar, click the arrow next to the **Add Simulink Parameter** button. In the drop-down list, select **myPackage Parameter**.

   A `myPackage.Parameter` object appears in the base workspace. You can delete this object.

Now, when you use tools such as the Model Data Editor to create parameter objects, Simulink creates `myPackage.Parameter` objects instead of `Simulink.Parameter` objects.

Apply Storage Class

In the example model `RollAxisAutopilot`, the `BasicRollMode` subsystem represents a PID controller. Configure the P, I, and D parameters as calibration parameters.

1 Open the model.

```
openExample('RollAxisAutopilot');
```

2 In the model, navigate into the `BasicRollMode` subsystem.
3 In the Apps gallery, click **Embedded Coder**.
4 Underneath the block diagram, open the Model Data Editor by selecting the **Model Data Editor** tab.
5 In the Model Data Editor, select the **Parameters** tab and update the block diagram.

Now, the data table contains rows that correspond to workspace variables used by the Gain blocks (which represent the P, I, and D parameters of the controller).

6. In the Model Data Editor, next to the **Filter contents** box, activate the **Filter using selection** button.
7. In the model, select the three Gain blocks.
8. In the **Filter contents** box, enter `model workspace`.

   The variables that the Gain blocks use are in the model workspace.
9. In the data table, select the three rows and, in the **Storage Class** column for a row, select `Convert to parameter object`.

   The Model Data Editor converts the workspace variables to `myPackage.Parameter` objects. Now, you can apply a storage class to the objects.
10. In the **Storage Class** column for a row, select `CalParam`.

Generate and Inspect Code

1. Generate code from the model.
2. In the code generation report, inspect the `calPrms.c` file. The file defines the calibration parameters.

```
/* Exported data definition */
#pragma SEC(CALPRM)

/* Definition for custom storage class: CalParam */
const volatile real32_T dispGain = 0.75F;
const volatile real32_T intGain = 0.5F;
const volatile real32_T rateGain = 2.0F;

#pragma SEC()
```

   The file `calPrms.h` declares the parameters.
3. Inspect the interface file `RollAxisAutopilot.a2l`. The file contains information about each parameter, for example, for `dispGain`.

```
/begin CHARACTERISTIC
  /* Name                   */      dispGain
  /* Long Identifier        */      ""
  /* Type                   */      VALUE
  /* ECU Address            */      0x0000 /* @ECU_Address@dispGain@ */
  /* Record Layout          */      Scalar_FLOAT32_IEEE
  /* Maximum Difference     */      0
  /* Conversion Method      */      RollAxisAutopilot_CM_single
  /* Lower Limit            */      -3.4E+38
  /* Upper Limit            */      3.4E+38
/end CHARACTERISTIC
```

Initialize Parameter Value From System Constant or Other Macro (Embedded Coder)

You can generate code that initializes a tunable parameter with a value calculated from some system constants (macros). For example, you can generate this code, which initializes a tunable parameter `totalVol` with a value calculated from macros `numVessels` and `vesInitVol`:

```
#define numVessels 16
#define vesInitVol 18.2
```

```
double totalVol = numVessels * vesInitVol;
```

This initialization technique preserves the mathematical relationship between the tunable parameter and the system constants, which can make the generated code more readable and easier to maintain. To generate this code:

1. Create parameter objects that represent the system constants.

   ```
   numVessels = Simulink.Parameter(16);
   vesInitVol = Simulink.Parameter(18.2);
   ```

2. Configure the objects to use the storage class `Define`, which yields a macro in the generated code.

   ```
   numVessels.CoderInfo.StorageClass = 'Custom';
   numVessels.CoderInfo.CustomStorageClass = 'Define';

   vesInitVol.CoderInfo.StorageClass = 'Custom';
   vesInitVol.CoderInfo.CustomStorageClass = 'Define';
   ```

3. Create another parameter object that represents the tunable parameter. Configure the object to use the storage class `ExportedGlobal`, which yields a global variable in the generated code.

   ```
   totalVol = Simulink.Parameter;
   totalVol.CoderInfo.StorageClass = 'ExportedGlobal';
   ```

4. Set the value of `totalVol` by using the expression `numVessels * vesInitVol`. To specify that the generated code preserve the expression, use the `slexpr` function.

   ```
   totalVol.Value = slexpr('numVessels * vesInitVol');
   ```

5. Use `totalVol` to set block parameter values in your model. The code that you generate from the model initializes the tunable parameter with a value based on the system constants.

For more information and limitations about using an expression to set the value of a `Simulink.Parameter` object, see “Set Variable Value by Using a Mathematical Expression”.

Code Generation Impact of Storage Location for Parameter Objects

You can create a parameter object in the base workspace, a model workspace, or a data dictionary. However, when you end your MATLAB session, variables in the base workspace are deleted. To determine where to store parameter objects and other variables that a model uses, see “Determine Where to Store Variables and Objects for Simulink Models”.

The location of a parameter object can impact the file placement of the corresponding data definition in the generated code.

- If you place a parameter object in the base workspace or a data dictionary, the code generator assumes that the corresponding parameter data (for example, a global variable) belongs to the system from which you generate code, not to a specific component in the system. For example, if a model in a model reference hierarchy uses a parameter object with a storage class other than `Auto`, the data definition appears in the code generated for the top model in the hierarchy, not in the code generated for the model that uses the object.

  However, if you have Embedded Coder, some storage classes enable you to specify the name of the model that owns a piece of data. When you specify an owner model, the code generated for that

model defines the data. For more information about data ownership, see “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.

- If you place a parameter object in a model workspace, the code generator assumes that the model owns the data. If you generate code from a reference hierarchy that includes the containing model, the data definition appears in the code generated for the containing model.

  For more information about data ownership, see “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.
- If you apply a storage class other than `Auto` to a parameter object, the object appears in the generated code as a global symbol. Therefore, in a model reference hierarchy, two such objects in different model workspaces or dictionaries cannot have the same name. The name of each object must be unique throughout the model hierarchy.

  However, if you have Embedded Coder, you can use the storage class `FileScope` to prevent name clashes between parameter objects in different model workspaces. See “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.

If you store an `AUTOSAR.Parameter` object in a model workspace, the code generator ignores the storage class that you specify for the object.

Configure Accessibility of Signal Data

When you tune the value of a parameter during algorithm execution, you monitor or capture output signal values to analyze the results of the tuning. To represent signals in the generated code as accessible data, you can use techniques such as test points and storage classes. See “Preserve Variables in Generated Code” on page 36-2.

Programmatic Interfaces for Tuning Parameters

You can configure the generated code to include:

- A C application programming interface (API) for tuning parameters independent of external mode. The generated code includes extra code so that you can write your own code to access parameter values. See “Exchange Data Between Generated and External Code Using C API” on page 62-2.
- A Target Language Compiler API for tuning parameters independently of external mode. See “Parameter Functions”.

Set Tunable Parameter Minimum and Maximum Values

It is a best practice to specify minimum and maximum values for tunable parameters.

You can specify these minimum and maximum values:

- In the block dialog box that uses the parameter object. Use this technique to store the minimum and maximum information in the model.
- By using the properties of a `Simulink.Parameter` object that you use to set the parameter value. Use this technique to store the minimum and maximum information outside the model.

For more information, see “Specify Minimum and Maximum Values for Block Parameters”.

Considerations for Other Modeling Goals

Goal	Considerations and More Information
Apply storage type qualifiers `const` and `volatile`	If you have Embedded Coder, to generate the storage type qualifiers, see “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.
Prevent name clashes between parameters in different components by applying the keyword `static`	If you have Embedded Coder, use the storage class `FileScope` or a similar storage class that you create. See “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.
Generate ASAP2 (`a2l`) description	You can generate an `a2l` file that uses the ASAP2 standard to describe your calibration parameters. For more information, see “Export ASAP2 File for Data Measurement and Calibration”.
Generate AUTOSAR XML (ARXML) description	If you have Embedded Coder, you can generate an ARXML file that describes calibration parameters used by models that you configure for the AUTOSAR standard. See “Model AUTOSAR Calibration Parameters and Lookup Tables” (AUTOSAR Blockset).
Store lookup table data for calibration	To store lookup table data for calibration according to the ASAP2 or AUTOSAR standards (for example, STD_AXIS, COM_AXIS, or CURVE), you can use `Simulink.LookupTable` and `Simulink.Breakpoint` objects in lookup table blocks. However, some limitations apply. See `Simulink.LookupTable`. To work around the limitations of `Simulink.LookupTable` and `Simulink.Breakpoint` objects, use `Simulink.Parameter` objects instead. For more information, see “Define ASAP2 Information for Lookup Tables” on page 63-7 and “Configure Lookup Tables for AUTOSAR Calibration and Measurement” (AUTOSAR Blockset).
Use pragmas to store parameter data in specific memory locations	If you have an Embedded Coder license, to generate code that includes custom pragmas, use storage classes and memory sections. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

See Also

`Simulink.Parameter` | `Simulink.LookupTable` | `Simulink.Breakpoint`

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Reuse Parameter Data in Different Data Type Contexts” on page 36-143
- “Limitations for Block Parameter Tunability in Generated Code”
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “C Code Generation Configuration for Model Interface Elements”
- “Access Structured Data Through a Pointer That External Code Defines” on page 40-10
- “Configure Lookup Tables for AUTOSAR Calibration and Measurement” (AUTOSAR Blockset)

Limitations for Block Parameter Tunability in Generated Code

A block parameter, MATLAB variable, or `Simulink.Parameter` object is tunable if it appears in the generated code as data stored in memory, such as a global variable. For example, when you apply the storage class `ExportedGlobal` to a parameter object, the parameter object appears tunable in the generated code. When you set **Default parameter behavior** to `Tunable`, MATLAB variables and parameter objects appear tunable in the generated code. By definition, model arguments also appear tunable.

Under certain conditions, the code generator cannot maintain tunability of a parameter, variable, object, or expression. In this case, the code generator inlines the numeric value, preventing you from changing the value during code execution.

To detect these conditions in your model, set the model configuration parameter **Detect loss of tunability** (see "Detect loss of tunability") to `warning` or `error`.

Tunable Expression Limitations

You can specify block parameter values as expressions that use `Simulink.Parameter` objects or workspace variables. For example, you can use the expression `5 * gainParam`. For general information about using expressions to set block parameter values, see "Use Mathematical Expressions, MATLAB Functions, and Custom Functions". For limitations with respect to expressions that you use to set the values of `Simulink.Parameter` objects, see "Code Generation of Parameter Objects With Expression Values".

A tunable workspace variable is a `Simulink.Parameter` object or workspace variable that appears tunable in the generated code. For example, an object or variable is tunable if you apply a storage class other than `Auto` or if you set **Default parameter behavior** to `Tunable`.

An expression that contains one or more tunable workspace variables, model arguments, or tunable mask parameters is called a tunable expression. The expression is tunable because the code generator attempts to preserve the expression in the code. Because the code generator preserves the expression, you can change the values of the parameter data during code execution.

The code generator reduces certain expressions to an inlined numeric value in the generated code. The inlining renders workspace variables in the expression nontunable. To avoid loss of tunability due to unsupported expressions, observe these guidelines:

- Expressions involving complex (`i`) workspace variables or parameter objects are not supported.
- Certain operators and functions cause the code generator to reduce expressions and remove tunability. To determine whether an operator or function causes loss of tunability, use the information in this table.

Category	Operators or Functions
1	`+ - .* ./ < > <= >= == ~= & |`
2	`* /`
3	`abs`, `acos`, `asin`, `atan`, `atan2`, `boolean`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `log`, `log10`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `single`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`
4	`: .^ ^ [] {} . \ .\ ' .' , ;`

  - Use operators from category 1 without loss of tunability.
  - Use operators from category 2 in expressions as long as at least one operand is a scalar. For example, scalar/scalar and scalar/matrix operand combinations are supported, but matrix/matrix combinations are not supported.
  - You can use tunable workspace variables as arguments for the functions in category 3. If you use other functions, the code generator removes the tunability of the arguments.
  - The operators in category 4 are not supported.
- The Fcn and If blocks do not support tunable expressions for code generation or in referenced models.
- You can write mask initialization code that creates and modifies variables. If you use those variables in an expression, the expression is not tunable in the generated code. This limitation also applies to Simulink library blocks with mask initialization code. For example, block parameters that specify bit operations for the Bitwise, Bit Set, and Bit Clear blocks are not tunable because the parameter is used in the mask initialization code for the block.
- You can specify a data type for the `Simulink.Parameter` objects or workspace variables that make up expressions. As long as the data type of these variables and objects and the data type of the corresponding block parameters are the same or `double`, the code generator can preserve tunability.

If the code generator preserves tunability of a parameter expression that includes at least one element of an integer type, simulation and execution of generated code can produce results that are numerically inconsistent. Evaluation of the expression in the generated code results in an overflow while the expression saturates during simulation. For more information, see “Numerical Consistency of Model and Generated Code Simulation Results”.

Linear Block Parameter Tunability

These blocks have a `Realization` parameter that affects the tunability of their numeric parameters:

- Transfer Fcn
- State-Space
- Discrete State-Space

To set the `Realization` parameter, you must use the command prompt:

```
set_param(gcb,'Realization','auto')
```

For the `Realization` parameter, you can choose these options:

- `general`: The block's numeric parameters appear tunable in the generated code.
- `sparse`: The generated code represents the block’s parameters as transformed values that increase efficiency. The parameters are not tunable.
- `auto`: The default. If one or more of the block’s parameters are tunable (for example, because you use a tunable parameter object to set a parameter value), then the block uses the `general` realization. Otherwise, the block uses the `sparse` realization.

  To tune the parameter values of one of these blocks during an external mode simulation, the block must use the `general` realization.

Parameter Structures

As described in “Organize Data into Structures in Generated Code” on page 36-147, you can create structures of parameter data in the generated code.

- If a parameter structure is tunable, the numeric fields of the structure are tunable in the generated code. However, if a field contains a nontunable entity, such as a multidimensional array, the structure fields are not tunable.
- You cannot declare individual substructures or fields within a parameter structure as tunable. You cannot use a `Simulink.Parameter` object as the value of a structure field. Instead, you must store the entire structure in the parameter object.

See Also

Related Examples

- “Create Tunable Calibration Parameter in the Generated Code”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Code Generation of Parameter Objects With Expression Values

As described in “Set Variable Value by Using a Mathematical Expression”, you can set the value of a `Simulink.Parameter` object to an expression involving other MATLAB variables or parameter objects. When you generate code from a model that uses such a parameter object, to achieve your goal, use the information in the table.

Goal	Technique and More Information
Generate code without Embedded Coder.	For the parameter object that uses the expression, apply an available storage class (see “C Code Generation Configuration for Model Interface Elements”). For the parameter objects used in the expression, apply only the storage class `Auto`.
Generate code that defines a global variable and initializes it by using an expression involving system constants and other macros (requires Embedded Coder).	See “Initialize Parameter Value From System Constant or Other Macro (Embedded Coder)” on page 36-118.
Generate code that defines a macro whose value is an expression involving other macros (requires Embedded Coder).	For the parameter object that uses the expression, apply a storage class that yields a macro in the generated code, such as `Define` (see “Macro Definitions (#define)” on page 26-56). For the parameter objects used in the expression, apply storage classes that yield imported macros. For example, use the storage class `ImportedDefine`. Your external code must define these macros.

Considerations and Limitations

- To avoid errors that prevent code generation, if you apply a storage class other than `Auto` to the dependent parameter object (which uses the expression as its value), the parameter objects used in the expression must use either `Auto` or a storage class that yields a macro in the generated code.
- You cannot set the value of a parameter object that represents a symbolic dimension (see “Implement Symbolic Dimensions for Array Sizes in Generated Code” on page 27-2) to an expression.
- With Embedded Coder, you can generate code that preserves the expression, but only for the purpose of statically initializing the value of a global variable or macro that corresponds to the dependent parameter object. Follow the guidelines in “Expression Preservation”.

Expression Preservation

If you want Embedded Coder to preserve expressions in the generated code, adhere to these restrictions and guidelines.

- The dependent parameter object must use a storage class other than `Auto`. For example, to generate a global variable and initialize it by using the expression, use `ExportedGlobal` or `ExportToFile`.
- The parameter objects used in the expression must:
  - Use a storage class that yields a macro in the generated code, such as `Define`.
  - Have scalar, real values if the expression uses operators.
- The expression can use only these operators (in MATLAB syntax):
  - Mathematical: `+`, `-`, `*`
  - Relational: `==`, `~=`, `<`, `>`, `<=`, `>=`
- These data typing guidelines apply:
  - If possible, for a parameter object used in an expression, leave the `DataType` property at the default value, `auto`.

    To use a value other than `auto`, you must set the `DataType` property of the object that uses the expression to the same value. Otherwise, the code generator does not preserve the expression.
  - If you want to use a parameter object in multiple different expressions, and the dependent parameter objects have different data types, leave the `DataType` property of the independent object at the default value, `auto`. In the generated code, the value of the macro is expressed as a floating-point number (with effective data type `double`), and the code initializes the dependent parameters by typecasting the result of each expression.

See Also

`Simulink.Parameter`

Related Examples

- “Create Tunable Calibration Parameter in the Generated Code”
- “Share and Reuse Block Parameter Values by Creating Variables”

Configure Packaging of Parameter Arguments in Generated Code

When you create a reusable component as a referenced model, configure block parameters to use the same value for instances of the component or a different value for each instance.

To customize the implementation of parameter arguments in the generated code, you can configure the packaging of these parameters in the model where they are defined.

Configure the default mapping for model parameter arguments by using one of these storage classes:

- `Default` — Each instance of the parameter is allocated a piece of memory in an exported structure whose attributes are defined by identifier naming rules.
- A structured storage class — Each instance of the parameter is allocated a piece of memory as defined by the code definition of that storage class.

Individually specify the storage class for model parameter arguments by using one of these storage classes:

- `Auto` — The parameter appears as an individual model function argument in the generated code. If a block in the model does not reference the parameter, it can be optimized away by the code generator.
- `Model default` — Each instance of the parameter is allocated a piece of memory as defined by the code definition of the default storage class. Specify this default storage class through the model default mapping.

If you use edit-time checking for code generation, you get an error for an invalid storage class.

For a parameter argument with a non-`Auto` storage class:

- On the Model block, you can specify the per-instance value as either a literal, a numeric MATLAB variable, or a `Simulink.Parameter` object.
- If the per-instance value is a `Simulink.Parameter` object, the object must have storage class `Auto`.
- If the per-instance value is a value expression, the evaluated value of the expression is used to statically initialize the parameter.

If you do not specify a default or individual storage class for a parameter argument, the code generator applies an internal storage class, `InstP`, to the parameter. The `Instp` storage class has these properties:

- Exported scope
- Type naming rule is `$R_InstP$M`
- Instance name is `InstP$R$M`

Code Generation Behavior

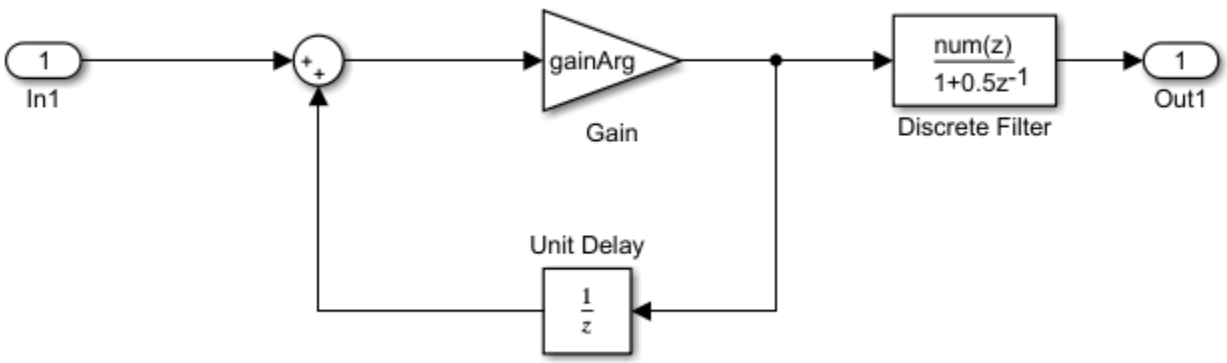
For single-instance referenced models, the parameter arguments reside in a standalone global structure. The top model defines the structure. The referenced model declares and uses the structure.

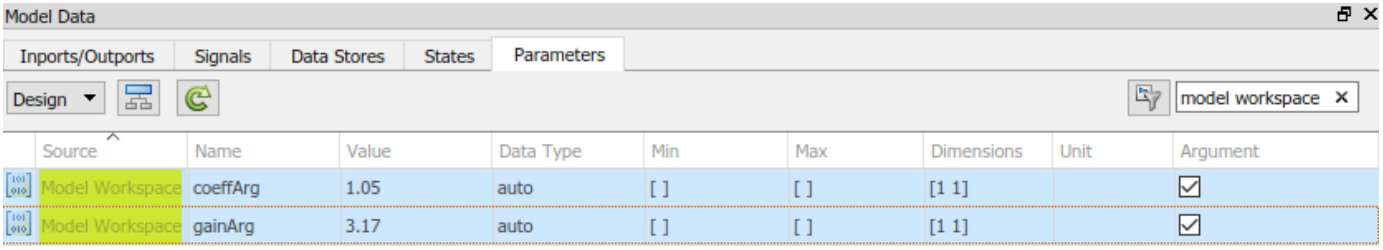
For multi-instance referenced models, the parameter arguments are part of a hierarchically nested structure that the top model declares and defines. The referenced model has access to its parameter values through a pointer in its self structure. The self structure points to a substructure of the structure that the top model defines.

For parameter arguments defined in the top model, if the argument does not have a storage class, it is inlined. If it has a storage class, the resulting code depends on the value of the **Code interface packaging** parameter:

- `Nonreusable function` — The parameter is generated in a structure defined in the top model.
- `Reusable function` — The parameter is generated in a structure passed as part of the first argument of the function.

To understand the behavior of instance-specific parameters in the generated code, consider this referenced model that defines two parameter arguments, `gainArg` and `coeffArg`.





First consider the case where this model is configured to support only a single instance and the default mapping for model parameter arguments does not have a storage class. In the `ex_arg_code_ref.h` file, there is a type definition and a declaration for the instance-specific parameters.

```
/* instance parameters */
extern ex_arg_code_ref_InstP ex_arg_code_refrtInstP;
```

```
/* instance parameters, for model 'ex_arg_code_ref' */
typedef struct {
  real32_T coeffArg;                    /* Variable: coeffArg
                                         * Referenced by: synthesized block
                                         */
  real32_T gainArg;                     /* Variable: gainArg
                                         * Referenced by: synthesized block
                                         */
} ex_arg_code_ref_InstP;
```

When the top model references this model and supplies values for the instance-specific parameters, the code for the top model defines the instance-specific parameters by using the values provided by the top model.

```
/* Define buffer for instance parameters, Block: '<Root>/Model' */
ex_arg_code_ref_InstP ex_arg_code_refrtInstP = {
  1.98F,
  3.98F
};
```

Now consider if the referenced model is configured to support multiple instances. `ex_arg_code_ref.h` contains the same type definition as the single-instance, but the self for the model also contains a pointer to an instance of that data structure. There is no pointer declaration to a standalone global variable of that structure.

```
/* instance parameters, for model 'ex_arg_code_ref' */
typedef struct {
  real32_T coeffArg;                    /* Variable: coeffArg
                                         * Referenced by: synthesized block
                                         */
  real32_T gainArg;                     /* Variable: gainArg
                                         * Referenced by: synthesized block
                                         */
} ex_arg_code_ref_InstP;

/* Real-time Model Data Structure */
struct ex_arg_code_ref_tag_RTM {
  ex_arg_code_ref_InstP *ex_arg_code_ref_InstP_ref;
};
```

The top model declares a parent structure, including an instance of this substructure, and initializes the pointer in the self structure for the referenced model.

```
  /* Assign pointer for instance parameters, Block: '<Root>/Model' */
  ex_arg_code_DW->Model_InstanceData.rtm.ex_arg_code_ref_InstP_ref =
    &ex_arg_code_M->ex_arg_code_InstP_ref->InstP_Model;

  /* Assign pointer for instance parameters, Block: '<Root>/Model1' */
  ex_arg_code_DW->Model1_InstanceData.rtm.ex_arg_code_ref_InstP_ref =
    &ex_arg_code_M->ex_arg_code_InstP_ref->InstP_Model1;
}
```

Limitations

When you configure an instance-specific parameter for code generation, these limitations apply:

- The default mapping for model parameter arguments cannot be reused for model parameters or external parameter objects.
- Instance-specific parameter values must be finite.
- Parameter arguments with a non-`Auto` storage class are not supported for C++ code generation with a C++ class code interface.

See Also

Related Examples

- “Generate Code for Model Reference Hierarchy”
- “Parameterize Instances of a Reusable Referenced Model”

Parameter Data Types in the Generated Code

The data type of a block parameter (such as the **Gain** parameter of a Gain block), numeric MATLAB variable, or `Simulink.Parameter` object determines the data type that the corresponding entity in the generated code uses (for example, a global variable or an argument of a function). To generate more efficient code, you can match parameter data types with signal data types or store parameters in smaller data types.

For basic information about setting block parameter data types in a model, see “Control Block Parameter Data Types”.

Significance of Parameter Data Types

The data type that a block parameter, MATLAB variable, or parameter object uses determines the data type that the generated code uses to store the parameter value in memory. For example:

- If you set the model configuration parameter **Default parameter behavior** (see Default parameter behavior) to `Tunable`, the **Gain** parameter of a Gain block appears in the generated code as a field of a global structure that stores parameter data. If you apply the data type `single` to the block parameter in the model, the structure field in the code uses the corresponding data type, `real32_T`.
- If you apply the storage class `ExportedGlobal` to a `Simulink.Parameter` object, the object appears in the generated code as a separate global variable. If you set the `DataType` property of the object to `int8`, the global variable in the code uses the corresponding data type, `int8_T`.
- If you configure a `Simulink.Parameter` object in a model workspace as a model argument, the object appears in the generated code as a formal parameter (argument) of a model entry-point function, such as the `step` function. The `DataType` property of the object determines the data type of the formal parameter.

Other than determining the data type that the generated code uses to store parameter values in memory, the data type of a parameter, variable, or object can also:

- Cause the block to cast the value of the parameter prior to code generation. The cast can result in overflow, underflow, or quantization.
- Cause the generated code to include extra code, for example saturation code.

Typecasts Due to Parameter Data Type Mismatches

When the data types of block parameters, workspace variables, and signals differ, blocks can use typecasts to reconcile the data type mismatches. These typecasts can cause the generated code algorithm, including the model `step` function, to include explicit casts to reconcile mismatches in storage data type and C bit shifts to reconcile mismatches in fixed-point scaling.

Parameter data type mismatches can occur when:

- The data type that you specify for a MATLAB variable or parameter object (`Simulink.Parameter`) differs from the data type of a block parameter. The block parameter typecasts the value of the variable or object.
- The data type that you specify for an initial value differs from the data type of the initialized signal or state.

- The data type that you specify for a block parameter differs from the data type of the signal or signals that the parameter operates on. Some blocks typecast the parameter to perform the operation. For example, the Gain block performs this typecast.

If you configure a variable or object to use bias or fractional fixed-point slope, the block parameter cannot perform the typecast. In this case, you must match the data type of the variable or parameter object with the data type of the block parameter. Use one of these techniques:

- Use context-sensitive data typing for the variable or parameter object. For a MATLAB variable, use a `double` number to set the value of the variable. For a parameter object, set the `DataType` property to `auto`.
- Use a `Simulink.AliasType` or `Simulink.NumericType` object to set the data type of the block parameter and the data type of a parameter object.

  Use this technique when you cannot rely on context-sensitive data typing, for example, when you use the field of a structure to set the value of the block parameter.
- Manually specify the same data type for the block parameter and the variable or parameter object.

  Use this technique to reduce the dependence of the model on inherited and context-sensitive data types and on external variables and objects.

For blocks that access parameter data through pointer or reference in the generated code, if you specify different data types for the workspace variable and the block parameter, the generated code implicitly casts the data type of the variable to the data type of the block parameter. Note, that an implicit cast requires a data copy which could significantly increase RAM consumption and slow down code execution speed for large data sets. For example, Lookup Table blocks often access large vectors or matrices through pointer or reference in the generated code.

For information about matching parameter data types when you use model arguments, see “Control Data Types of Model Arguments and Argument Values” on page 6-49.

Detect Downcast and Loss of Precision due to Data Type Mismatches

You can configure diagnostic configuration parameters to detect unintentional data type mismatches that result in quantization and loss of parameter precision. See “Model Configuration Parameters: Data Validity Diagnostics”.

Considerations for Other Modeling Patterns

When you use specific modeling patterns and constructs such as fixed-point data types, parameter structures, and lookup table objects, use different techniques to control parameter data types.

- “Tunable Parameters and Best-Precision Fixed-Point Scaling” on page 36-132
- “Control Data Types of Structure Fields” on page 36-133
- “Control Data Types of Lookup Table Objects” on page 36-133
- “Data Typing for Expressions in Parameter Objects” on page 36-134

Tunable Parameters and Best-Precision Fixed-Point Scaling

To apply best-precision fixed-point scaling to a tunable block parameter or parameter object, you can use the Fixed-Point Tool to autoscale an entire system or use the Data Type Assistant to configure individual parameters or objects. See “Calculate Best-Precision Fixed-Point Scaling for Tunable Block Parameters”.

If a tunable parameter uses best-precision fixed-point scaling, Simulink chooses a data type based on the minimum and maximum values that you specify for the parameter (see “Specify Minimum and Maximum Values for Block Parameters”). You can specify these values in the block dialog box that uses the parameter or in the properties of a `Simulink.Parameter` object.

If you do not specify a minimum or maximum, Simulink chooses a data type based on the value of the parameter. The chosen scaling might restrict the range of possible tuning values. Therefore, it is a best practice to specify minimum and maximum values for each tunable parameter.

A tunable parameter can use best-precision scaling even if you do not specify it in the parameter data type. For example, the Gain block can choose a best-precision scaling if the **Parameter data type** in the block dialog box is set to `Inherit: Inherit via internal rule`. This setting is the default for the block.

Control Data Types of Structure Fields

When you use a structure as the value of a block parameter (for example to initialize a bus signal), or when you organize multiple block parameter values into a single structure, you can create a `Simulink.Bus` object to use as the data type of a `Simulink.Parameter` object. You can then control the data types of individual fields in the structure. See “Control Field Data Types and Characteristics by Creating Parameter Object” and “Control Data Types of Initial Condition Structure Fields”.

Control Data Types of Lookup Table Objects

When you use `Simulink.LookupTable` and `Simulink.Breakpoint` objects to store table and breakpoint data for a lookup table block, to control the data types of the table and breakpoint data, use one of these techniques:

- Set the `Value` property of the embedded `Simulink.lookuptable.Table` and `Simulink.lookuptable.Breakpoint` objects by using untyped expressions such as `[1 2 3]`, which returns a `double` vector. If **Breakpoint specification** is set to `Explicit values`, use the MATLAB expression text box and tabular interface at the bottom of the property dialog box. To control the data type, set the `DataType` property to a value other than `auto`.

  Use this technique to separate the value of the table or breakpoint data from the data type, which can improve readability and understanding of your design. You can then use a `Simulink.NumericType` or `Simulink.AliasType` object to:

  - Customize the name of the data type in the generated code.
  - Match the data type of the table or breakpoint data with the data type of a signal in the model.
- Set the `Value` property of the embedded objects by using typed expressions such as `single([1 2 3])`. To use a fixed-point data type, set the `Value` property with an `fi` object. If **Breakpoint specification** is set to `Explicit values`, use the MATLAB expression text box and tabular interface at the bottom of the property dialog box.

  Set the `DataType` property of the embedded objects to the default value, `auto`. The table and breakpoint data then acquire the data type that you use to set the `Value` property.

  Use this technique to store the data type information in the `Value` property, which can simplify the way you interact with the `Simulink.LookupTable` and `Simulink.Breakpoint` objects. You can leave the `DataType` property at the default value.

  When you later change the breakpoint or table data in the `Value` property, preserve the data type information by using a typed expression. Alternatively, if you use a command at the command

prompt or a script to change the data, to avoid using a typed expression, use subscripted assignment, `(:)`.

```
myLUTObject.Table.Value(:) = [4 5 6];
```

When you change the data stored in the `Value` property, if you do not use a typed expression or subscripted assignment, you lose the data type information.

When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override (see "Control Fixed-Point Instrumentation and Data Type Override") only on the subsystem. Instead, set data type override on the entire model.

Data Typing for Expressions in Parameter Objects

You can use an expression to set the value of a parameter object (such as `Simulink.Parameter`). The expression encodes mathematical relationships between different objects. When you use this technique, different data typing rules apply. See "Set Variable Value by Using a Mathematical Expression".

See Also

Related Examples

- "Generate Efficient Code by Specifying Data Types for Block Parameters" on page 36-135
- "Reuse Parameter Data in Different Data Type Contexts" on page 36-143
- "Data Types Supported by Simulink"
- "Control Data Types of Signals"
- "How Generated Code Stores Internal Signal, State, and Parameter Data" on page 36-49

Generate Efficient Code by Specifying Data Types for Block Parameters

To generate more efficient code, match the data types of block parameters (such as the **Gain** parameter of a Gain block) with signal data types. Alternatively, you can store parameters in smaller data types.

Eliminate Unnecessary Typecasts and Shifts by Matching Data Types

These examples show how to generate efficient code by configuring a block parameter to use the same data type as a signal that the block operates on.

Store Data Type Information in Model

Open the example model rtwdemo_configrpinterface and configure it to show the generated names of blocks.

```
open_system('rtwdemo_configrpinterface')
set_param('rtwdemo_configrpinterface','HideAutomaticNames','off')
```

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, select the **Parameters** tab.

In the model, select the Gain block. In the Model Data Editor, the **Data Type** column shows that the data type of the **Gain** parameter of the block is set to `Inherit: Same as input`. With this setting, the **Gain** parameter of this block uses the same data type as the input signal.

In the Code Mappings editor, under **Parameters**, set the storage class for K1 to `ExportedGlobal`.

In the Model Data Editor, set **Change view** to `Design`.

In the data table, for the row that represents `K1`, in the **Data Type** column, select Auto. With this setting, the parameter object acquires its data type from the block parameters that use the object (in this case, the **Gain** block parameter).

Alternatively, to create and configure the object, use these commands at the command prompt:

```
cm = coder.mapping.api.get('rtwdemo_configrpinterface');
setModelParameter(cm,'K1','StorageClass','ExportedGlobal');
hws = get_param(bdroot, 'modelworkspace');
hws.setVariablePart('K1.DataType','Auto');
```

Generate code from the model.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of code generation for: rtwdemo_configrpinterface

Build Summary

Top model targets built:
```

```
Model                       Action           Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 18.782s
```

The generated file rtwdemo_configrpinterface.c defines the global variable K1 by using the data type real_T.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Exported block parameters */','real_T K1 = 2.0;',1,1)
```

```
/* Exported block parameters */
real_T K1 = 2.0;                        /* Variable: K1
```

The generated code algorithm in the model step function uses K1 directly without typecasting.

```
rtwdemodbtype(file,'output =',...
    '10U',1,1)
```

```
    output = K1 * look1_binlc((*input2), rtCP_Table1_bp01Data,
      rtCP_Table1_tableData, 10U);
```

In the Code Mappings editor, under **Parameters**, you can optionally set the storage class for K1 to Inherit: Inherit via internal rule (the default).

In this case, the block parameter chooses the same data type as the input signal, double. However, when you use Inherit: Inherit via internal rule, under other circumstances (for example, when you use fixed-point data types), the block parameter might choose a different data type.

Store Data Type Information in Parameter Object

When you use a Simulink.Parameter object to export or import parameter data from the generated code to your external code, for example by applying the storage class ImportedExtern, you can specify data type information in the parameter object. To match the data type of the parameter object with a signal data type, create a Simulink.NumericType or Simulink.AliasType object. You can strictly control the data type that the parameter object uses in the generated code, eliminating the risk that Simulink® chooses a different data type when you make changes to the model.

At the command prompt, create a Simulink.NumericType object that represents the data type double.

```
myType = Simulink.NumericType;
myType.DataTypeMode = 'Double';
```

Use the Model Data Editor **Data Type** column to set these data types to myType:

- The parameter object. Use the **Parameters** tab.
- The Inport block named In2. Use the **Inports/Outports** tab. Due to data type propagation, the input signal of the Gain block also uses myType.

Use the Model Data Editor **Parameters** tab to set the data type of the **Gain** block parameter to Inherit: Inherit from 'Gain'.

Use this data type object as the data type of the parameter object.

Alternatively, to configure the object and the blocks, use these commands at the command prompt:

```
K1.DataType = 'myType';
set_param('rtwdemo_configrpinterface/In2','OutDataTypeStr','myType')
set_param('rtwdemo_configrpinterface/Gain','ParamDataTypeStr','Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of code generation for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                       Action           Rebuild Reason
============================================================================
rtwdemo_configrpinterface  Code generated.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 18.38s
```

The global variable K1 continues to use the data type real_T.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Exported block parameters */','real_T K1 = 2.0;',1,1)
```

```
/* Exported block parameters */
real_T K1 = 2.0;                      /* Variable: K1
```

Close the code generation report.

```
rtwdemoclean;
```

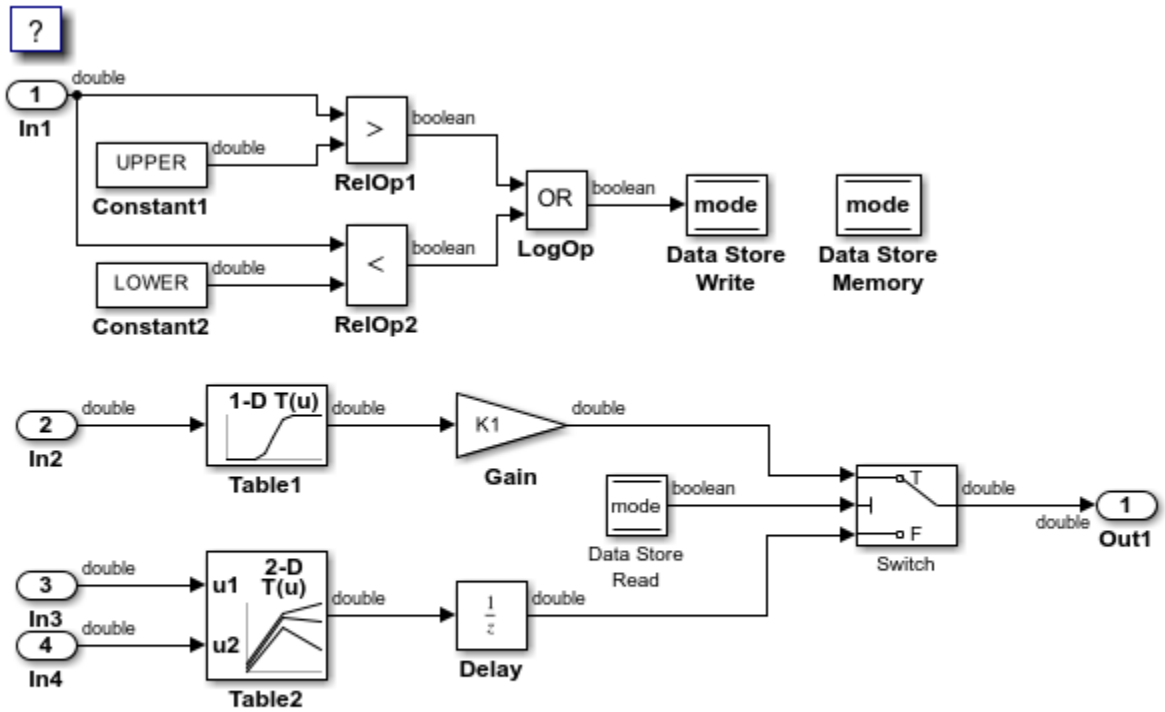
Reduce Memory Consumption by Storing Parameter Value in Small Data Type

When you use a parameter object (for example, `Simulink.Parameter`) to set block parameter values, you can configure the object to appear in the generated code as a tunable global variable. By default, the parameter object and the corresponding global variable typically uses the same data type as the signal or signals on which the block operates. For example, if the input signal of a Gain block uses the data type `int16`, the parameter object typically use the same data type. To reduce the amount of memory that this variable consumes, specify that the variable use a smaller integer data type such as `int8`.

Store Parameter Value in Integer Data Type

Open the model `rtwdemo_configrpinterface` and set the properties of the model.

```
open_system('rtwdemo_configrpinterface')
set_param('rtwdemo_configrpinterface','HideAutomaticNames','off')
set_param('rtwdemo_configrpinterface','ShowPortDataTypes','on')
set_param('rtwdemo_configrpinterface','SimulationCommand','Update')
```



Copyright 1994-2020 The MathWorks, Inc.

Many of the signals in the model use the data type `double`.

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, inspect the **Parameters** tab.

Click the **Show/refresh additional information** button.

Next to the **Filter contents** box, activate the **Filter using selection** button.

In the model, click the Gain block. The Model Data Editor shows one row that corresponds to the **Gain** parameter of the block and one row that corresponds to the MATLAB® variable `K1`, which sets the parameter value to `2`. `K1` resides in the model workspace.

In the C Code tab, select **Code Interface > Default Code Mappings**. In the Code Mappings editor, under **Parameters**, set the storage class for K1 to `ExportedGlobal`.

```
cm = coder.mapping.api.get('rtwdemo_configrpinterface');
setModelParameter(cm,'K1','StorageClass','ExportedGlobal');
```

In the C Code tab, select Code Interface > Default Code Mappings. In the Model Data Editor, set **Change view** to `Design` and ensure that the **Data Type** column for `K1` is `int8`.

```
hws = get_param(bdroot, 'modelworkspace');
hws.setVariablePart('K1.DataType','int8');
```

For the row that represents the **Gain** block parameter, in the **Data Type** column, set the drop-down list to `Inherit: Inherit from 'Gain'`. With this setting, the **Gain** parameter of the block inherits the `int8` data type from the parameter object.

```
set_param('rtwdemo_configrpinterface/Gain','ParamDataTypeStr',...
    'Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
slbuild('rtwdemo_configrpinterface')
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of code generation for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                       Action           Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.264s
```

The generated file `rtwdemo_configrpinterface.c` defines the global variable `K1` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink®.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Exported block parameters */','int8_T K1 = 2;',1,1)
```

```
/* Exported block parameters */
int8_T K1 = 2;                          /* Variable: K1
```

The code algorithm in the model `step` function uses `K1` to calculate the output of the Gain block. The algorithm casts `K1` to the data type `real_T` (`double`) because the signals involved in the calculation use the data type `real_T`.

```
rtwdemodbtype(file,'output =',...
    '10U',1,1)
```

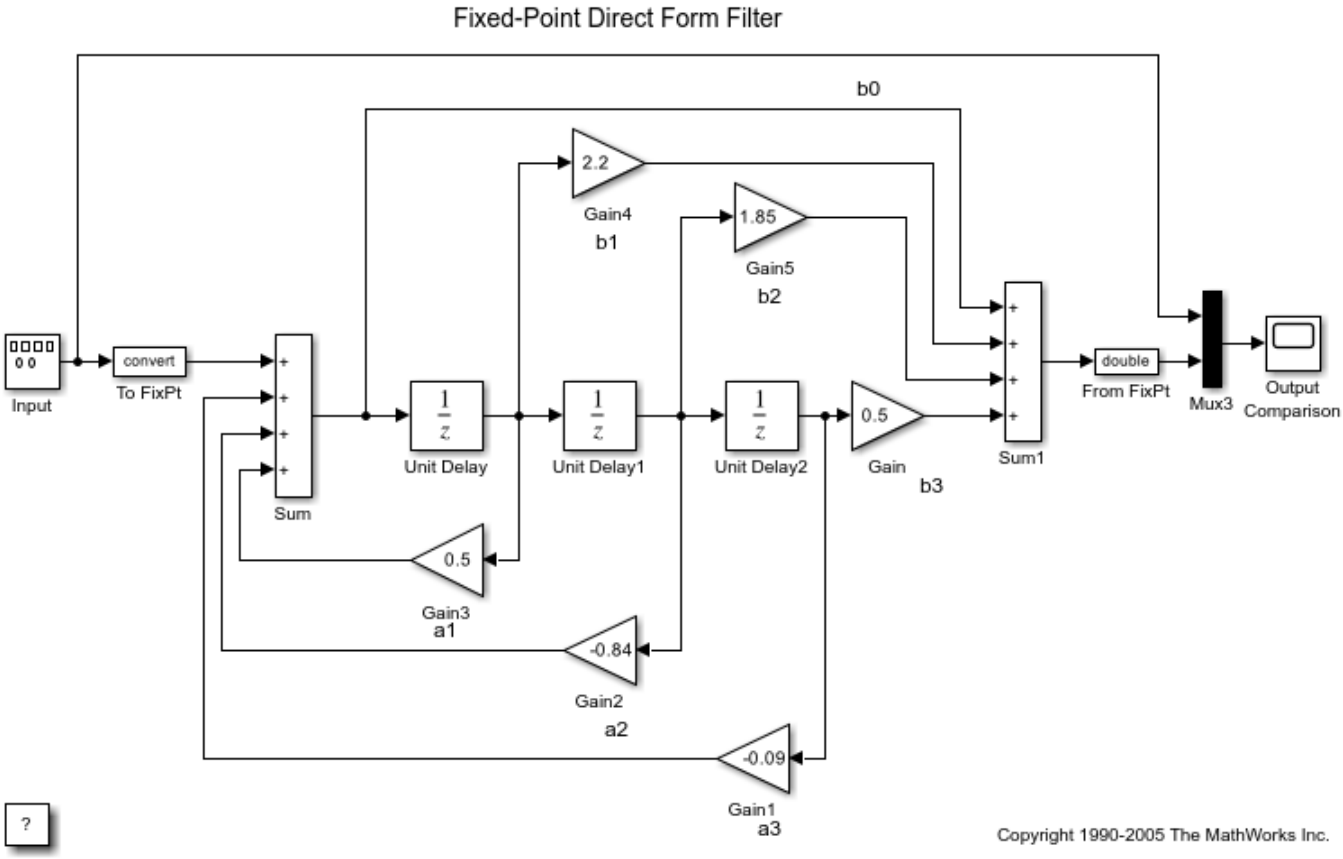
```
    output = (real_T)K1 * look1_binlc((*input2), rtCP_Table1_bp01Data,
      rtCP_Table1_tableData, 10U);
```

Store Fixed-Point Parameter Value in Smaller Integer Data Type

Suppose you configure the signals in your model to use fixed-point data types. You want a gain parameter to appear in the generated code as a tunable global variable. You know the range of real-world values that you expect the parameter to assume (for example, between 0 and 4). If you can meet your application requirements despite reduced precision, you can reduce memory consumption by configuring the parameter to use a different data type than the input and output signals of the block.

Open the example model `fxpdemo_direct_form2` and configure it to show the generated names of blocks.

```
load_system('fxpdemo_direct_form2')
set_param('fxpdemo_direct_form2','HideAutomaticNames','off')
open_system('fxpdemo_direct_form2')
```



Update the block diagram. Signals in this model use signed fixed-point data types with a word length of 16 and binary-point-only scaling.

Open the Gain5 block dialog box. The **Gain** parameter is set to `1.85`. Suppose you want to configure this parameter.

Set **Gain** to `myGainParam` and click **Apply**.

Click the action button (with three vertical dots) next to the parameter value. Select **Create**.

In the Create New Data dialog box, set **Value** to `Simulink.Parameter(1.85)` and click **Create**. The `Simulink.Parameter` object `myGainParam` appears in the base workspace.

Open the **Simulink Coder** app. In the C Code tab, select **Code Interface > Default Code Mappings**. In the Code Mappings editor, under **Parameters**, set the storage class for `myGainParam` to `ExportedGlobal`. With this setting, `myGainParam` appears in the generated code as a global variable.

In the block dialog box, on the **Parameter Attributes** tab, set **Parameter minimum** to `0` and **Parameter maximum** to 4.

Set **Parameter data type** to `fixdt(0,8)` and click **Apply**.

Click the **Show Data Type Assistant** button. The Data Type Assistant shows that the expression `fixdt(0,8)` specifies an unsigned fixed-point data type with a word length of 8 and best-precision scaling. When you simulate or generate code, the block parameter chooses a fraction length (scaling) that enables the data type to represent values between the parameter minimum and maximum (0 and 4) with the best possible precision.

In the Data Type Assistant, set **Scaling** to `Binary point`. Click **Calculate Best-Precision Scaling**, **Fixed-point details**, and **Refresh Details**. The information under **Fixed-point details** shows that a fraction length of 5 can represent the parameter values with a precision of 0.03125.

Set **Scaling** back to `Best precision` and click **OK**. In this example, when you simulate or generate code, the block parameter chooses a fraction length of 5.

Alternatively, you can use these commands at the command prompt to create the object and configure the block:

```
myGainParam = Simulink.Parameter(1.85);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
set_param('fxpdemo_direct_form2/Gain5','Gain','myGainParam')
set_param('fxpdemo_direct_form2/Gain5','ParamMin','0','ParamMax','4')
set_param('fxpdemo_direct_form2/Gain5','ParamDataTypeStr','fixdt(0,8)')
```

Configure the model to produce a code generation report. To reduce clutter in the Command Window, clear the configuration parameter **Verbose build**.

```
set_param('fxpdemo_direct_form2','GenerateReport','on',...
    'LaunchReport','on','RTWVerbose','off')
```

Set hardware settings for the model

```
set_param('fxpdemo_direct_form2','TargetUnknown','off');
set_param('fxpdemo_direct_form2','PortableWordSizes','off');
set_param('fxpdemo_direct_form2','ProdEqTarget','off');
set_param('fxpdemo_direct_form2','TargetHWDeviceType','Custom Processor->MATLAB Host Computer');
```

Generate code from the model.

```
evalc('slbuild(''fxpdemo_direct_form2'')');
```

The generated file `fxpdemo_direct_form2.c` defines the global variable `myGainParam` by using the data type `uint8_T`, which corresponds to the specified word length, 8. The code initializes the variable by using an integer value that, given the fraction length of 5, represents the real-world parameter value 1.85.

```
file = fullfile('fxpdemo_direct_form2_grt_rtw','fxpdemo_direct_form2.c');
rtwdemodbtype(file,'/* Exported block parameters */','uint8_T myGainParam = 59U;',1,1)
```

```
/* Exported block parameters */
uint8_T myGainParam = 59U;             /* Variable: myGainParam
```

The code algorithm uses `myGainParam` to calculate the output of the Gain5 block. The algorithm uses a C shift to scale the result of the calculation.

```
rtwdemodbtype(file,'/* Gain: ''<Root>/Gain5'' incorporates:',...
'/* Gain: ''<Root>/Gain'' incorporates:',1,0)
```

```
  /* Gain: '<Root>/Gain5' incorporates:
```

```
   *  UnitDelay: '<Root>/Unit Delay1'
   */
  fxpdemo_direct_form2_B.Gain5 = (int16_T)((myGainParam *
    fxpdemo_direct_form2_B.UnitDelay1) >> 5);
```

```
rtwdemoclean;
```

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Parameter Data Types in the Generated Code” on page 36-131
- “Reuse Parameter Data in Different Data Type Contexts” on page 36-143

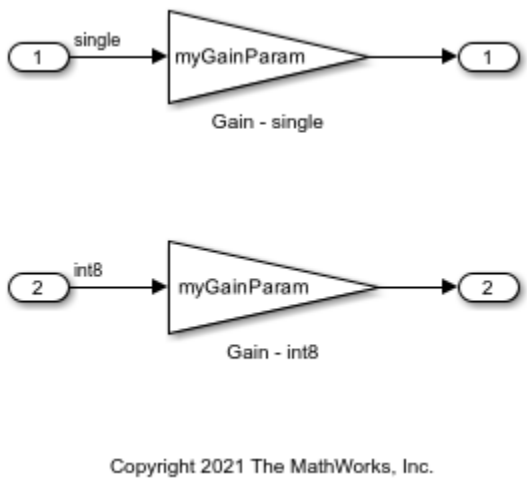
Reuse Parameter Data in Different Data Type Contexts

When you use a `Simulink.Parameter` object or a numeric MATLAB® variable to set two or more block parameter values, if the block parameters have different data types, you must explicitly specify the data type of the object or variable. For example, you cannot leave the data type of the parameter object at the default value, `auto`.

Create and Configure Example Model

Create the model `ex_paramdt_contexts`.

```
ex_paramdt_contexts
```



On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, on the **Inports/Outports** tab, use the **Data Type** column to set the data type of the `In1` Inport block to `single` and the data type of the `In2` block to `int8`.

On the **Signals** tab, set the data types of the Gain block outputs to `Inherit: Same as input`.

On the **Parameters** tab, for the **Gain** parameters of the Gain blocks, set **Data Type** to `Inherit: Same as input`.

For the **Gain** parameters, set **Value** to `myGainParam`.

Alternatively, to configure the blocks, use these commands at the command prompt:

```
set_param('ex_paramdt_contexts/In1','OutDataTypeStr','single')
set_param('ex_paramdt_contexts/In2','OutDataTypeStr','int8')
set_param('ex_paramdt_contexts/Gain - single','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
set_param('ex_paramdt_contexts/Gain - int8','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
```

In the Model Data Editor, for either **Gain** block parameter, click the cell in the **Value** column. Next to `myGainParam`, click the action button (with three vertical dots) and select **Create**.

In the **Create New Data** dialog box, set **Value** to `Simulink.Parameter(3)` and click **Create**. A `Simulink.Parameter` object with value `3` appears in the model workspace.

In the **myGainParam** property dialog box, set **Data type** to `int8`.

On the `Code Generation` tab, click **Configure in Coder App**.

In the Code Mappings editor, set the storage class of `myGainParam` to `ExportedGlobal`. With the storage class `ExportedGlobal`, the object appears in the generated code as a global variable.

Alternatively, to create and configure the parameter object, use these commands at the command prompt:

```
mws = get_param('ex_paramdt_contexts', 'modelworkspace');
mws.assignin('myGainParam',Simulink.Parameter(3));
setVariablePart(mws,'myGainParam.DataType','int8');
cm = coder.mapping.utils.create('ex_paramdt_contexts');
setModelParameter(cm,'myGainParam','StorageClass','ExportedGlobal');
```

In this model, you use the parameter object `myGainParam` to set two block parameter values. The block parameters inherit different data types from the block input signals (`single` or `int8`). To use `myGainParam` in these different data type contexts, you explicitly specify the data type of the parameter object by setting the `DataType` property to `int8`.

Match Parameter Object Data Type with Signal Data Type

Optionally, use a `Simulink.NumericType` or `Simulink.AliasType` object to set the parameter object data type and one of the signal data types. This technique eliminates unnecessary typecasts and shifts in the generated code due to a mismatch between the parameter object data type and the signal data type.

At the command prompt, create a `Simulink.NumericType` object to represent the data type `int8`.

```
sharedType_int8 = fixdt('int8');
```

In the Model Data Editor, on the **Inports/Outports** tab, set the data type of the `In2` Inport block to `sharedType_int8`.

On the **Parameters** tab, update the block diagram. The data table now contains a row that represents the parameter object, `myGainParam`.

Use the **Data Type** column to set the data type of the parameter object to `sharedType_int8`.

Alternatively, to configure the block and the object, use these commands at the command prompt:

```
myGainParam.DataType = 'sharedType_int8';
set_param('ex_paramdt_contexts/In2','OutDataTypeStr','sharedType_int8')
```

The parameter object and the signal use the data type `int8`. To change this data type, adjust the properties of the data type object `sharedType_int8`.

Generate and Inspect Code

Generate code from the model.

```
slbuild('ex_paramdt_contexts')
```

```
### Starting build procedure for: ex_paramdt_contexts
### Successful completion of build procedure for: ex_paramdt_contexts

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
ex_paramdt_contexts  Code generated and compiled.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.763s
```

The generated file `ex_paramdt_contexts.c` defines the global variable `myGainParam` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink®.

```
file = fullfile('ex_paramdt_contexts_grt_rtw','ex_paramdt_contexts.c');
rtwdemodbtype(file,'/* Exported block parameters */','int8_T myGainParam = 3;',1,1)
```

```
/* Exported block parameters */
int8_T myGainParam = 3;                /* Variable: myGainParam
```

The generated code algorithm in the model `step` function uses `myGainParam` to calculate the outputs of the two Gain blocks. In the case of the Gain block whose input signal uses the data type `single`, the code algorithm casts `myGainParam` to the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
rtwdemodbtype(file,'/* Model step function */',...
    '/* Model initialize function */',1,0)
```

```
/* Model step function */
void ex_paramdt_contexts_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain - single'
   *  Inport: '<Root>/In1'
   */
  ex_paramdt_contexts_Y.Out1 = (real32_T)myGainParam * ex_paramdt_contexts_U.In1;

  /* Outport: '<Root>/Out2' incorporates:
   *  Gain: '<Root>/Gain - int8'
   *  Inport: '<Root>/In2'
   */
  ex_paramdt_contexts_Y.Out2 = (int8_T)(myGainParam * ex_paramdt_contexts_U.In2);
}
```

See Also

Related Examples

- “Parameter Data Types in the Generated Code” on page 36-131

- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 36-135
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112

Organize Data into Structures in Generated Code

In C code, you use structures (`struct`) to store data in contiguous locations in memory. You can organize data, such as signals and states, by using meaningful names. Each structure acts as a namespace, so you can reuse a name to designate multiple data items. Like arrays, with structures, you can write code that efficiently transfers and operates on large amounts of data by using pointers.

By default, the code generator aggregates data into standard structures (see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49). To control the characteristics of these structures or to override this default behavior by creating different structures, use the information in the table.

Goal	Technique
Control the characteristics of the standard data structures. For example, specify the names of the structure types, structure variables, and field names.	With Embedded Coder, see “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2.
Control the placement of the structures in memory, for example, by inserting pragmas or other code decorations.	With Embedded Coder, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Goal	Technique
Aggregate data into structures whose names and other basic characteristics you can control. As you add blocks and signals to the model, the resulting new data appears in these structures by default.	With Embedded Coder, apply a structured storage class to a category of data by using the Code Mapping Editor. • When you generate nonreetrant, single-instance code by setting the model configuration parameter **Code interface packaging** to `Nonreusable function`, create flat, global structure variables by applying the built-in storage class `Struct`. Alternatively, use example storage classes created by Quick Start tool, `ParamStruct` and `SignalStruct`, or create and apply your own structured storage class. • When you generate multi-instance (reentrant) code from a model or a component, for example by setting **Code interface packaging** to a value other than `Nonreusable function`, in the Code Mappings editor, you cannot use the built-in storage class `Struct` or a structured storage class that you create in a package. Instead, create your own structured storage class by using an Embedded Coder Dictionary. If you use the Embedded Coder Quick Start tool to prepare the model for code generation (see "Generate Code by Using the Quick Start Tool" on page 53-9), you can use example storage classes `ParamStruct` and `SignalStruct`. • You cannot use this technique to aggregate global data stores or global parameters (which include parameter objects that you store in the base workspace or a data dictionary). For these kinds of data, use other techniques to create structures. For more information about the Code Mapping Editor, see "Configure Default C Code Generation for Categories of Data Elements and Functions" on page 33-15. To create your own storage class, see "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2.

Goal	Technique
Improve readability of the generated code by organizing individual data items into a custom structure whose characteristics you can finely control.	Create a `Simulink.Bus` object that represents the structure type that you want. Use the bus object to set the data types of nonvirtual bus signals and parameter structures in your model. • For an example that shows how to organize parameter data, see “Structures of Parameters” on page 36-152. • For an example that shows how to organize signal data, see “Structures of Signals” on page 36-154. To create an array of structures, see “Arrays of Structures” on page 36-160.
Exchange structured data between generated code and your external code, for example, when the external code already defines a custom structure type and a corresponding global variable.	Create a `Simulink.Bus` object that represents the structure type that you want. If your external code already defines the type, use the `Simulink.importExternalCTypes` function to generate the bus object. Use the bus object to set the data types of nonvirtual bus signals and parameter structures in your model. For an example, see “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30.

Goal	Technique

Reduce the number of arguments (formal parameters) for generated functions.	• To reduce the number of arguments for the entry-point functions of the model from which you generate code, such as *model*_step, see "Reduce Number of Arguments by Using Structures" on page 36-47.
	• To reduce the number of arguments for the entry-point functions of a referenced model, in the referenced model:
	  • Configure root-level Inport and Outport blocks to appear in the code as structure fields. Replace multiple such blocks with a single block that you configure as a nonvirtual bus signal. With this technique, you have the greatest control over the structure characteristics. As you add blocks to the model, you must manually modify the bus definitions. See "Nonvirtual Buses and Parameter Structures" on page 36-152.
	  • Organize model arguments into a custom structure (see "Specify Instance-Specific Parameter Values for Reusable Referenced Model" on page 6-27).
	• To reduce the number of arguments for the entry-point functions of a masked, reentrant subsystem, aggregate mask parameters into a custom structure.

Organize lookup table data into a structure.	Use `Simulink.LookupTable` and `Simulink.Breakpoint` objects. See `Simulink.LookupTable`.
Generate bit fields.	See "Bitfields" on page 26-74 and "Optimize Generated Code by Packing Boolean Data into Bitfields" on page 80-11.

Techniques to Create Structures

To create structures in the generated code, you can use these techniques:

- Apply a structured storage class to categories of data by using the Code Mapping Editor. As you add blocks and signals to a model, new data elements acquire this storage class by default.
- Apply a structured storage class, such as the built-in storage class `Struct`, directly to individual data items by using the Model Data Editor.
- Create custom nonvirtual buses and parameter structures.

To decide which techniques to use, use the information in the table.

Capability	Default Application of Structured Storage Class	Direct Application of Structured Storage Class	Nonvirtual Buses and Parameter Structures
Aggregate new data items into a structure by default	Yes	No	No
Prevent elimination of the target data by optimizations (specify that the data appear in the generated code)	No	Yes	Only if you directly apply a storage class such as `ExportedGlobal` to the bus or structure
Aggregate data into a structure without changing the appearance of the block diagram	Yes	Yes	No
Place signal, state, and parameter data in the same structure	No	Yes	No
Include state data in a structure	Yes	Yes	No
Organize structures into nested structures	No	No	Yes
Organize structures into an array of structures	No	No	Yes
Control name of structure type	Yes	Yes, but the type name derives from the variable name, which you specify	Yes
Create a structure to reduce the number of arguments in a generated function	Yes, but you must create your own storage class by using an Embedded Coder Dictionary	No	Yes
Requires Embedded Coder	Yes	Yes	No

Default Application of Structured Storage Class

You can apply a default storage class to a category of model data. As you add blocks and signals to the model, the associated data acquire the default storage class that you specify. To aggregate new data into structures by default, you can apply structured storage classes. You must use example storage classes `ParamStruct` and `SignalStruct` created by the Quick Start tool or create your own storage class by using an Embedded Coder Dictionary.

You cannot use default application to aggregate global data stores or global parameters (parameter objects that you store in the base workspace or a data dictionary).

To apply default storage classes, use the Code Mapping Editor. In a model window, in the Apps gallery, click **Embedded Coder**. Then, under **Code Mappings** > **Data Defaults**, apply storage classes by using the **Storage Class** column.

For more information about the Code Mapping Editor, see “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15. To create your own storage class by using an Embedded Coder Dictionary, see “Create Code Definitions for Use in the Code Mappings Editor” on page 32-2.

Direct Application of Structured Storage Class

You can apply a structured storage class to individual data items. The direct application prevents code generation optimizations, such as **Default parameter behavior** and **Signal storage reuse**, from eliminating each data item from the generated code. The direct application also overrides the default storage classes that you specify with **Code Mappings** > **Data Defaults**.

To directly apply a storage class, use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). Set **Change view** to `Code` and apply the storage class by using the **Storage Class** column.

For an example that shows how to use `Struct`, see “Configure Parameters for C Code Generation” on page 33-61. For more information about applying storage classes, see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.

Nonvirtual Buses and Parameter Structures

- To create a nonvirtual bus, use a Bus Creator block to organize multiple signal lines into a single bus, or configure an Inport block or Outport block as a nonvirtual bus. You must create a `Simulink.Bus` object that represents the structure type. For an example, see “Structures of Signals” on page 36-154. For information about nonvirtual buses, see “Create Nonvirtual Buses”.
- To create a parameter structure, use MATLAB commands or the Variable Editor to organize the values of multiple block parameters into a MATLAB structure. Optionally, create a `Simulink.Bus` object so that you can control the name of the structure type and other characteristics such as the data type and dimensions of each field. For an example, see “Structures of Parameters” on page 36-152. For general information about parameter structures, see “Organize Related Block Parameter Definitions in Structures”.

Structures of Parameters

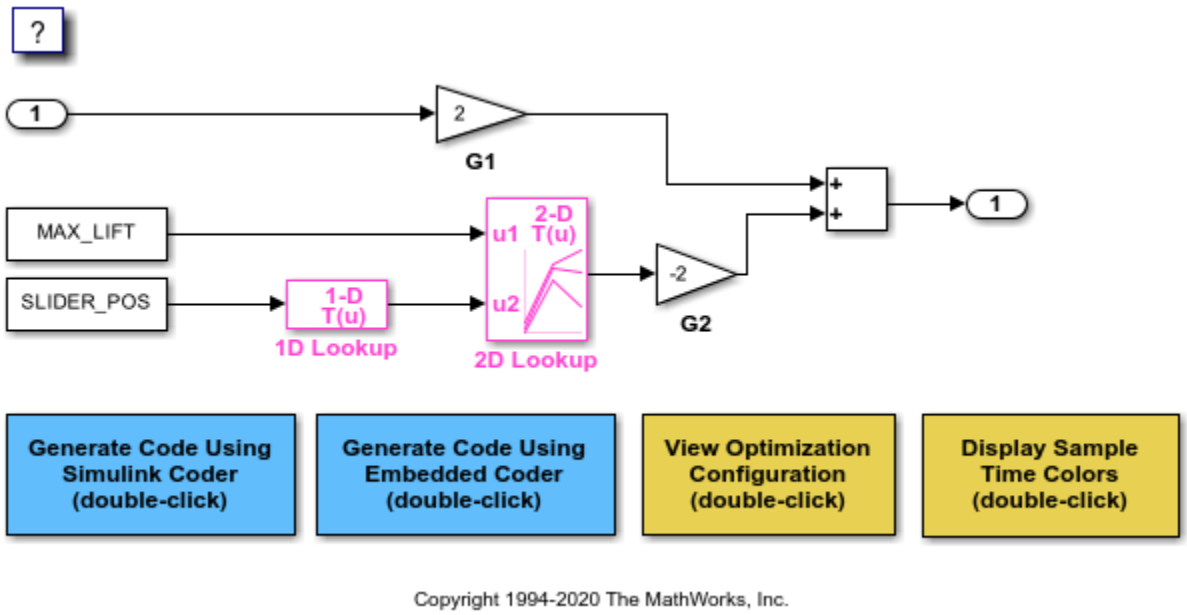
Create a structure in the generated code. The structure stores parameter data.

C Construct

```
typedef struct {
  double G1;
  double G2;
} myStructType;

myStructType myStruct = {
  2.0,
  -2.0
} ;
```

Procedure

1. Open the example model `rtwdemo_paraminline`.



2. On the **Modeling** tab, click **Model Data Editor**. In the Model Data Editor, select the **Parameters** tab.

3. In the model, click the Gain block labeled `G1`. In the Model Data Editor, use the **Value** column to set the value of the **Gain** parameter to `myStruct.G1`.

4. Set the value of the **Gain** parameter in the `G2` block to `myStruct.G2`.

5. Next to `myStruct.G2`, click the action button (with three vertical dots) and select **Create**.

6. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(struct)` and click **Create**. A `Simulink.Parameter` object named `myStruct` appears in the base workspace.

7. In the Simulink.Parameter property dialog box, next to the **Value** property, click the action button and select **Open Variable Editor**.

8. Right-click the white space under the **Field** column and select **New**. Name the new structure field `G1`. Use the **Value** column to set the value of the field to `2`.

9. Add a field `G2` whose value is `-2`, and then close the Variable Editor.

10. In the Simulink.Parameter property dialog box, on the **Code Generation** tab, set **Storage class** to `ExportedGlobal`. The structure `myStruct` appears in the generated code as a global variable.

11. Generate code from the model.

Results

The generated header file `rtwdemo_paraminline_types.h` defines a structure type that has a random name.

```
typedef struct {
  real_T G1;
  real_T G2;
} struct_6h72eH5WFuEIyQr5YrdGuB;
```

The source file `rtwdemo_paraminline.c` defines and initializes the structure variable `myStruct`.

```
/* Exported block parameters */
struct_6h72eH5WFuEIyQr5YrdGuB myStruct = {
  2.0,
  -2.0
} ;                                     /* Variable: myStruct
                                         * Referenced by:
                                         *   '<Root>/G1'
                                         *   '<Root>/G2'
                                         */
```

Specify Name of Structure Type

1. Optionally, specify a name to use for the structure type definition (`struct`). At the command prompt, use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object that represents the structure type.

2. The default name of the object is `slBus1`. Change the name by copying the object into a new MATLAB® variable.

3. In the Model Data Editor, click the **Show/refresh additional information** button.

4. In the data table, find the row that corresponds to `myStruct`. Use the **Data Type** column to set the data type of `myStruct` to `Bus: myStructType`.

5. Generate code from the model.

The code generates the definition of the structure type `myStructType` and uses this type to define the global variable `myStruct`.

```
myStructType myStruct = {
  2.0,
  -2.0
} ;                                     /* Variable: myStruct
```

Structures of Signals

This example shows how to create a structure of signal data in the generated code.

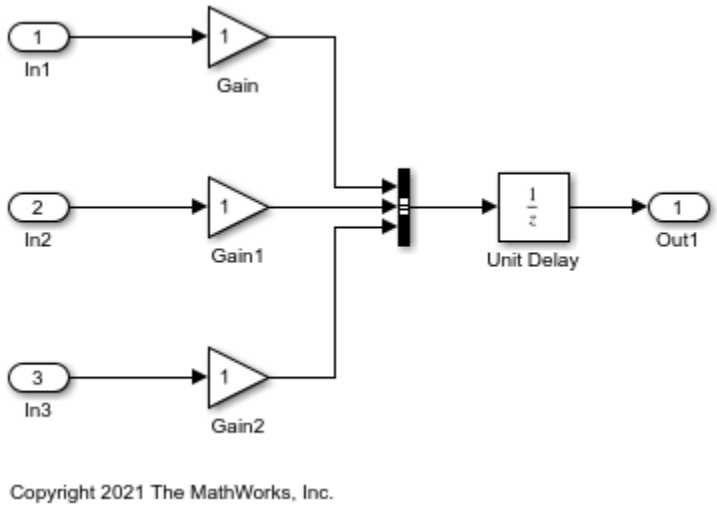
C Construct

```
typedef struct {
   double signal1;
   double signal2;
   double signal3;
} my_signals_type;
```

Procedure

To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of buses in your model.

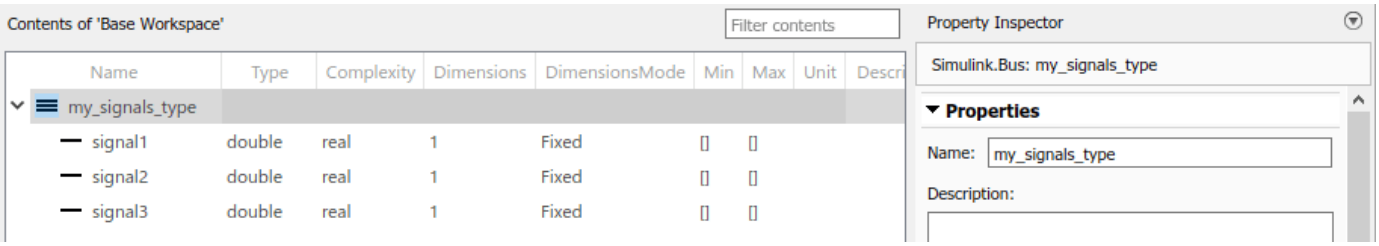
1. Create the `ex_signal_struct` model by using Gain blocks, a Bus Creator block, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



2. To configure the Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

3. In the toolstrip, on the **Modeling** tab, under **Design**, click **Type Editor**.

4. In the Type Editor toolstrip, in the **Add** gallery, click **Bus** to create a `Simulink.Bus` object. Then, name it `my_signals_type`. With `my_signals_type` selected, in the **Add** gallery, click **Bus Element** three times to add three elements. Name them `signal1`, `signal2`, and `signal3`. For more information, see Type Editor.



5. Save the bus objects in your current folder as `ex_signal_struct_data.mat`.

This bus object represents the structure type that you want the generated code to use.

6. In the Bus Creator block dialog box, set **Output data type** to `Bus: my_signals_type`.

7. Select **Output as nonvirtual bus**. Click **OK**. A nonvirtual bus appears in the generated code as a structure.

8. Open the **Simulink Coder** app. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

9. Open the **Signals/States** tab. In the model, select the output signal of the Bus Creator block and click the **Add selected signals to code mappings** button in the Code Mappings editor.

11. For the added signal, set **Storage Class** to `ExportedGlobal`.

12. In the Property Inspector, set the **Code** > **Identifier** property to `sig_struct_var`. The output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

13. Generate code from the model.

Results

The generated header file `ex_signal_struct_types.h` defines the structure type `my_signals_type`.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} my_signals_type;
```

The source file `ex_signal_struct.c` allocates memory for the global variable `sig_struct_var`, which represents the output of the Bus Creator block.

```
/* Exported block signals */
my_signals_type sig_struct_var;        /* '<Root>/Bus Creator' */
```

In the same file, in the model `step` function, the algorithm accesses `sig_struct_var` and the fields of `sig_struct_var`.

Nested Structures of Signals

You can create nested structures of signal data in the generated code.

C Construct

```
typedef struct {
  double signal1;
  double signal2;
  double signal3;
} B_struct_type;

typedef struct {
  double signal1;
  double signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
```

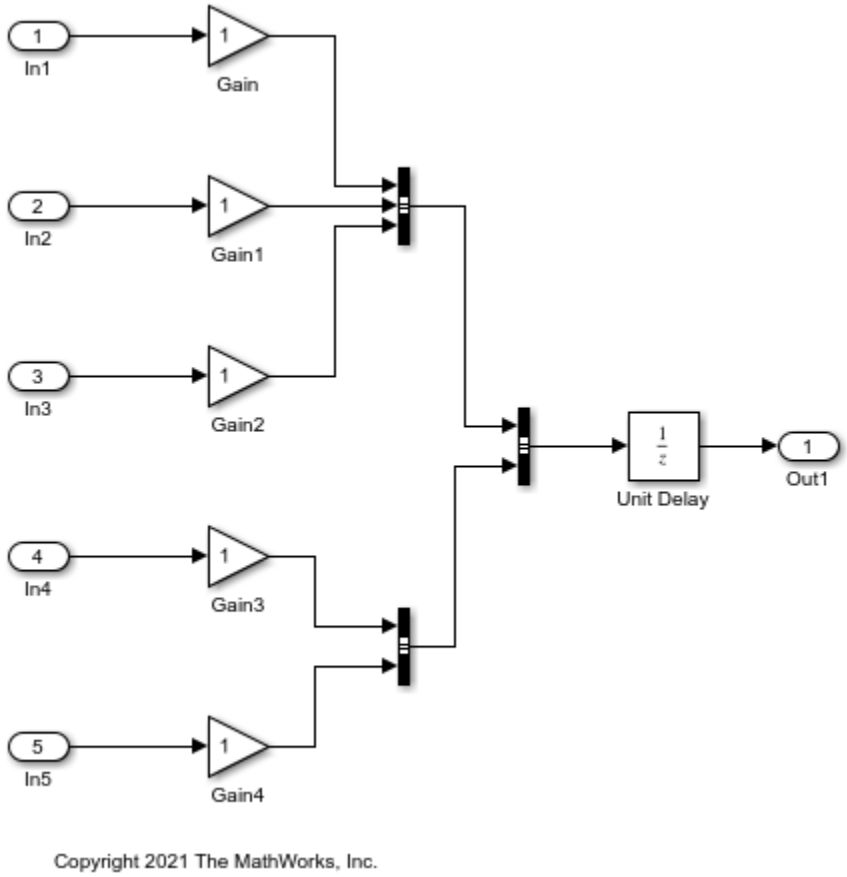
```
  C_struct_type subStruct_C;
} A_struct_type;
```

Procedure

To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of buses in your model.

To nest a structure inside another structure, use a bus object as the data type of an element in another bus object.

1. Create the `ex_signal_nested_struct` model with Gain blocks, Bus Creator blocks, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



2. To configure a Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

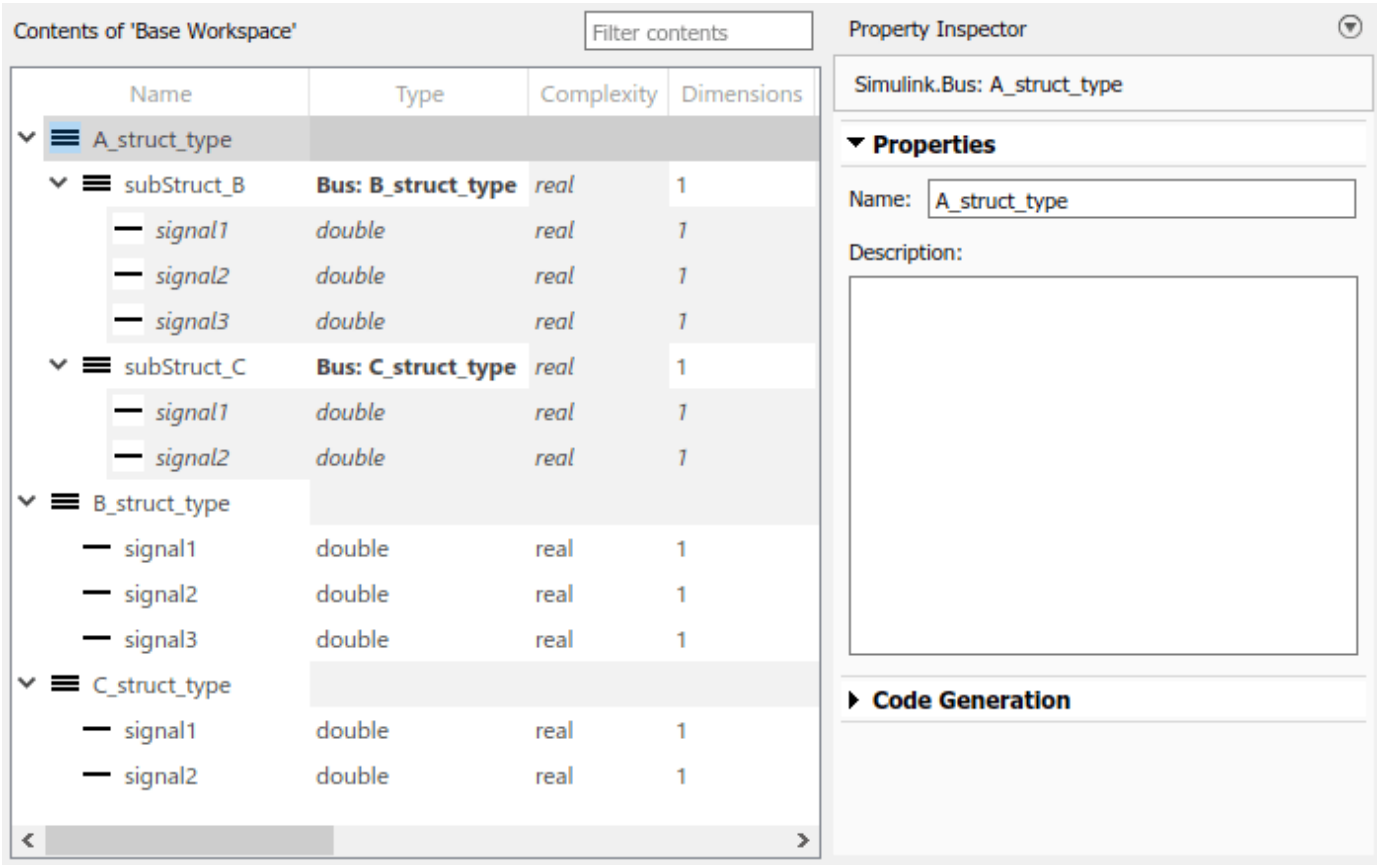
3. In the toolstrip, on the **Modeling** tab, under **Design**, click **Type Editor**.

In the Type Editor toolstrip, in the **Add** gallery, click **Bus** to create a `Simulink.Bus` object. Then, name it `A_struct_type`. With `A_struct_type` selected, in the **Add** gallery, click **Bus Element** twice to add two elements. Name the elements `subStruct_B` and `subStruct_C`. For more

information, see Type Editor. This bus object represents the top-level structure type that you want the generated code to use.

4. Similarly, create `Simulink.Bus` objects `B_struct_type` (with three elements) and `C_struct_type` (with two elements).

5. In the `A_struct_type` object, for the `subStruct_B` element, set **DataType** to `Bus: B_struct_type`. Similarly, for the `subStruct_C` element, set **DataType** to `Bus: C_struct_type`.



Each element in `A_struct_type` uses another bus object as a data type. These elements represent substructures.

6. In the dialog box of the Bus Creator block that collects the three Gain signals, set **Output data type** to `Bus: B_struct_type`. Click **Apply**.

7. Select **Output as nonvirtual bus** and click **OK**.

8. In the dialog box of the other subordinate Bus Creator block, set **Output data type** to `Bus: C_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

9. In the last Bus Creator block dialog box, set **Output data type** to `Bus: A_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

10. Open the **Simulink Coder** app. In the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

11. Open the **Signals/States** tab. In the model, select the output signal of the `A_struct_type` Bus Creator block, which feeds the Unit Delay block. Click the **Add selected signals to code mappings** button in the Code Mappings editor.

12. For the added signal, set **Storage Class** to `ExportedGlobal`.

13. In the Property Inspector, set the **Code** > **Identifier** property to `sig_struct_var`. The output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

14. Generate code from the model.

Results

The generated header file `ex_signal_nested_struct_types.h` defines the structure types. Each structure type corresponds to a `Simulink.Bus` object.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} B_struct_type;

typedef struct {
  real_T signal1;
  real_T signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
  C_struct_type subStruct_C;
} A_struct_type;
```

The generated source file `ex_signal_nested_struct.c` allocates memory for the global structure variable `sig_struct_var`. By default, the name of the `A_struct_type` Bus Creator block is `Bus Creator2`.

```
/* Exported block signals */
A_struct_type sig_struct_var;          /* '<Root>/Bus Creator2' */
```

In the same file, in the model `step` function, the algorithm accesses `sig_struct_var` and the fields of `sig_struct_var`.

Combine Techniques to Work Around Limitations

To work around some of the limitations of each technique, you can combine structured storage classes with nonvirtual buses and parameter structures. For example, you can:

- Include a structure of signal data and a structure of parameter data in the same parent structure.
- Nest structures while aggregating new data by default.

In the generated code, a flat parent structure, which corresponds to the structured storage class, contains substructures, which correspond to each bus and parameter structure. Choose one of these combined techniques:

- Apply the structured storage class directly to each bus and parameter structure. For example, set the storage class of two nonvirtual bus signals to `Struct`. Each bus appears in the generated code as a field (substructure) of a single structure.
- Leave the storage class of each bus and parameter structure at the default setting, `Auto`, or at `Model default`, which prevents code generation optimizations from eliminating the bus or parameter structure. Then, configure default storage classes such that signal data and parameter data use the structured storage class by default.

Arrays of Structures

You can further package multiple consistent bus signals or parameter structures into an array. The array of buses or parameter structures appears in the generated code as an array of structures. To create arrays of buses, see "Group Nonvirtual Buses in Arrays of Buses". To create arrays of parameter structures, see "Group Multiple Parameter Structures into an Array".

Structure Padding

By default, the code generator does not explicitly add padding fields to structure types. Structure types can appear in the generated code through, for example, the standard data structures (see "Data Structures in the Generated Code" on page 36-31), `Simulink.Bus` objects, and parameter structures that you use in a model.

However, when you use a code replacement library with Embedded Coder, you can specify data alignment (including structure padding) as part of the replacement library. For more information, see "Provide Data Alignment Specifications for Compilers" on page 72-106.

Limitations

- With built-in Simulink Coder and Embedded Coder features, you cannot generate or use a custom structure that contains a field whose value is a pointer. You can do so manually by creating an advanced storage class and writing accompanying TLC code (see "Finely Control Data Representation by Writing TLC Code for a Storage Class" on page 40-23).
- You cannot use the built-in storage class `Struct`, or a structured storage class that you create with the Custom Storage Class Designer (you set the storage class property **Type** to `FlatStructure`), to set data defaults in the Code Mapping Editor.

See Also

`Simulink.Bus`

Related Examples

- "How Generated Code Stores Internal Signal, State, and Parameter Data"
- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code"
- "C Code Generation Configuration for Model Interface Elements" on page 33-2
- "Control Data and Function Placement in Memory by Inserting Pragmas" on page 45-2
- "Flexible Storage Class for Different Model Hierarchy Contexts" on page 32-34
- "Composite Interface Guidelines"

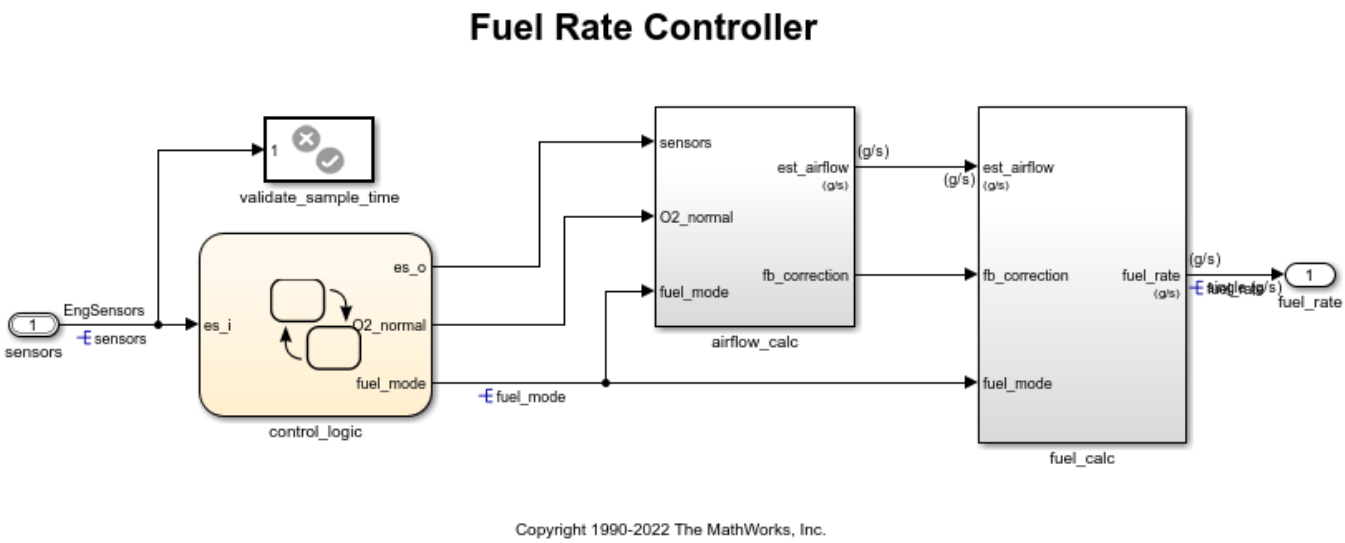
Switch Between Sets of Parameter Values During Simulation and Code Execution

To store multiple independent sets of values for the same block parameters, you can use an array of structures. To switch between the parameter sets, create a variable that acts as an index into the array, and change the value of the variable. You can change the value of the variable during simulation and, if the variable is tunable, during execution of the generated code.

Explore Example Model

Open the example model.

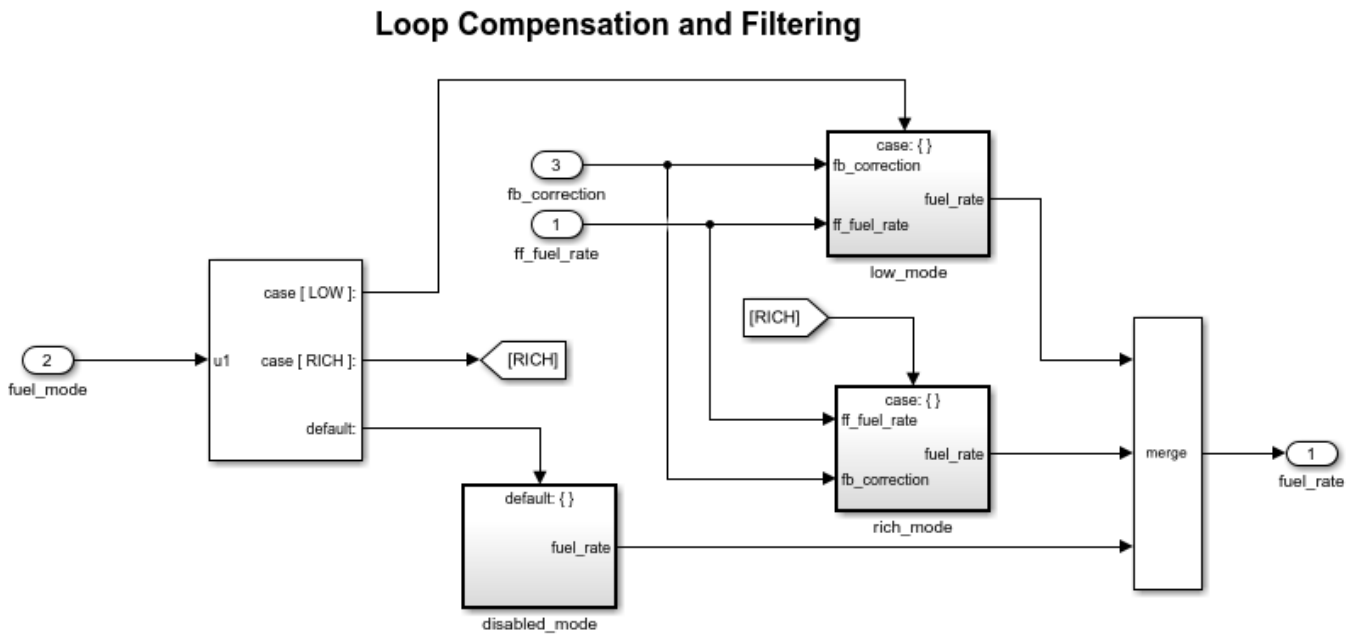
```
open_system('sldemo_fuelsys_dd_controller')
```



This model represents the fueling system of a gasoline engine. The output of the model is the rate of fuel flow to the engine.

Navigate to the `switchable_compensation` nested subsystem.

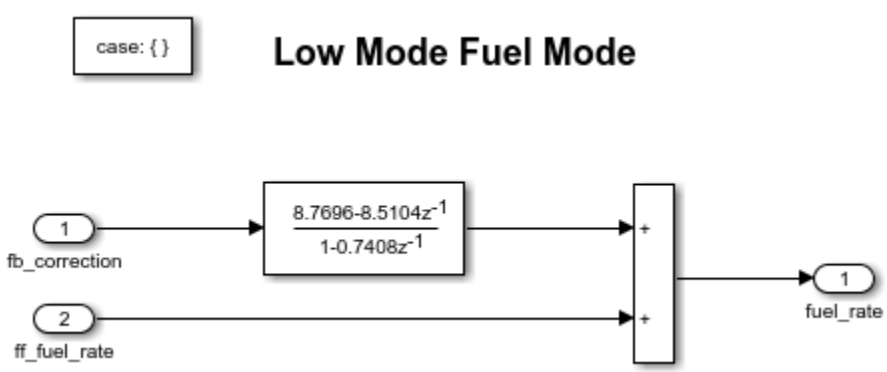
```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation'])
```



This subsystem corrects and filters noise out of the fuel rate signal. The subsystem uses different filter coefficients based on the fueling mode, which the control logic changes based on sensor failures in the engine. For example, the control algorithm activates the `low_mode` subsystem during normal operation. It activates the `rich_mode` subsystem in response to sensor failure.

Open the `low_mode` subsystem.

```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation/low_mode'])
```



The Discrete Filter block filters the fuel rate signal. In the block dialog box, the **Numerator** parameter sets the numerator coefficients of the filter.

The sibling subsystem `rich_mode` also contains a Discrete Filter block, which uses different coefficients.

Update the model diagram to display the signal data types. The input and output signals of the block use the single-precision, floating-point data type `single`.

In the lower-left corner of the model, click the model data badge, then click the **Data Dictionary** link. The data dictionary for this model, `sldemo_fuelsys_dd_controller.sldd`, opens in the Model Explorer.

In the Model Explorer **Model Hierarchy** pane, select the **Design Data** node.

In the **Contents** pane, view the properties of the `Simulink.NumericType` objects, such as `s16En15`. All of these objects currently represent the single-precision, floating-point data type `single`. The model uses these objects to set signal data types, including the input and output signals of the Discrete Filter blocks.

Suppose that during simulation and execution of the generated code, you want each of these subsystems to switch between different numerator coefficients based on a variable whose value you control.

Store Parameter Values in Array of Structures

Store the existing set of numerator coefficients in a `Simulink.Parameter` object whose value is a structure. Each field of the structure stores the coefficients for one of the Discrete Filter blocks.

```
lowBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/low_mode/Discrete Filter'];
richBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/rich_mode/Discrete Filter'];
params.lowNumerator = eval(get_param(lowBlock,'Numerator'));
params.richNumerator = eval(get_param(richBlock,'Numerator'));
params = Simulink.Parameter(params);
```

Copy the value of `params` into a temporary variable. Modify the field values in this temporary structure, and assign the modified structure as the second element of `params`.

```
temp = params.Value;
temp.lowNumerator = params.Value.lowNumerator * 2;
temp.richNumerator = params.Value.richNumerator * 2;
params.Value(2) = temp;
clear temp
```

The value of `params` is an array of two structures. Each structure stores one set of filter coefficients.

Create Variable to Switch Between Parameter Sets

Create a `Simulink.Parameter` object named `Ctrl`.

```
Ctrl = Simulink.Parameter(2);
Ctrl.DataType = 'uint8';
```

In the `low_mode` subsystem, in the Discrete Filter block dialog box, set the **Numerator** parameter to the expression `params(Ctrl).lowNumerator`.

```
set_param(lowBlock,'Numerator','params(Ctrl).lowNumerator');
```

In the Discrete Filter block in the `rich_mode` subsystem, set the value of the **Numerator** parameter to `params(Ctrl).richNumerator`.

```
set_param(richBlock,'Numerator','params(Ctrl).richNumerator');
```

The expressions select one of the structures in `params` by using the variable `Ctrl`. The expressions then dereference one of the fields in the structure. The field value sets the values of the numerator coefficients.

To switch between the sets of coefficients, you change the value of `Ctrl` to the corresponding index in the array of structures.

Use Bus Object as Data Type of Array of Structures

Optionally, create a `Simulink.Bus` object to use as the data type of the array of structures. You can:

- Control the shape of the structures.
- For each field, control characteristics such as data type and physical units.
- Control the name of the `struct` type in the generated code.

Use the function `Simulink.Bus.createObject` to create the object and rename the object as `paramsType`.

```
Simulink.Bus.createObject(params.Value)
paramsType = slBus1;
clear slBus1
```

You can use the `Simulink.NumericType` objects from the data dictionary to control the data types of the structure fields. In the bus object, use the name of a data type object to set the `DataType` property of each element.

```
paramsType.Elements(1).DataType = 's16En15';
paramsType.Elements(2).DataType = 's16En7';
```

Use the bus object as the data type of the array of structures.

```
params.DataType = 'Bus: paramsType';
```

Use Enumerated Type for Switching Variable

Optionally, use an enumerated type as the data type of the switching variable. You can associate each of the parameter sets with a meaningful name and restrict the allowed values of the switching variable.

Create an enumerated type named `FilterCoeffs`. Create an enumeration member for each of the structures in `params`. Set the underlying integer value of each enumeration member to the corresponding index in `params`.

```
Simulink.defineIntEnumType('FilterCoeffs',{'Weak','Aggressive'},[1 2])
```

Use the enumerated type as the data type of the switching variable. Set the value of the variable to `Aggressive`, which corresponds to the index 2.

```
Ctrl.Value = FilterCoeffs.Aggressive;
```

Add New Objects to Data Dictionary

Add the objects that you created to the data dictionary `sldemo_fuelsys_dd_controller.sldd`.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
sectObj = getSection(dictObj,'Design Data');
addEntry(sectObj,'Ctrl',Ctrl)
```

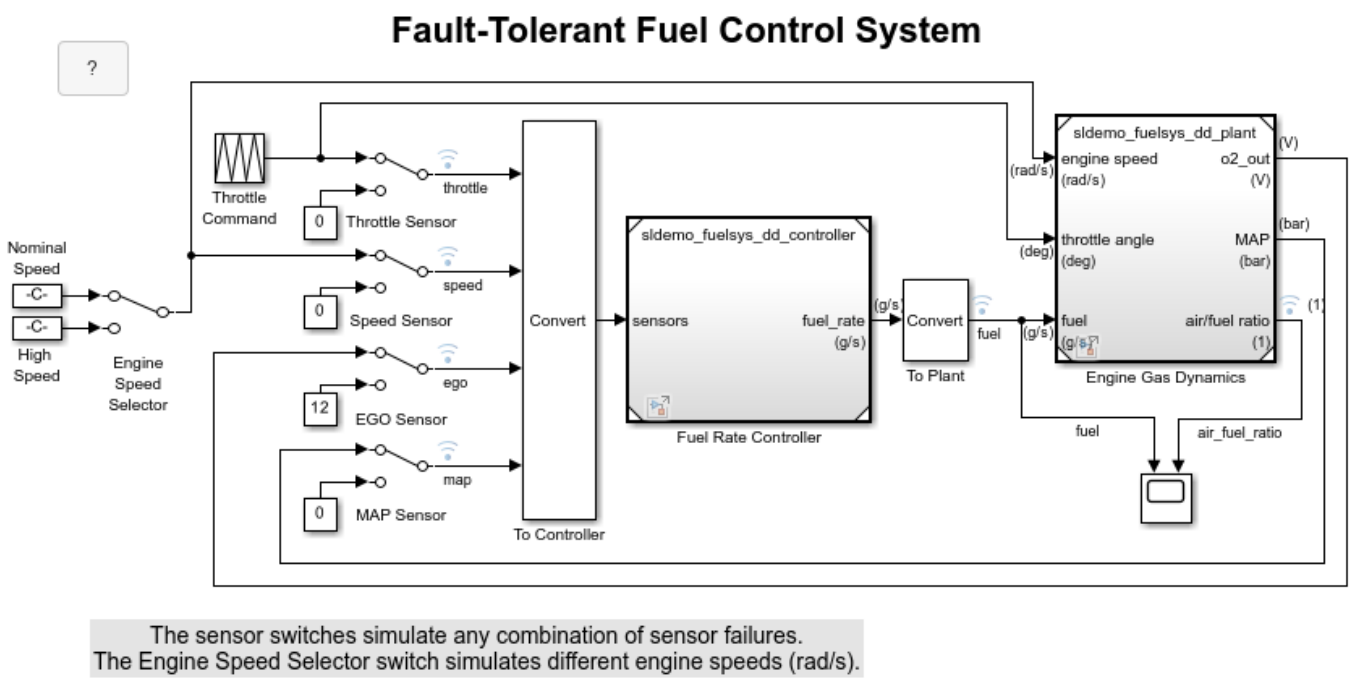
```
addEntry(sectObj,'params',params)
addEntry(sectObj,'paramsType',paramsType)
```

You can also store enumerated types in data dictionaries. However, you cannot import the enumerated type in this case because you cannot save changes to `sldemo_fuelsys_dd_controller.sldd`. For more information about storing enumerated types in data dictionaries, see “Enumerations in Data Dictionary”.

Switch Between Parameter Sets During Simulation

Open the example model `sldemo_fuelsys_dd`, which references the controller model `sldemo_fuelsys_dd_controller`.

```
open_system('sldemo_fuelsys_dd')
```



Set the simulation stop time to `Inf` so that you can interact with the model during simulation.

Begin a simulation run and open the Scope block dialog box. The scope shows that the fuel flow rate (the `fuel` signal) oscillates with significant amplitude during normal operation of the engine.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. Set the value of `Ctrl` to `FilterCoeffs.Weak`.

Update the `sldemo_fuelsys_dd` model diagram. The scope shows that the amplitude of the fuel rate oscillations decreases due to the less aggressive filter coefficients.

Stop the simulation.

Generate and Inspect Code

If you have Simulink Coder software, you can generate code that enables you to switch between the parameter sets during code execution.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. In the **Contents** pane, set **Column View** to `Storage Class`.

Use the **StorageClass** column to apply the storage class `ExportedGlobal` to `params` so that the array of structures appears as a tunable global variable in the generated code. Apply the same storage class to `Ctrl` so that you can change the value of the switching variable during code execution.

Alternatively, to configure the objects, use these commands:

```
tempEntryObj = getEntry(sectObj,'params');
params = getValue(tempEntryObj);
params.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,params);

tempEntryObj = getEntry(sectObj,'Ctrl');
Ctrl = getValue(tempEntryObj);
Ctrl.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,Ctrl);
```

Generate code from the controller model.

```
slbuild('sldemo_fuelsys_dd_controller')
```

```
### Starting build procedure for: sldemo_fuelsys_dd_controller
### Successful completion of code generation for: sldemo_fuelsys_dd_controller

Build Summary

Top model targets built:

Model                          Action           Rebuild Reason
========================================================================================
sldemo_fuelsys_dd_controller  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 6.3938s
```

In the code generation report, view the header file `sldemo_fuelsys_dd_controller_types.h`. The code defines the enumerated data type `FilterCoeffs`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_types.h');
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_
#define DEFINED_TYPEDEF_FOR_FilterCoeffs_

typedef enum {
  Weak = 1,                             /* Default value */
```

```
  Aggressive
} FilterCoeffs;

#endif
```

The code also defines the structure type `paramsType`, which corresponds to the `Simulink.Bus` object. The fields use the single-precision, floating-point data type from the model.

```
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_paramsType_',...
    '#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_paramsType_
#define DEFINED_TYPEDEF_FOR_paramsType_

typedef struct {
  real32_T lowNumerator[2];
  real32_T richNumerator[2];
} paramsType;

#endif
```

View the source file `sldemo_fuelsys_dd_controller.c`. The code uses the enumerated type to define the switching variable `Ctrl`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller.c');
rtwdemodbtype(file,'FilterCoeffs Ctrl = Aggressive;',...
    '/* Block signals (default storage) */',1,0)
```

```
FilterCoeffs Ctrl = Aggressive;        /* Variable: Ctrl
                                        * Referenced by:
                                        *   '<S12>/Discrete Filter'
                                        *   '<S13>/Discrete Filter'
                                        */
```

The code also defines the array of structures `params`.

```
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Variable: params',1,1)
```

```
/* Exported block parameters */
paramsType params[2] = { {
    { 8.7696F, -8.5104F },

    { 0.0F, 0.2592F }
  }, { { 17.5392F, -17.0208F },

    { 0.0F, 0.5184F }
  } } ;                                /* Variable: params
```

The code algorithm in the model `step` function uses the switching variable to index into the array of structures.

To switch between the parameter sets stored in the array of structures, change the value of `Ctrl` during code execution.

Close the connections to the data dictionary. This example discards unsaved changes. To save the changes, use the `'-save'` option.

```
Simulink.data.dictionary.closeAll('sldemo_fuelsys_dd_controller.sldd','-discard')
```

See Also

Related Examples

- “Tune and Experiment with Block Parameter Values”
- “Create Tunable Calibration Parameter in the Generated Code”
- “Organize Related Block Parameter Definitions in Structures”
- “Access Structured Data Through a Pointer That External Code Defines” on page 40-10

Design Data Interface by Configuring Inport and Outport Blocks

The data interface of a model is the means by which the model exchanges data (for example, signal values) with other, external models or systems. Customize the data interface of a model to:

- Enable integration of the generated code with your own code.
- Improve traceability and readability of the code.

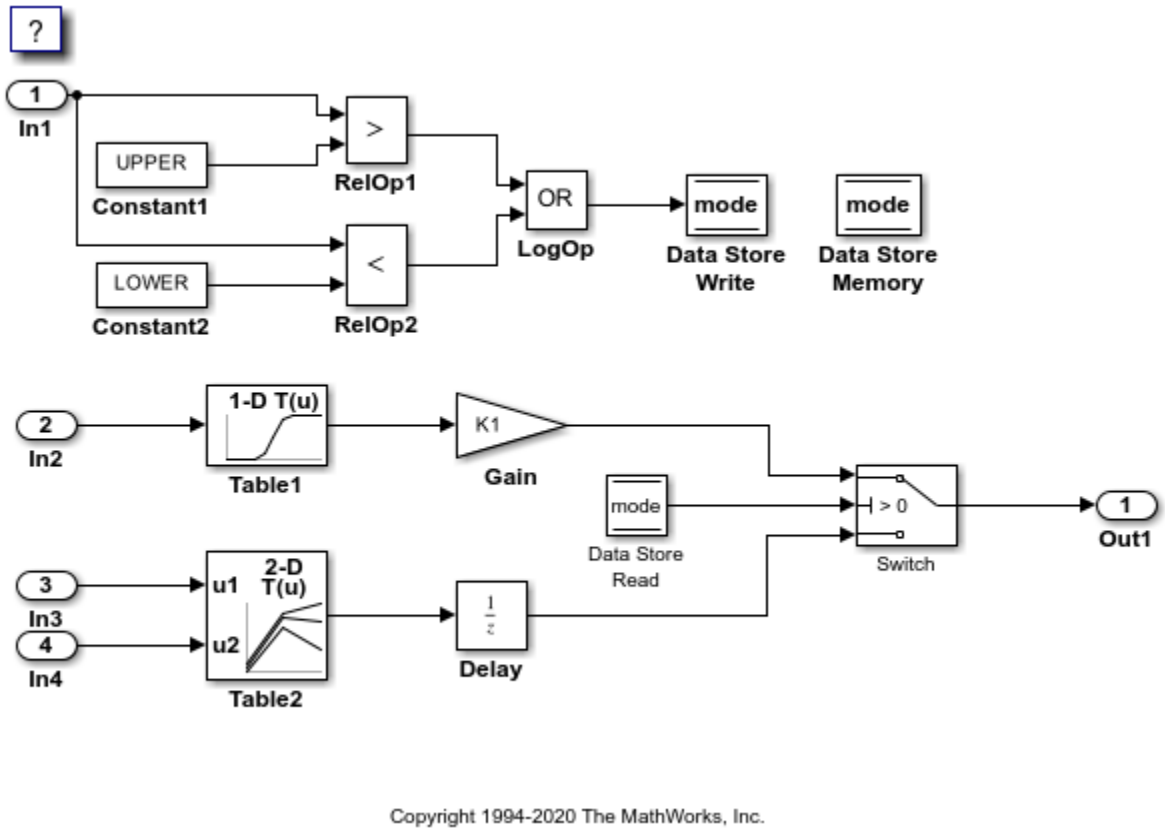
At the top level of a model, Inport and Outport blocks represent the input and output signals of the model. To customize the data interface in the generated code, configure these blocks. Early in the design process, when a model can contain unconnected Inport and Outport blocks, use this technique to specify the interface before developing the internal algorithm.

When you apply storage classes to Inport and Outport blocks, each block appears in the generated code as a field of a global structure or as a separate global variable that the generated algorithm references directly. If you have Embedded Coder®, you can use function prototype control instead of storage classes to pass data into and out of the model `step` function as formal parameters. For more information, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

Design Data Interface

Open the example model `rtwdemo_configrpinterface`.

```
open_system('rtwdemo_configrpinterface')
```



Configure the model to show the generated names of blocks.

```
set_param('rtwdemo_configrpinterface','HideAutomaticNames','off')
```

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, select the **Inports/Outports** tab. Each row in the table represents an Outport block or a signal that exits an Inport block.

For each of the signals that exit the Inport blocks, set **Data Type** to `single` or to a different data type. Due to the data type inheritance settings that the other blocks in the model use by default, downstream signals in the rest of the model use the same or a similar data type.

Optionally, configure other design attributes such as **Min** and **Max** (minimum and maximum values).

Configure Default Storage Class

You can configure a default storage class for Inport blocks and Outport blocks then select the blocks that should use the specified default.

1. Open the **Simulink Coder** app.
2. In the C Code tab, select **Code Interface > Default Code Mappings**.
3. In the Code Mappings editor, under **Inports and Outports**, set the default storage class for both **Inports** and **Outports** to `ExportedGlobal`.

4. Next, on the **Inports** tab, select the data items that should use the default storage class. Press **Ctrl** and click each inport.
5. For the selected inports, set the storage class to `Model default: ExportedGlobal`.
6. Similarly, on the **Outports** tab, set the storage class for the outport to `Model default: ExportedGlobal`.

Alternatively, to configure the data defaults at the command prompt, use these commands:

```
cm = coder.mapping.api.get('rtwdemo_configrpinterface');
setDataDefault(cm,'Inports','StorageClass','ExportedGlobal');
setDataDefault(cm,'Outports','StorageClass','ExportedGlobal');

setInport(cm,'In1','StorageClass','Model default');
setInport(cm,'In2','StorageClass','Model default');
setInport(cm,'In3','StorageClass','Model default');
setInport(cm,'In4','StorageClass','Model default');

setOutport(cm,'Out1','StorageClass','Model default','Identifier','output_sig');
```

Generate code from the model.

```
slbuild('rtwdemo_configrpinterface');
```

```
### Starting build procedure for: rtwdemo_configrpinterface
### Successful completion of code generation for: rtwdemo_configrpinterface

Build Summary

Top model targets built:

Model                       Action           Rebuild Reason
========================================================================================
rtwdemo_configrpinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.312s
```

View the generated file `rtwdemo_configrpinterface.c`. Because you applied the storage class `ExportedGlobal` to the Inport and Outport blocks, the code creates separate global variables that represent the inputs and the output.

```
file = fullfile('rtwdemo_configrpinterface_grt_rtw','rtwdemo_configrpinterface.c');
rtwdemodbtype(file,'/* Data with Exported storage */','real_T output_sig;',1,1)
```

```
/* Data with Exported storage */
real_T input1;                          /* '<Root>/In1' */
real_T input2;                          /* '<Root>/In2' */
real_T input3;                          /* '<Root>/In3' */
real_T input4;                          /* '<Root>/In4' */
real_T output_sig;                      /* '<Root>/Out1' */
```

The generated algorithm in the model `step` function directly references these global variables to calculate and store the output signal value, `output_sig`.

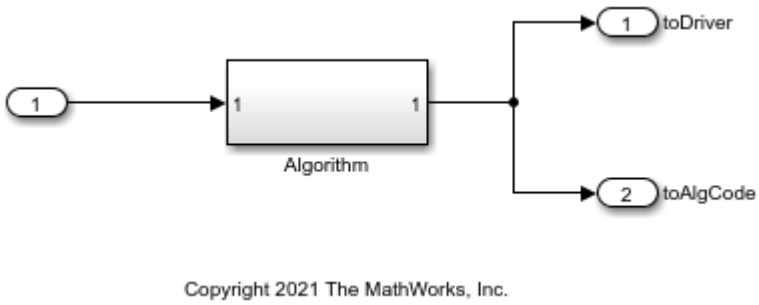
Route Signal Data to Multiple Outputs

Once you have configured the default storage classes for Inports and Outports, you can choose to override this default and set the storage class for some data elements individually. For example, you

can route a single signal to multiple Outport blocks and apply a different storage class to each Outport. For example, use this technique to send signal data to a custom function (such as a device driver) and to a global variable that your custom algorithmic code can use:

1. Branch the target signal line to each Outport block.
2. For more efficient code, leave the storage class of the target signal line configured to `Auto` (the default). Optimizations can then eliminate the signal line from the generated code.
3. In the Code Mappings editor, on the **Outports** tab, apply the storage class `GetSet` to one Outport block and `ExportToFile` to the other Outport block.
4. In the Property Inspector, configure the properties associated with each applied storage class.

```
open_system('ex_route_sig')
```



Limitations

You cannot apply a storage class to an Outport block if the input to the block has a variable size. Instead, apply the storage class to the signal line.

See Also

Related Examples

- “Configure Root-Level Inport Blocks for C Code Generation”
- “Configure Root-Level Outport Blocks for C Code Generation”
- “Analyze Generated Data Code Interface” on page 54-13
- “Define Interfaces of Model Components”
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Configure Generated Code According to Interface Control Document Specifications” on page 40-54
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114

See Also

Code Mappings Editor – C | `coder.mapping.api.CodeMapping`

Generate Efficient Code for Bus Signals

In this section...
“Code Efficiency for Bus Signals” on page 36-173 “Set Bus Diagnostics” on page 36-173 “Optimize Virtual and Nonvirtual Buses” on page 36-174

In a model, you use bus signals to package multiple signals together into a single signal line. You can create virtual or nonvirtual bus signals. The representation in the generated code depends on:

- For a virtual bus, the generated code appears as if the bus did not exist.
- Generated code for a nonvirtual bus represents the bus data with a structure. When you want to trace the correspondence between the model and the code, the use of a structure in the generated code can be helpful. To generate structures using nonvirtual bus signals, see “Organize Data into Structures in Generated Code” on page 36-147.

For general information about buses, see and “Composite Interface Guidelines”.

To generate efficient code from models that contain bus signals, eliminate unnecessary data copies by following best practices as you construct the model.

Code Efficiency for Bus Signals

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

- Setting bus diagnostic configuration parameters can make model development easier.
- The bus implementation techniques, and the choice of a nonvirtual or virtual bus, can influence the speed, size, and clarity of the generated code.
- Some useful bus implementation techniques are not immediately obvious.

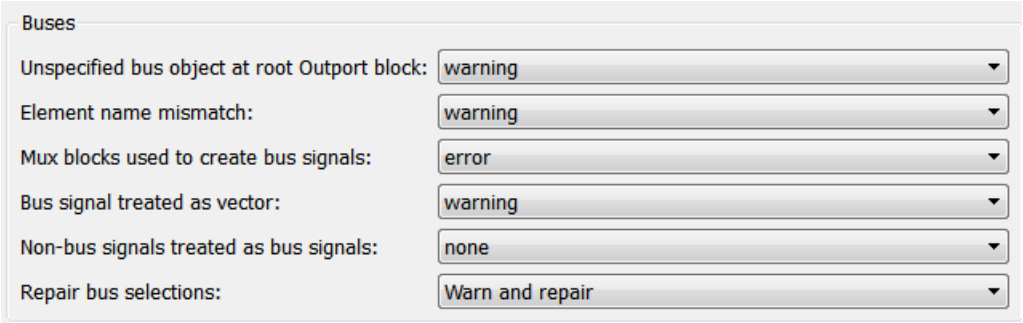
When you work with buses, these guidelines help you to improve the results. The guidelines describe techniques to:

- Simplify the layout of the model.
- Increase the efficiency of generated code.
- Define data structures for function (subsystem) interfaces.
- Define data structures that match existing data structures in external C code.

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

Set Bus Diagnostics

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



Optimize Virtual and Nonvirtual Buses

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. Use virtual buses rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses by using Signal Conversion blocks. In some cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a Stateflow chart converts an input virtual bus to a nonvirtual bus.

To bundle function-call signals, you must use a virtual bus.

You must use nonvirtual buses for:

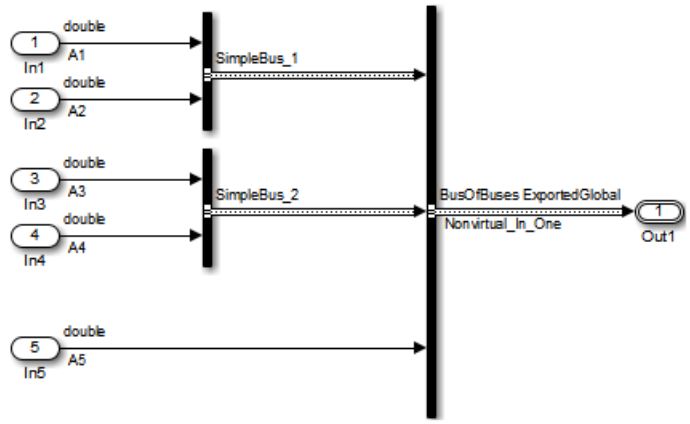
- Nonauto storage classes
- Generating a specific structure from the bus
- Root-level Inport or Outport blocks when the bus has mixed data types

Avoid Nonlocal Nested Buses in Nonvirtual Buses

Buses can contain subordinate buses. To generate efficient code, set the storage classes of subordinate buses to `Auto`. Setting the storage class to `Auto` eliminates:

- Allocation of redundant memory for the subordinate bus signal and for the parent bus signal
- Additional copy operations (copying data to the subordinate bus, and then copying from the subordinate bus to the final bus)

This model contains nonvirtual bus signals. The subordinate bus signals `Sub_Bus_1` and `Sub_Bus_2` use the storage class `Auto`.



The generated code algorithm efficiently assigns the input signal data to the bus signals.

```
void ex_nonvirtual_buses_step(void)
{
  Nonvirtual_In_One.SimpleBus_1.A1 = A1;
  Nonvirtual_In_One.SimpleBus_1.A2 = A2;
  Nonvirtual_In_One.SimpleBus_2.A3 = A3;
  Nonvirtual_In_One.SimpleBus_2.A4 = A4;
  Nonvirtual_In_One.A5 = A5;
}
```

See Also

`Simulink.Bus`

Related Examples

- “Organize Data into Structures in Generated Code” on page 36-147
- “Modify Sample Times for Nonvirtual Buses”

Control Signal and State Initialization in the Generated Code

To initialize signals and discrete states with custom values for simulation and code generation, you can use signal objects and block parameters. Data initialization increases application reliability and is a requirement of safety critical applications. Initializing signals for both simulation and code generation can expedite transitions between phases of Model-Based Design.

For basic information about specifying initial values for signals and discrete states in a model, see “Initialize Signals and Discrete States”.

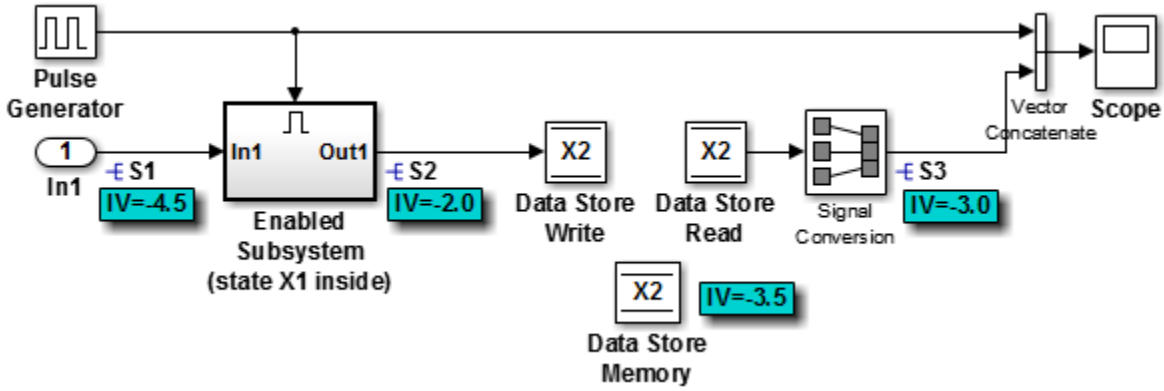
Signal and State Initialization in the Generated Code

The initialization behavior for code generation is the same as that for model simulation with the following exceptions:

- RSim executables can use the **Data Import/Export** pane of the Configuration Parameters dialog box to load input values from MAT-files. GRT and ERT executables cannot load input values from MAT-files.
- The initial value for a block output signal or root level input or output signal can be overwritten by an external (calling) program.
- Setting the initial value for persistent signals is relevant if the value is used or viewed by an external application.

When you generate code, initialization statements are placed in *model*`.c` or *model*`.cpp` in the model's initialize code.

For example, consider the model `rtwdemo_sigobj_iv`.



If you create and initialize signal objects in the base workspace, the code generator places initialization code for the signals in the file `rtwdemo_sigobj_iv.c` under the `rtwdemo_sigobj_iv_initialize` function, as shown below.

```
/* Model initialize function */

    void rtwdemo_sigobj_iv_initialize(void)

    {
       .
       .
       .
```

```
 /* exported global signals */
 S3 = -3.0;

 S2 = -2.0;
          .
          .
          .
/* exported global states */
X1 = 0.0;
X2 = 0.0;

/* external inputs */
S1 = -4.5;
          .
          .
          .
```

The following code shows the initialization code for the enabled subsystem's Unit Delay block state `X1` and output signal `S2`.

```
void MdlStart(void) {
           .
           .
           .
 /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
  X1 = aa1;

 /* Start for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */

  /* virtual outports code */

  /* (Virtual) Outport Block: '<S2>/Out1' */

  S2 = aa2;

}
```

For an enabled subsystem, the initial value is also used as a reset value if the subsystem's Outport block parameter **Output when disabled** is set to `reset`. The following code from `rtwdemo_sigobj_iv.c` shows the assignment statement for `S3` as it appears in the model output function `rtwdemo_sigobj_iv_output`.

```
/* Model output function */

static void rtwdemo_sigobj_iv_output(void)
{
          .
          .
          .
 /* Disable for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */

    /* (Virtual) Outport Block: '<S2>/Out1' */

    S2 = aa2;
```

Generate Tunable Initial Conditions

You can represent initial conditions for signals and states by creating tunable global variables in the generated code. These variables allow you to restart an application by using initial conditions that are stored in memory.

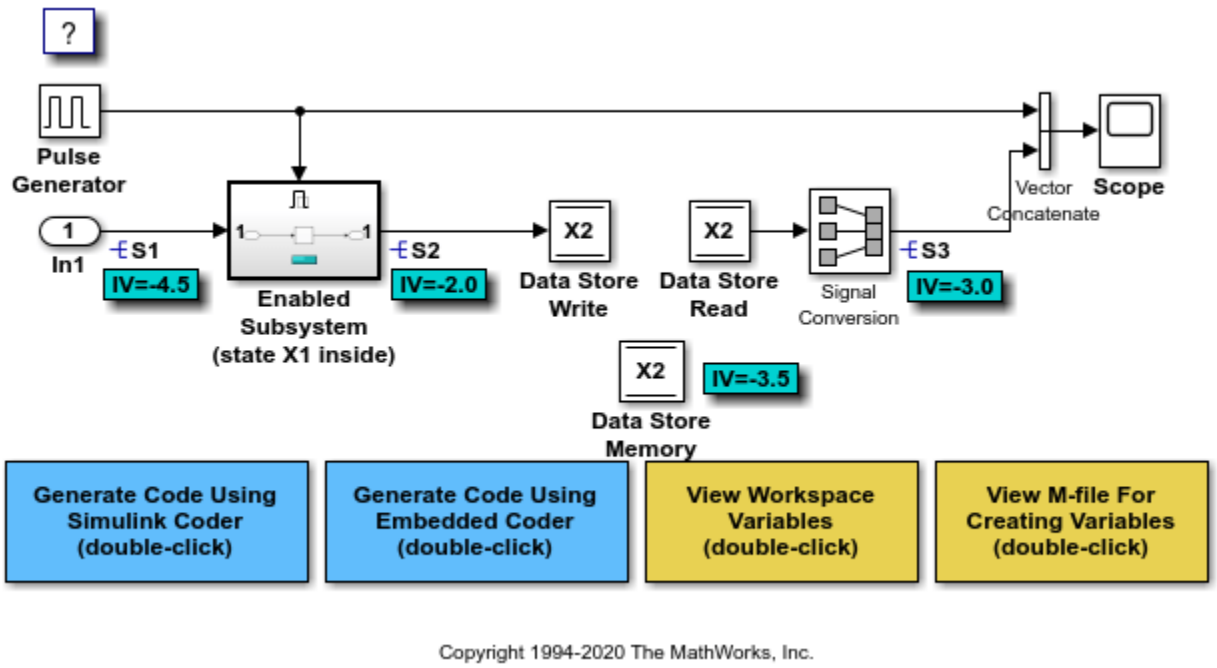
If you set **Configuration Parameters > Optimization > Default parameter behavior** to `Tunable`, initial conditions appear as tunable fields of the global parameters structure.

Whether you set **Default parameter behavior** to `Tunable` or `Inlined`, you can use a tunable parameter to specify the `InitialValue` property of a signal object or the **Initial condition** parameter of a block. For basic information about tunable parameters, see “Create Tunable Calibration Parameter in the Generated Code”.

This example shows how to use tunable parameters to specify initial conditions for signals and states.

Explore Example Model

Open the example model `rtwdemo_sigobj_iv` and configure it to show the generated names of blocks. The signal `S2` uses a `Simulink.Signal` object in the base workspace.



Double-click the `Simulink.Signal` object `S2` to view its properties. The **Initial value** property is set to `aa2`. The object uses the variable `aa2` to specify an initial condition for the signal `S2`. The **Storage class** property is set to `ExportedGlobal`. To use a `Simulink.Signal` object to initialize a signal, the signal object must use a storage class other than `Auto` or, if the corresponding data category in the Code Mapping Editor uses a storage class setting other than `Default`, `Model default`.

On the **Optimization** pane, in the Configuration Parameters dialog box, click **Configure**. The variable `aa2` is a tunable parameter that uses the storage class `ExportedGlobal`.

In the model, open the Enabled Subsystem. In the Outport block dialog box, the parameter **Output when disabled** is set to `reset`. When the subsystem becomes disabled, the output signal `S2` resets to the initial value `aa2`.

Open the Unit Delay block dialog box. On the **State Attributes** tab, the **State name** box is set to `X1`.

Open the Enable block dialog box. The parameter **States when enabling** is set to `reset`. When the subsystem transitions from a disabled state to an enabled state, it resets internal block states, such as `X1`, to their initial values.

In the base workspace, double-click the `Simulink.Signal` object `X1` to view its properties. The **Initial value** property is set to `aa1`.

Double-click the `Simulink.Parameter` object `aa1` to view its properties. The **Storage class** property is set to `ExportedGlobal`. You can generate tunable initial conditions for block states by using tunable parameters such as `aa1` and `Simulink.Signal` objects such as `X1`.

Generate and Inspect Code

Generate code with the example model.

```
### Starting build procedure for: rtwdemo_sigobj_iv
### Successful completion of build procedure for: rtwdemo_sigobj_iv

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
rtwdemo_sigobj_iv  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.57s
```

In the code generation report, view the file `rtwdemo_sigobj_iv.c`. The code uses global variables to represent the block state `X1` and the signal `S2`.

```
/* Exported block states */
real_T X1;                             /* '<S1>/Unit Delay' */

/* Exported block signals */
real_T S1;                             /* '<Root>/In1' */
real_T S3;                             /* '<Root>/Signal Conversion' */
real_T S2;                             /* '<S1>/Unit Delay' */
```

The code uses global variable to represent the tunable parameter `aa1`.

```
/* Exported block parameters */
real_T aa1 = -2.5;                     /* Variable: aa1
```

The model initialization function uses the tunable parameter `aa1` to initialize the state `X1`. The function also uses the tunable parameter `aa2` to initialize the signal `S2`.

```
  /* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem (state X1 inside)' */
  /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
  X1 = aa1;

  /* SystemInitialize for UnitDelay: '<S1>/Unit Delay' incorporates:
   *  Outport: '<S1>/Out1'
   */
  S2 = 0.0;
```

In the model step function, when the Enabled Subsystem transitions from a disabled state to an enabled state, the Unit Delay block state `X1` resets to its initial value.

```
  if (rtb_PulseGenerator > 0) {
```

```
    if (!rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
      /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
      X1 = aa1;
      rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = true;
    }

    /* UnitDelay: '<S1>/Unit Delay' */
    S2 = X1;

    /* Update for UnitDelay: '<S1>/Unit Delay' incorporates:
     *  Inport: '<Root>/In1'
     */
    X1 = S1;
  } else if (rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
    /* Disable for UnitDelay: '<S1>/Unit Delay' incorporates:
     *  Outport: '<S1>/Out1'
     */
    S2 = 0.0;
    rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = false;
  }

  /* End of Outputs for SubSystem: '<Root>/Enabled Subsystem (state X1 inside)' */

  /* DataStoreWrite: '<Root>/Data Store Write' */
  X2 = S2;

  /* SignalConversion: '<Root>/Signal Conversion' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   */
  S3 = X2;
}

/* Model initialize function */
void rtwdemo_sigobj_iv_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtwdemo_sigobj_iv_M, (NULL));

  /* block I/O */

  /* exported global signals */
  S3 = -3.0;

  /* states (dwork) */
  (void) memset((void *)&rtwdemo_sigobj_iv_DW, 0,
                sizeof(DW_rtwdemo_sigobj_iv_T));

  /* exported global states */
  X1 = 0.0;
  X2 = 0.0;

  /* external inputs */
  S1 = -4.5;

  /* Start for DiscretePulseGenerator: '<Root>/Pulse Generator' */
  rtwdemo_sigobj_iv_DW.clockTickCounter = -2;
```

```
  /* Start for DataStoreMemory: '<Root>/Data Store Memory' */
  X2 = -3.5;

  /* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem (state X1 inside)' */
  /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
  X1 = aa1;

  /* SystemInitialize for UnitDelay: '<S1>/Unit Delay' incorporates:
   *  Outport: '<S1>/Out1'
   */
  S2 = 0.0;

  /* End of SystemInitialize for SubSystem: '<Root>/Enabled Subsystem (state X1 inside)' */
}

/* Model terminate function */
void rtwdemo_sigobj_iv_terminate(void)
{
  /* (no terminate code required) */
}
```

If the Enabled Subsystem becomes disabled during code execution, the algorithm uses the tunable initial condition `aa2` to set the value of the signal `S2`.

```
  } else {

  } else if (rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
    /* Disable for UnitDelay: '<S1>/Unit Delay' incorporates:
     *  Outport: '<S1>/Out1'
     */
    S2 = 0.0;
    rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = false;
  }
```

Generate Tunable Initial Condition Structure for Bus Signal

When you use a MATLAB® structure to specify initialization values for the signal elements in a bus, you can create a tunable global structure in the generated code.

If you set **Configuration Parameters > Optimization > Default parameter behavior** to `Tunable`, the initial condition appears as a tunable substructure of the global parameters structure.

Whether you set **Default parameter behavior** to `Tunable` or `Inlined`, you can specify the initial condition by using a tunable `Simulink.Parameter` object whose value is a structure. If you apply a storage class other than `Auto` to the parameter object, the structure is tunable in the generated code.

To generate efficient code by avoiding data type mismatches between the structure and the bus signal, use either:

- Typed expressions to specify the values of the structure fields. Match the data type of each field with the data type of the corresponding signal element.

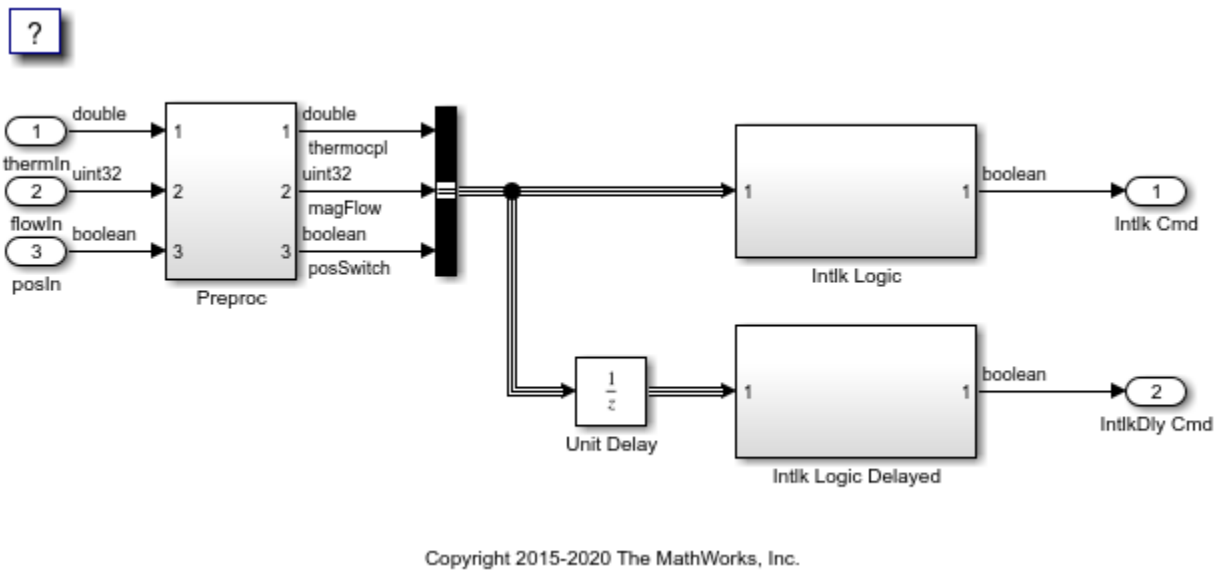
- A `Simulink.Bus` object to control the data types of the structure fields and the signal elements.

For basic information about using structures to initialize bus signals, and to decide how to control field data types, see “Specify Initial Conditions for Bus Elements”.

Generate Tunable Initial Condition Structure

This example shows how to use a tunable structure parameter to initialize a virtual bus signal.

Open the example model `rtwdemo_tunable_init_struct` and configure it to show the generated names of blocks.



On the **Modeling** tab, click **Model Data Editor**.

On the **Inports/Outports** tab, the **Data Types** column shows that each Inport block in the model uses a different output data type.

Open the Bus Creator block dialog box. The block output is a virtual bus.

In the Configuration Parameters dialog box, open the **Optimization** pane. The configuration parameter **Default parameter behavior** is set to `Tunable`. By default, block parameters, including initial conditions, appear in the generated code as tunable fields of the global parameters structure.

In the Model Data Editor, inspect the **States** tab.

For the Unit Delay block, set **Initial Value** to a structure that specifies an initial condition for each of the three signal elements. To generate efficient code, match the data types of the structure fields with the data types of the corresponding signal elements. For example, set **Initial Value** to the expression `struct('thermocpl',15.23,'magFlow',uint32(79),'posSwitch',false)`.

```
set_param('rtwdemo_tunable_init_struct/Unit Delay','InitialCondition',...
'struct(''thermocpl'',15.23,''magFlow'',uint32(79),''posSwitch'',false)')
```

Generate code from the example model.

```
### Starting build procedure for: rtwdemo_tunable_init_struct
### Successful completion of build procedure for: rtwdemo_tunable_init_struct

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_tunable_init_struct  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.124s
```

In the code generation report, view the file `rtwdemo_tunable_init_struct_types.h`. The code defines a structure type whose fields use the data types that you specified in the `struct` expression.

```
#ifndef DEFINED_TYPEDEF_FOR_struct_mqGi1jsItE0G7cf1bNqMu_
#define DEFINED_TYPEDEF_FOR_struct_mqGi1jsItE0G7cf1bNqMu_

typedef struct {
  real_T thermocpl;
  uint32_T magFlow;
  boolean_T posSwitch;
} struct_mqGi1jsItE0G7cf1bNqMu;
```

View the file `rtwdemo_tunable_init_struct.h`. The `struct` type definition of the global parameters structure contains a substructure, `UnitDelay_InitialCondition`, which represents the **Initial condition** parameter of the Unit Delay block.

```
struct P_rtwdemo_tunable_init_struct_T_ {
  struct_mqGi1jsItE0G7cf1bNqMu UnitDelay_InitialCondition;
```

View the file `rtwdemo_tunable_init_struct_data.c`. This source file allocates memory for the global parameters structure. The substructure `UnitDelay_InitialCondition` appears.

```
/* Block parameters (default storage) */
P_rtwdemo_tunable_init_struct_T rtwdemo_tunable_init_struct_P = {
  /* Mask Parameter: UnitDelay_InitialCondition
   * Referenced by:
   *   '<Root>/Unit Delay'
   *   '<Root>/Unit Delay'
   *   '<Root>/Unit Delay'
   */
  {
    15.23,
    79U,
    0
  },
```

View the file `rtwdemo_tunable_init_struct.c`. The model initialization function uses the fields of the substructure to initialize the block states.

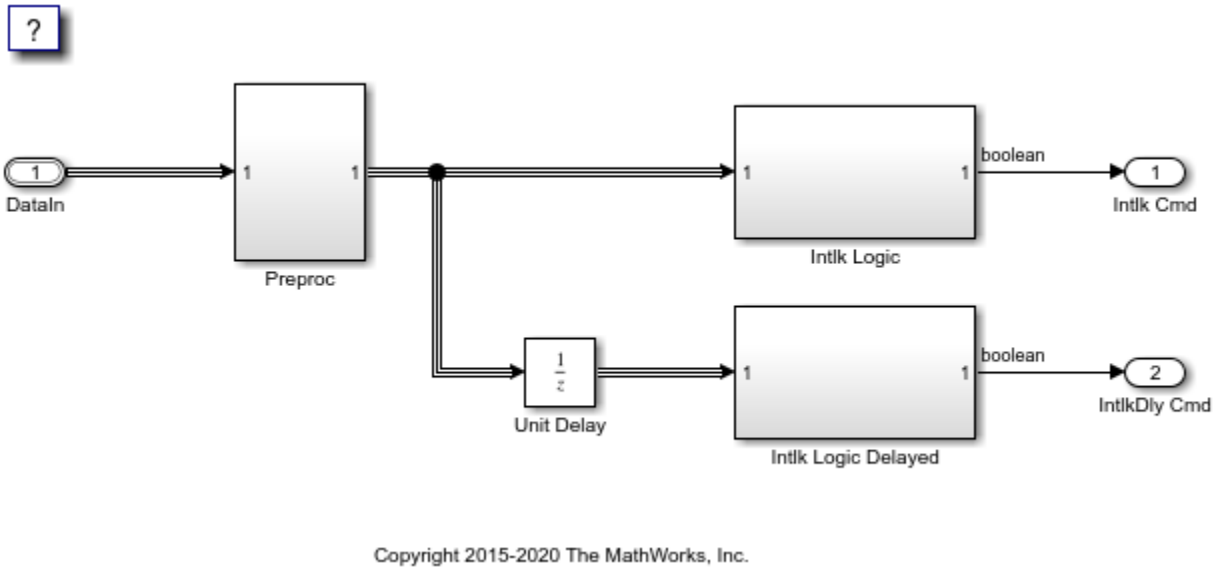
```
  /* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
```

```
rtwdemo_tunable_init_struct_DW.UnitDelay_1_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.thermocpl;

/* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_2_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.magFlow;

/* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_3_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.posSwitch;
```

Use Bus Object to Specify Data Types

If you create a bus object, you can use it to specify the data type of the bus signal and the tunable initial condition structure. Before code generation, the `Simulink.Parameter` object casts the values of the structure fields to the data types of the signal elements. For basic information about creating bus objects and using them in models, see “Specify Bus Properties with Simulink.Bus Object Data Types”.

Open the example model `rtwdemo_init_struct_busobj` and configure it to show the generated names of blocks.



In the base workspace, double-click the `Simulink.Bus` object `ComponentData`. The object defines three signal elements: `thermocpl`, `magFlow`, and `posSwitch`. The elements each use a different data type.

In the model, open the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). The **Inports/Outports** tab shows that for the Inport block `DataIn`, the output data type (**Data Type** column) is set to `Bus: ComponentData`.

At the command prompt, create the structure parameter `initStruct`. You can specify the field values by using untyped expressions. To improve readability, specify the field `posSwitch` with a Boolean value.

```
initStruct = struct(...
    'thermocpl',15.23,...
    'magFlow',79,...
    'posSwitch',false ...
    );
```

```
initStruct = Simulink.Parameter(initStruct);
```

In the Model Data Editor, inspect the **Parameters** tab.

In the model, click the Unit Delay block. The Model Data Editor highlights the row that corresponds to the **Initial condition** parameter of the block.

In the Model Data Editor, set the parameter value (**Value** column) to `initStruct`.

Click the **Show/refresh additional information** button. The parameter object, `initStruct`, appears in the data table as a row.

Use the **Data Type** column to set the data type of `initStruct` to `Bus: ComponentData`.

```
initStruct.DataType = 'Bus: ComponentData';
```

Set the **Change View** drop-down list to `Code`.

Use the **Storage Class** column to apply the storage class `ExportedGlobal` to `initStruct`.

```
initStruct.StorageClass = 'ExportedGlobal';
```

Generate code from the example model.

```
### Starting build procedure for: rtwdemo_init_struct_busobj
### Successful completion of build procedure for: rtwdemo_init_struct_busobj

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
==============================================================================================
rtwdemo_init_struct_busobj  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.224s
```

In the code generation report, view the file `rtwdemo_init_struct_busobj_types.h`. The code creates a structure type `ComponentData` whose fields use the data types in the bus object.

```
#ifndef DEFINED_TYPEDEF_FOR_ComponentData_
#define DEFINED_TYPEDEF_FOR_ComponentData_

typedef struct {
  real_T thermocpl;
  uint32_T magFlow;
  boolean_T posSwitch;
} ComponentData;
```

View the file `rtwdemo_init_struct_busobj.c`. The code creates a global variable to represent the tunable parameter object `initStruct`.

```
/* Exported block parameters */
ComponentData initStruct = {
  15.23,
  79U,
  0
} ;                                   /* Variable: initStruct
```

The model initialization function uses the structure fields to initialize the block states.

```
  /* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
  rtwdemo_init_struct_busobj_DW.UnitDelay_1_DSTATE = initStruct.thermocpl;

  /* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
  rtwdemo_init_struct_busobj_DW.UnitDelay_2_DSTATE = initStruct.magFlow;

  /* InitializeConditions for UnitDelay generated from: '<Root>/Unit Delay' */
  rtwdemo_init_struct_busobj_DW.UnitDelay_3_DSTATE = initStruct.posSwitch;
```

To change the data type of a signal element, specify the new type in the bus object. The signal element in the model uses the new type. Before simulation and code generation, the parameter object `initStruct` casts the corresponding structure field to the new type.

See Also

State Reader | State Writer

Related Examples

- “Initialization Behavior Summary for Signal Objects”
- “Specify Initial Conditions for Bus Elements”
- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Startup, Reset, and Shutdown Function Interfaces” on page 3-10
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Initialization of Signal, State, and Parameter Data in the Generated Code” on page 36-187
- “Remove Zero-Initialization Code” on page 79-4

Initialization of Signal, State, and Parameter Data in the Generated Code

Signal lines, block parameters, and block states in a model can appear in the generated code as data, for example, as global variables. By default, the code initializes this data before execution of the primary algorithm. To match the numerics of simulation in Simulink, the code generator chooses the initial values based on specifications that you make in the model.

By understanding how the generated code initializes data, you can:

- Model a system that reinitializes states during execution, which means that the application can restart the entire system.
- Store initial values in memory as variables, which can persist between execution runs. You can then overwrite these values before starting or restarting a system.
- Generate efficient code by eliminating storage for invariant initial values and by preventing the generation of code that unnecessarily or redundantly initializes data.

For basic information about initializing signals and states in a model, see “Initialize Signals and Discrete States” and “Load State Information”.

Static Initialization and Dynamic Initialization

To initialize a data item such as a global variable, an application can use static or dynamic initialization.

- Static initialization occurs in the same statement that defines (allocates memory for) the variable. The initialization does not occur inside a function definition.

  The code can be more efficient because none of the generated model functions execute initialization statements.
- Dynamic initialization occurs inside a function. For each model or nonvirtual subsystem, the code generator typically creates one or more functions that are dedicated to initialization.

  The generated code or your code can restart a system by calling an initialization function during execution.

Real-World Ground Initialization Requiring Nonzero Bit Pattern

Each data item has a real-world ground value. This value can depend on the data type of the item. For example, for a signal whose data type is `double` or `int8`, the real-world ground value is zero. For an enumeration, the ground value is the default enumeration member.

For some kinds of data, to represent a real-world ground value, a computer stores zero in memory (all bits zero). However, for some other data, a computer stores a nonzero value in memory. This data includes, for example:

- Fixed-point data with bias. The code initializes such a data item to the stored integer value that, given the scaling and bias, represents real-world zero.
- An enumeration whose default member maps to an integer value other than zero. For example, if the default member is `High` with an underlying integer value of 3, the code initializes such a data item to `High`.

Initialization of Signal and State Data

In a model, you can control signal and state initial values through block parameters. For example, to set the initial value of a Unit Delay block, you use the **Initial condition** parameter. In some cases, you can also use the `InitialValue` property of a `Simulink.Signal` object. For most of these block parameters, the default value is `0`.

You can also initialize states by using State Writer blocks in Initialize Function subsystems and the **Initial states** model configuration parameter.

By default, the generated code dynamically initializes signal and state data (and other data, such as the model error status) in the generated initialization function. The function, named *`model`*`_initialize` by default, performs signal and state initialization operations in this order:

1. Initializes the signal and state data in the default generated structures, such as the DWork structure, to a stored value of zero.
2. Initializes additional signal and state data that are not in the default generated structures to the relevant real-world ground value.

   This initialization applies only to data that meet both of these criteria:

   - The generated code defines (allocates memory for) the data.
   - The data use a storage class other than `Auto` (the default storage class) or `Model default` (when the Code Mapping Editor specifies the `Default` storage class, the default setting).

   For example, the code applies this operation to data items that use the storage class `ExportedGlobal`.
3. Initializes each signal and state to the real-world value that the model specifies, for example, through the **Initial condition** parameter of a Unit Delay block.
4. Initializes each state to the real-world value that you assign by using a State Writer block. The function performs this initialization only if you use an Initialize Function subsystem in the model.
5. Initializes each state to the real-world value that you specify with the configuration parameter **Initial states**.

After this initialization function executes, each data item has the last real-world value that the function assigned. For example, if you use a State Writer block to initialize a block state to `5` while also using the **Initial states** configuration parameter to initialize the same state to `10`, the state ultimately uses `10` as an initial value.

memset for Bulk Initialization

To initialize signal or state data items that have contiguous storage to a stored value of zero, the generated code can call `memset` in an initialization function. Data items that have contiguous storage include the DWork structure, arrays, or data items that use a multiword data type.

If your application requires it, you can prevent the generated code from using `memset` for initializing floating-point data to stored zero. See Use memset to initialize floats and doubles to 0.0.

Tunable Initial Values

You can configure the way that tunable block parameters, such as the **Gain** parameter of a Gain block, appear in the generated code. Most block parameters that set initial values (for example,

Initial condition) are tunable. For example, the configuration parameter **Default parameter behavior** can determine whether the initial values appear in the generated code as inlined constants or as tunable global data. You can also use parameter objects and storage classes to control the representation of these initial values.

For tunable initial values, in the model initialization function, the right-hand side of the assignment statement is a global variable, structure field, or other data whose value you can change in memory.

To make initial values tunable in the generated code, see “Control Signal and State Initialization in the Generated Code” on page 36-176.

Initialization of Parameter Data

The generated code statically initializes parameter data to the values that you specify in Simulink.

To express the initial value of a tunable parameter (a global variable) as a mathematical expression involving system constants or other macros, which requires Embedded Coder, see “Initialization by Mathematical Expression (Embedded Coder)”.

Kinds of Parameter Data

These model elements appear in the generated code as parameter data:

- Tunable block parameters, such as the **Gain** parameter of a Gain block, when you set **Default parameter behavior** to `Tunable`. Each parameter appears as a field of a dedicated global structure.
- Some tunable block parameters when you set **Default parameter behavior** to `Inlined`. When the code generator cannot inline the value of a parameter as a literal number, the parameter appears as a field of a dedicated global `const` structure.
- `Simulink.Parameter` objects to which you apply a storage class that has an exported data scope. For example, the built-in storage class `ExportedGlobal` has an exported data scope, but the storage class `ImportedExtern` does not.

Initialization by Mathematical Expression (Embedded Coder)

You can set the value of a parameter object (such as `Simulink.Parameter`) to a mathematical expression involving numbers, MATLAB variables, and other parameter objects. See “Set Variable Value by Using a Mathematical Expression”.

If you use this technique with Embedded Coder, you can generate code that initializes the corresponding parameter data (global variable) by using the specified expression. For an example, see “Initialize Parameter Value From System Constant or Other Macro (Embedded Coder)” on page 36-118. For general information, including limitations, see “Code Generation of Parameter Objects With Expression Values”.

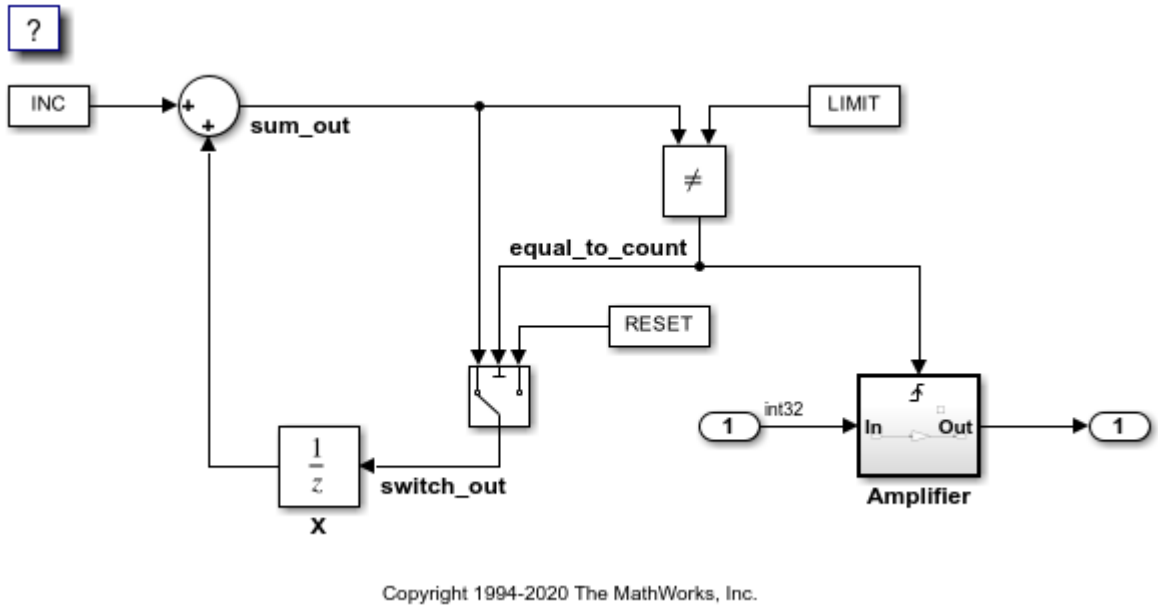
Data Initialization in the Generated Code

This example shows how the generated code initializes signal, state, and parameter data.

Explore Example Model

Open the example model, `rtwdemo_rtwintro`.

```
open_system('rtwdemo_rtwintro')
```



In the Apps gallery, click **Simulink Coder**.

Open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**.

Inspect the **States** tab. For the Unit Delay block state, the **Initial Value** is set to `0`, the default, which means the initial value of the state is zero. The state is named `X`.

Set **Initial Value** to a nonzero number, for example, `5`.

```
set_param('rtwdemo_rtwintro/X','InitialCondition','5')
```

Open the Code Mappings editor. In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**.

On the **Signals/States** tab, under **States** set **Storage Class** for `X` to `ExportedGlobal`. With this setting, the block state appears in the generated code as a separate global variable.

```
coder.mapping.utils.create('rtwdemo_rtwintro');
cm = coder.mapping.api.get('rtwdemo_rtwintro');
setState(cm,'rtwdemo_rtwintro/X','StorageClass','ExportedGlobal');
```

In the model, open the `Amplifier` subsystem, which is a triggered subsystem.

In the Model Data Editor, inspect the **Parameters** tab. To set and inspect initial values of signals and states, you can use the **Parameters** tab instead of the **States** tab.

The Model Data Editor shows that for the Outport block, the **Initial output** (`InitialOutput`) parameter is set to `0`, the default. This subsystem output requires an initial value because the subsystem executes conditionally. Leave the initial value at the default.

In the model, clear **Configuration Parameters > Signal storage reuse**. When you clear this setting, signal lines appear in the generated code as fields of a generated structure whose purpose is

to store signal data. This representation of the signals makes it easier to see how the code generator initializes data.

```
set_param('rtwdemo_rtwintro','OptimizeBlockIOStorage','off')
```

In the model, inspect the configuration parameter **Configuration Parameters > Optimization > Default parameter behavior**. The configuration parameter is set to `Inlined`, which means block parameters, including initial values, appear in the generated code as inlined literals or as `const` data.

Navigate to the root level of the model.

In the Model Data Editor, click the **Show/refresh additional information** button.

Near the **Filter contents** box, click the **Filter using selection** button.

In the model, click the Constant block labeled `INC`. The Model Data Editor shows that the **Constant value** (`Value`) parameter of the block is set to `INC`. `INC` is a MATLAB variable in the base workspace.

For `INC`, set the value in the **Value** column to `Simulink.Parameter(uint8(1))`. MATLAB converts `INC` to a `Simulink.Parameter` object.

In the Code Mappings editor, on the **Parameters** tab, apply the storage class `ExportedGlobal` to `INC`. With this setting, the generated code defines `INC` as a global variable. Concerning initialization, `INC` is an item of parameter data.

```
INC = Simulink.Parameter(INC);
INC.StorageClass = 'ExportedGlobal';
```

Generate and Inspect Code

To build the model and generate code, press **Ctrl+B** .

```
slbuild('rtwdemo_rtwintro');
```

```
### Starting build procedure for: rtwdemo_rtwintro
### Successful completion of build procedure for: rtwdemo_rtwintro

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
========================================================================================
rtwdemo_rtwintro  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.449s
```

In the generated file `rtwdemo_rtwintro.c`, outside the definition of a function, the code statically initializes the parameter data `INC`. The value of `INC` is `1`.

```
file = fullfile('rtwdemo_rtwintro_grt_rtw','rtwdemo_rtwintro.c');
rtwdemodbtype(file,'/* Exported block parameters */','uint8_T INC = 1U;',1,1)
```

```
/* Exported block parameters */
uint8_T INC = 1U;                      /* Variable: INC
```

In the same file, inspect the definition of the `rtwdemo_rtwintro_initialize` function. First, the function uses `memset` to initialize the internal signals in the model to a stored value of `0`.

```
rtwdemodbtype(file,'/* block I/O */','sizeof(B_rtwdemo_rtwintro_T));',1,1)
```

```
/* block I/O */
(void) memset(((void *) &rtwdemo_rtwintro_B), 0,
              sizeof(B_rtwdemo_rtwintro_T));
```

The function then initializes the Unit Delay state, `X`, to a ground value. In this case, the ground value is zero.

```
rtwdemodbtype(file,'/* exported global states */','X = 0U;',1,1)
```

```
/* exported global states */
X = 0U;
```

The function also initializes other data, including the root-level inputs and outputs (Inport and Outport blocks), to ground values.

```
rtwdemodbtype(file,'/* external inputs */','Amplifier_Trig_ZCE = POS_ZCSIG;',1,1)
```

```
/* external inputs */
rtwdemo_rtwintro_U.Input = 0;

/* external outputs */
rtwdemo_rtwintro_Y.Output = 0;
rtwdemo_rtwintro_PrevZCX.Amplifier_Trig_ZCE = POS_ZCSIG;
```

The function then initializes `X` to the value that you specified in the **Initial condition** block parameter.

```
rtwdemodbtype(file,'/* InitializeConditions for UnitDelay: ''<Root>/X'' */',...
    'X = 5U;',1,1)
```

```
/* InitializeConditions for UnitDelay: '<Root>/X' */
X = 5U;
```

Finally, the function initializes the output of the `Amplifier` subsystem.

```
rtwdemodbtype(file,'SystemInitialize for Triggered SubSystem',...
    'End of SystemInitialize for SubSystem',1,1)
```

```
/* SystemInitialize for Triggered SubSystem: '<Root>/Amplifier' */
/* SystemInitialize for Outport: '<Root>/Output' incorporates:
 *  Outport: '<S1>/Out'
 */
rtwdemo_rtwintro_Y.Output = 0;
```

Inspect Difference Between Stored Value and Real-World Value

For some data, even if the real-world initial value is zero, a computer stores a nonzero value in memory. To observe this difference, apply a slope-bias fixed-point data type to the block state, X. To run this example, you must have Fixed-Point Designer™.

In the Model Data Editor, inspect the **Signals** tab.

In the model, click the output signal of the Switch block, `switch_out`.

Use the **Data Type** column of the Model Data Editor to set the signal data type to `fixdt(1,16,1,3)`. This expression represents a fixed-point data type with a slope of 1 and a bias of 3.

```
set_param('rtwdemo_rtwintro/Switch','OutDataTypeStr',...
    'fixdt(1,16,1,3)')
```

In the model, select the Sum block. In the Property Inspector, under **Signal Attributes**, clear **Require all inputs to have the same data type**.

```
set_param('rtwdemo_rtwintro/Sum','InputSameDT','off')
```

To prevent compilation errors on different platforms, select the model configuration parameter **Generate code only**. This setting causes the model to generate only code.

```
set_param('rtwdemo_rtwintro','GenCodeOnly','on')
```

To build the model and generate code, press **Ctrl+B** .

```
slbuild('rtwdemo_rtwintro')
```

```
### Starting build procedure for: rtwdemo_rtwintro
### Successful completion of code generation for: rtwdemo_rtwintro

Build Summary

Top model targets built:

Model             Action           Rebuild Reason
========================================================================
rtwdemo_rtwintro  Code generated.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.933s
```

The model initialization function first initializes X to a real-world ground value, 0. Due to the slope-bias data type, which has bias 3, this real-world value corresponds to a stored value of -3.

```
rtwdemodbtype(file,'/* exported global states */','X = -3;',1,1)
```

```
  /* exported global states */
  X = -3;
```

The function then initializes X to the real-world initial value that you specified in the **Initial condition** block parameter, 5. This real-world value corresponds to a stored value of 2.

```
rtwdemodbtype(file,'/* InitializeConditions for UnitDelay: ''<Root>/X'' */',...
    'X = 2;',1,1)
```

```
/* InitializeConditions for UnitDelay: '<Root>/X' */
X = 2;
```

Modeling Goals

Goal	More Information
Explicitly model initialization behavior by using blocks	You can explicitly model initialization and reset behavior by using Initialize Function and Reset Function subsystems. In the subsystems, use State Writer blocks to calculate and assign an initial value for a state dynamically. The corresponding code appears in the model initialization function. For more information, see “Startup, Reset, and Shutdown Function Interfaces”.
Prevent generation of code that explicitly initializes data to zero	If your application environment already initializes global variables to zero, for more efficient code, you can prevent the generation of statements that explicitly initialize global variables to zero. This optimization applies only to signals and states whose initial values are stored in memory as zero. For example, the code generator does not apply the optimization to: • Data for which you specify a nonzero initial value by using a block parameter. • Data whose real-world initial value is zero but whose corresponding stored value is not zero. • Enumerated data whose default member maps to a nonzero integer. The optimization requires Embedded Coder. For more information, see “Remove Zero-Initialization Code” on page 79-4.
Generate code that imports data from your external code	You can generate code that reuses (imports) data that your external code defines. For example, you can apply the storage class `ImportedExtern` to a signal line, block state, or parameter object. For imported data: • The generated code does not initialize parameter data. Your code must initialize imported parameter data. • The generated initialization functions dynamically initialize some signal and state data. Unlike data that the generated code allocates, the code does not initialize imported signal or state data to a stored value of zero. Instead, the code immediately initializes the data to the real-world value that you specify in Simulink. For more information, including how to prevent the generated code from initializing imported data, see “Exchange Data Between External Calling Code and Generated Code”.

See Also

`model_initialize`

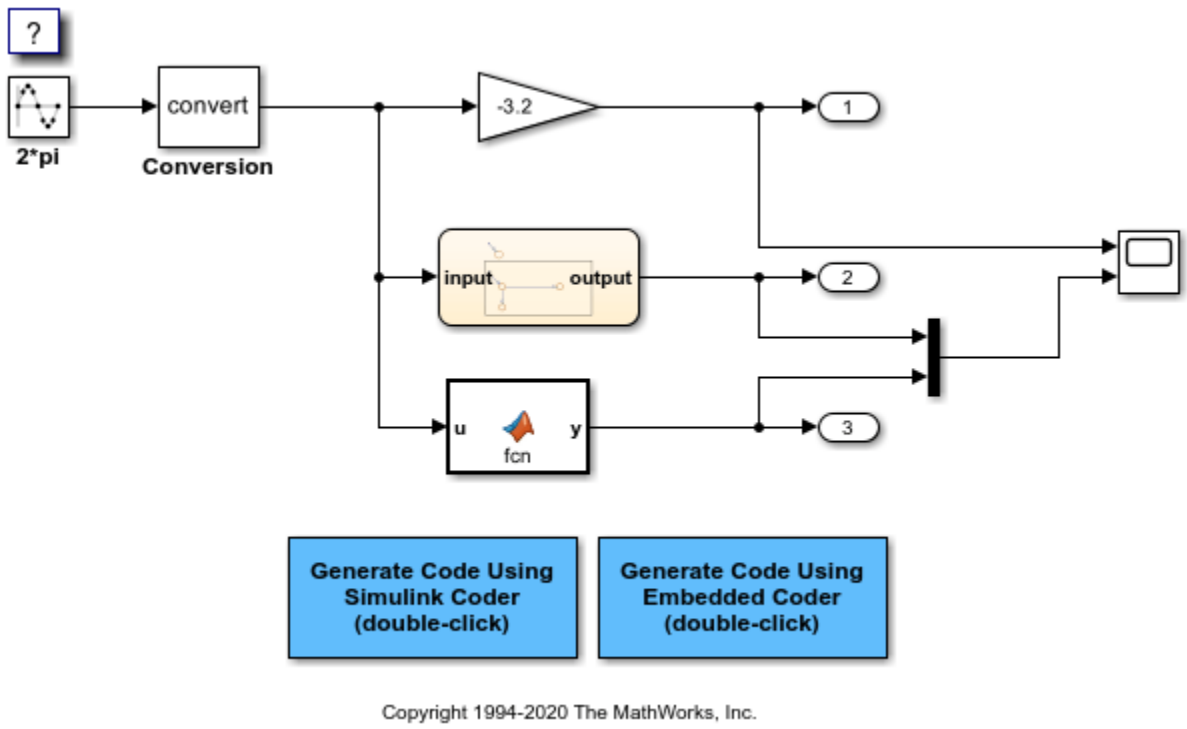
Related Examples

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

Optimize Generated Code Using Fixed-Point Data with Simulink, Stateflow, and MATLAB

This model shows fixed-point code generation in Simulink®, Stateflow®, and MATLAB®.

```
open_system ('rtwdemo_fixpt1');
```



See Also

More About

- “Fixed Point”

Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box

You can use the Model Parameter Configuration dialog box to declare numeric MATLAB variables in the base workspace as tunable parameters. You can select code generation options, such as storage class, for each tunable parameter.

However, it is a best practice to instead use parameter objects to declare tunable parameters. Do not use the Model Parameter Configuration dialog box to select parameter objects in the base workspace. To use parameter objects, instead of the Model Parameter Configuration dialog box, to declare tunable parameters, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Note You cannot use the Model Parameter Configuration dialog box to declare tunable parameters for a referenced model. Use `Simulink.Parameter` objects instead.

Declare Existing Workspace Variables as Tunable Parameters

Use the Model Parameter Configuration dialog box to declare existing workspace variables as tunable parameters for a model.

1. In the Configuration Parameters dialog box, on the **Code Generation** > **Optimization** pane, set the **Default parameter behavior** parameter to `Tunable` and click **Configure**.
2. In the Model Parameter Configuration dialog box, under **Source list**, select a method to populate the list of available workspace variables.
   - Select `MATLAB workspace` to view numeric variables that are defined in the base workspace.
   - Select `Referenced workspace variables` to view only the numeric variables in the base workspace that the model uses. Selecting this option begins a diagram update and a search for used variables, which can take time for a large model.
3. In the Model Parameter Configuration dialog box, under **Source list**, select one or more workspace variables.
4. Click **Add to table**. The variables appear as tunable parameters under **Global (tunable) parameters**, and appear in italic font under **Source list**.
5. Optionally, select a parameter under **Global (tunable) parameters**, and adjust the code generation settings for the parameter. For more information about adjusting the code generation options for tunable parameters, see “Set Tunable Parameter Code Generation Options” on page 36-198
6. Click **OK** to apply your selection of tunable parameters and close the dialog box.

Declare New Tunable Parameters

Use the Model Parameter Configuration dialog box to declare new tunable parameters. You can use this technique to declare the names of tunable parameters, and to adjust their code generation settings, before you create the corresponding workspace variables.

1. In the Configuration Parameters dialog box, on the **Optimization** pane, set the **Default parameter behavior** parameter to `Tunable` and click **Configure**.

2. In the Model Parameter Configuration dialog box, under **Global (tunable) parameters**, click **New**.
3. Under the **Name** column, specify a name for the new tunable parameter.
4. Optionally, adjust the code generation settings for the new parameter. For more information about adjusting the code generation options for tunable parameters, see “Set Tunable Parameter Code Generation Options” on page 36-198
5. Click **OK** to apply your changes and close the dialog box.

Set Tunable Parameter Code Generation Options

To set the properties of tunable parameters listed under **Global (tunable) parameters** in the Model Parameter Configuration dialog box, select a parameter and specify a storage class and, optionally, a storage type qualifier.

Property	Description
Storage class	Select one of the following to use for code generation: • `Model default` • `ExportedGlobal` • `ImportedExtern` • `ImportedExternPointer` For more information about tunable parameter storage classes, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.
Storage type qualifier	For variables with a storage class other than `Auto`, you can add a qualifier (such as `const` or `volatile`) to the generated storage declaration. To do so, you can select a predefined qualifier from the list, or add qualifiers not in the list by typing them in. The code generator does not check the storage type qualifier for validity, and includes the qualifier text in the generated code without checking syntax .

Programmatically Declare Workspace Variables as Tunable Parameters

Tune Parameters from the Command Line

When parameters are MATLAB workspace variables, the Model Parameter Configuration dialog box is the recommended way to see or set the properties of tunable parameters. In addition to that dialog box, you can also use MATLAB `get_param` and `set_param` commands.

Note You can also use `Simulink.Parameter` objects for tunable parameters. See “Create Tunable Calibration Parameter in the Generated Code” on page 36-112 for details.

The following commands return the tunable parameters and corresponding properties:

- `get_param(gcs,'TunableVars')`

- `get_param(gcs,'TunableVarsStorageClass')`
- `get_param(gcs,'TunableVarsTypeQualifier')`

The following commands declare tunable parameters or set corresponding properties:

- `set_param(gcs,'TunableVars',str)`

  The argument `str` (character vector) is a comma-separated list of variable names.
- `set_param(gcs,'TunableVarsStorageClass',str)`

  The argument `str` (character vector) is a comma-separated list of storage class settings.

  The valid storage class settings are

  - `Auto`
  - `ExportedGlobal`
  - `ImportedExtern`
  - `ImportedExternPointer`
- `set_param(gcs,'TunableVarsTypeQualifier',str)`

  The argument `str` (character vector) is a comma-separated list of storage type qualifiers.

The following example declares the variable `k1` to be tunable, with storage class `ExportedGlobal` and type qualifier `const`. The number of variables and number of specified storage class settings must match. If you specify multiple variables and storage class settings, separate them with a comma.

```
set_param(gcs,'TunableVars','k1')
set_param(gcs,'TunableVarsStorageClass','ExportedGlobal')
set_param(gcs,'TunableVarsTypeQualifier','const')
```

Share Data Between Code Generated from Simulink, Stateflow, and MATLAB

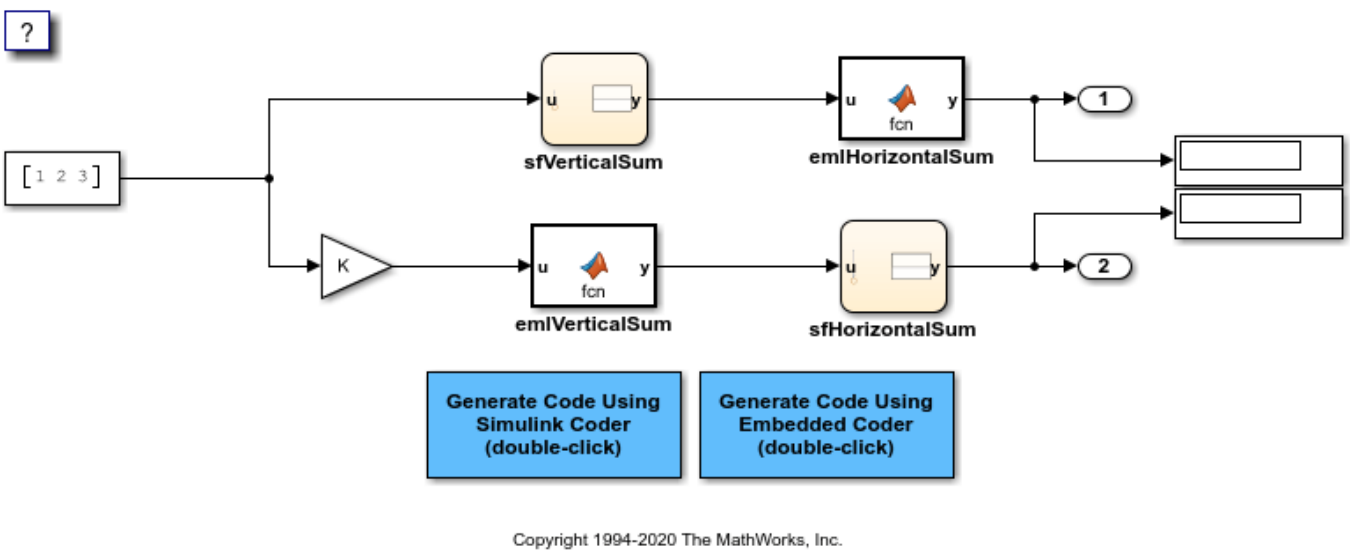
Stateflow® and MATLAB® Coder™ can fully define their data definitions, or they can inherit them from Simulink®. Data definition capabilities include:

- Inheriting input/output data types and sizes from Simulink.
- Parameterized data types and sizes. That is, data type and size may be specified as a function of another data's type and size, e.g., `type(y)=type(u)` and `size(y)=size(u)`.
- Inferred output size and type from Simulink via signal attribute back propagation.
- Parameter scoped data, which allows referencing Simulink parameters in Stateflow and MATLAB.

Open Example Model

Open the example model `rtwdemo_dynamicio`.

```
open_system('rtwdemo_dynamicio')
```



Instructions

1. Compile the model (on the **Simulation** tab, under **Prepare**, click **Update Model**) and note the displayed signal types and sizes.
2. Change the data type and/or size of the Constant block and recompile the model. Note that the attributes of the signals automatically adapt to the Constant block specification.
3. Generate and inspect code using the blue buttons in the model. Note that K is shared by the Gain and sfVerticalSum block.

Notes

- The data type and size of Stateflow and MATLAB data is inherited from Simulink.

- The Gain block and the Stateflow chart sfVerticalSum share the Simulink parameter `K`, which is defined in the MATLAB workspace as a Simulink.Parameter with `Model default` storage class (i.e., `rtP.K` in the generated code).

Air-Fuel Ratio Control System with Fixed-Point Data

This example shows how to generate and optimize the code for a fixed-point air-fuel ratio control system designed with Simulink® and Stateflow®. For a detailed explanation of the model, see:

- "Model a Fault-Tolerant Fuel Control System"
-

The example uses the Embedded Coder® system target file (`ert.tlc`).

Relevant Portions of the Model

Figures 1-4 show relevant portions of the `sldemo_fuelsys` model, which is a closed-loop system containing a plant subsystem and a controller subsystem. The plant allows engineers to validate the controller through simulation early in the design cycle. In this example, generate code for the relevant controller subsystem, `fuel_rate_control`. Figure 1 shows the top-level simulation model.

```
% Open |sldemo_fuelsys|, set the parameters and update the model diagram
% to view the signal data types.
close_system('sldemo_fuelsys',0)
load_system('sldemo_fuelsys');
rtwconfiguredemo('sldemo_fuelsys','ERT','fixed');
sldemo_fuelsys_data('sldemo_fuelsys','switch_data_type','fixed');
sldemo_fuelsys_data('sldemo_fuelsys','top_level_logging','on');
set_param('sldemo_fuelsys','ShowPortDataTypes','on');
set_param('sldemo_fuelsys','SampleTimeColors','on');
set_param('sldemo_fuelsys','Dirty','off');
sldemo_fuelsys([],[],[],'compile');
sldemo_fuelsys([],[],[],'term');
```

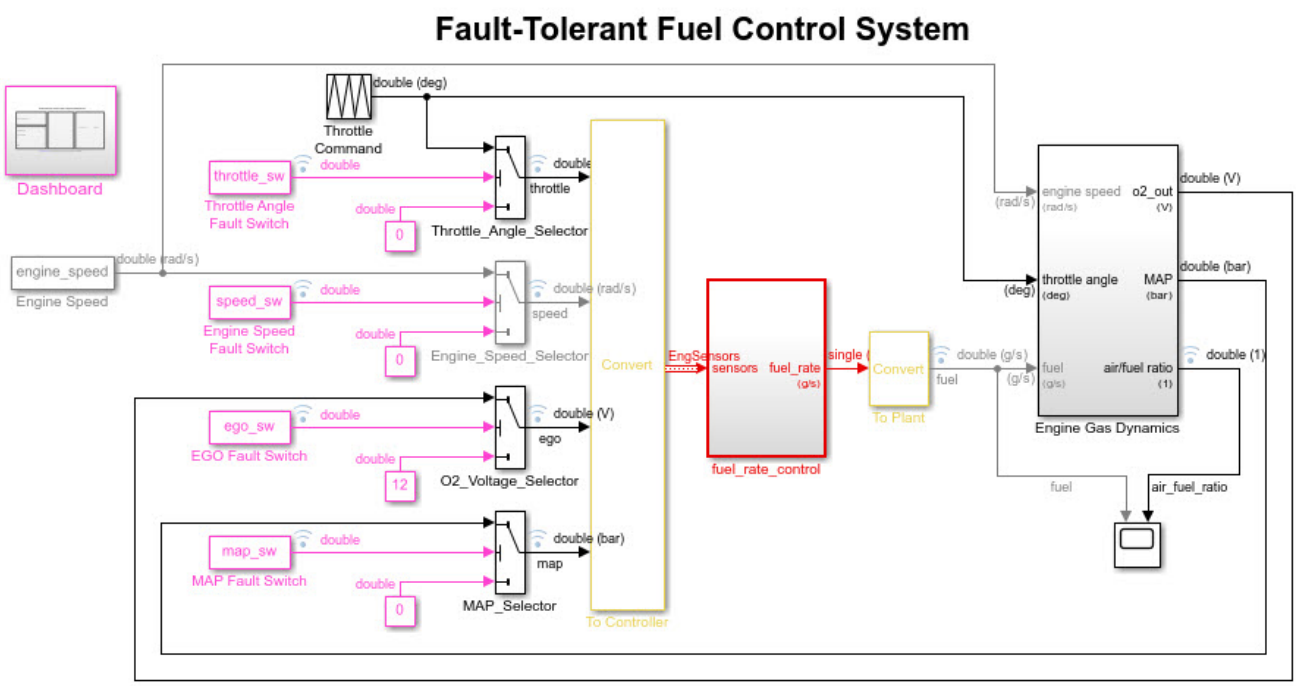


Figure 1: Top-level model of the plant and controller

The air-fuel ratio control system is composed of Simulink and Stateflow blocks. It is the portion of the model for which to generate code.

```
open_system('sldemo_fuelsys/fuel_rate_control');
```

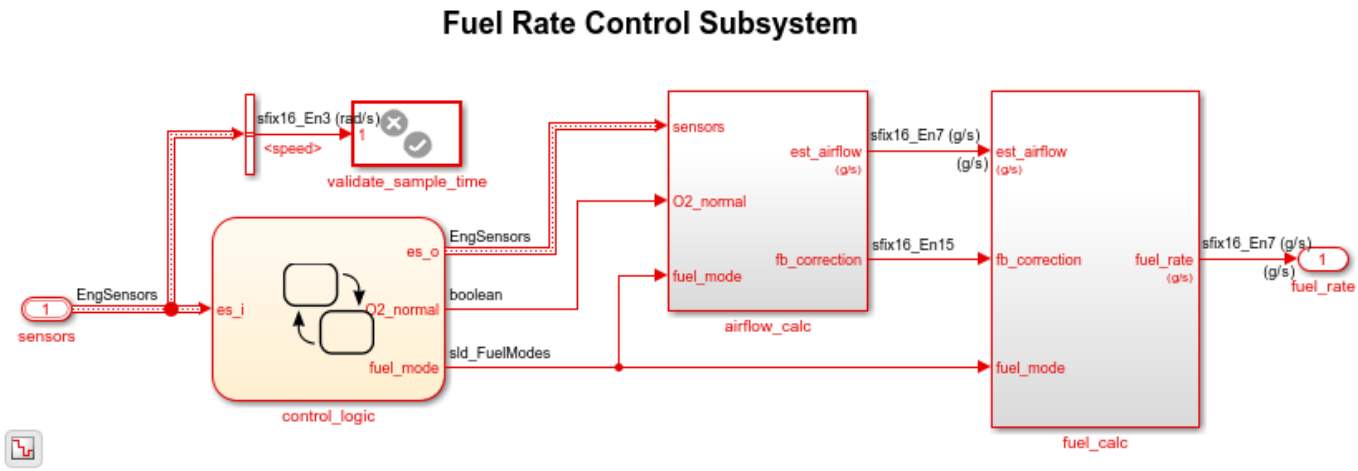


Figure 2: The air-fuel ratio controller subsystem

The intake airflow estimation and closed loop correction system contains two lookup tables, Pumping Constant and Ramp Rate Ki.

```
open_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
```

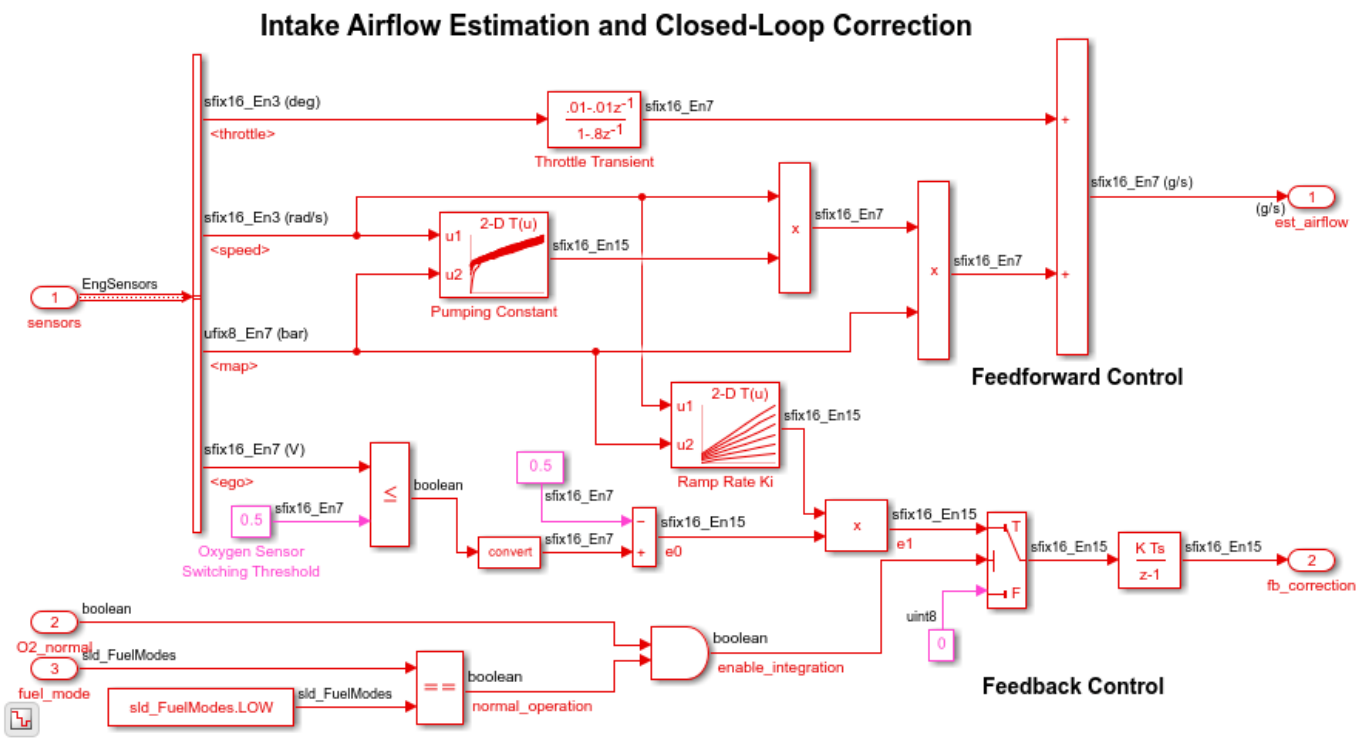


Figure 3: The airflow_calc subsystem

The control logic is a Stateflow chart that specifies the different modes of operation.

```
open_system('sldemo_fuelsys/fuel_rate_control/control_logic');
```

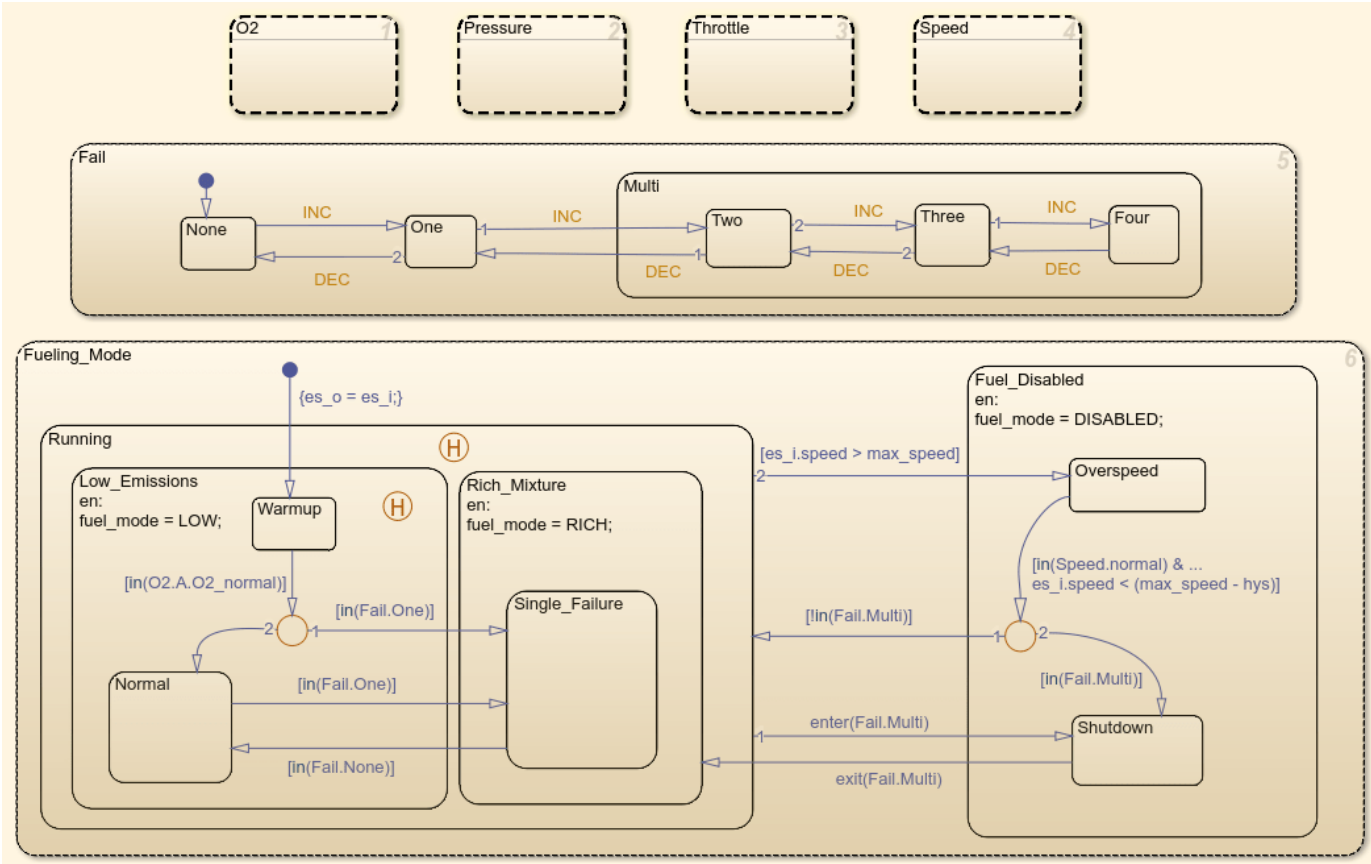


Figure 4: Fuel ratio controller logic

Remove the window clutter.

```
close_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
close_system('sldemo_fuelsys/fuel_rate_control/fuel_calc');
close_system('sldemo_fuelsys/fuel_rate_control/control_logic');
hDemo.rt=sfroot;hDemo.m=hDemo.rt.find('-isa','Simulink.BlockDiagram');
hDemo.c=hDemo.m.find('-isa','Stateflow.Chart','-and','Name','control_logic');
hDemo.c.visible=false;
close_system('sldemo_fuelsys/fuel_rate_control');
```

Build the air-fuel ratio control system only. Once the code generation process is complete, an HTML report detailing the generated code is displayed. The main body of the code is located in `fuel_rate_control.c`.

```
slbuild('sldemo_fuelsys/fuel_rate_control');
```

```
### Starting build procedure for: fuel_rate_control
### Successful completion of build procedure for: fuel_rate_control
```

```
Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
fuel_rate_control  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 24.845s
```

Figure 5 shows snippets of the generated code for the lookup table Pumping Constant.

To see the code for Pumping Constant, right-click the block and select **C/C++ Code > Navigate To C/C++ Code**.

```
rtwtrace('sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant');
```

The code for the pumping constant contains two breakpoint searches and a 2D interpolation. The `SpeedVect` breakpoint is unevenly spaced, and while the `PressVect` breakpoint is evenly spaced, neither have power of two spacing. The current spacing leads to extra code (ROM), including a division, and requires all breakpoints to be in memory (RAM).

```
/* Lookup_n-D: '<S2>/Pumping Constant' */
bpIdx = plook_u32s16u32n31_linca_s(rtB.es_o.speed, &rtConstP.pooled2[0], 17U, &frac);
fractions[0U] = frac;
bpIndices[0U] = bpIdx;
bpIdx = plook_u32u8u32n31_linca_s(rtb_map, &rtConstP.pooled8[0], 18U, &frac);
fractions[1U] = frac;
bpIndices[1U] = bpIdx;
rtb_Product2 = intrp2d_is16s32_u32u32n31_la_s(bpIndices, fractions,
  &rtConstP.PumpingConstant_tab[0], 18U, &rtConstP.pooled1[0]);
```

```
uint32_T plook_u32s16u32n31_linca_s(int16_T u, const int16_T bp[], uint32_T
  maxIndex, uint32_T *fraction)
{
  uint32_T bpIndex;
  int16_T bpLeftVar;

  /* Prelookup - Index and Fraction
     Index Search method: 'linear'
     Process out of range input: 'Clip to range'
     Use previous index: 'off'
     Use last breakpoint for index at or above upper limit: 'on'
     Rounding mode: 'simplest'
   */
  if (u < bp[0U]) {
    bpIndex = 0U;
    *fraction = 0U;
  } else if (u < bp[maxIndex]) {
    bpIndex = linsearch_u32s16(u, bp, (maxIndex + 1U) >> 1U);
    bpLeftVar = bp[bpIndex];
    *fraction = div_nzp_repeat_u32((uint32_T)(uint16_T)((uint32_T)u - (uint32_T)
      bpLeftVar) << 16U, (uint32_T)(uint16_T)(bp[bpIndex + 1U] - bpLeftVar), 15U);
  } else {
    bpIndex = maxIndex;
    *fraction = 0U;
  }

  return bpIndex;
}
```

Division

```
int16_T intrp2d_is16s32_u32u32n31_la_s(const uint32_T bpIndex[], const uint32_T
  frac[], const int16_T table[], uint32_T stride, const uint32_T maxIndex[])
{
  int16_T y;
  uint32_T offset_1d;
  int16_T yL_0d0;
  int16_T intermRes;
  int16_T intermRes_0;

  /* Interpolation 2-D
     Interpolation method: 'Linear'
     Use last breakpoint for index at or above upper limit: 'on'
     Rounding mode: 'simplest'
     Overflow mode: 'wrapping'
   */
  offset_1d = bpIndex[1] * stride + bpIndex[0];
  if (bpIndex[0] == maxIndex[0]) {
    yL_0d0 = table[offset_1d];
  } else {
    yL_0d0 = table[offset_1d];
    intermRes = (int16_T)mul_s32_s32_u32_sr31(table[offset_1d + 1U] - yL_0d0,
      frac[0]);
    yL_0d0 = (int16_T)(yL_0d0 + intermRes);
  }

  if (bpIndex[1] == maxIndex[1]) {
    y = yL_0d0;
  } else {
    offset_1d = offset_1d + stride;
    if (bpIndex[0] == maxIndex[0]) {
      intermRes = table[offset_1d];
    } else {
      intermRes = table[offset_1d];
      intermRes_0 = (int16_T)mul_s32_s32_u32_sr31(table[offset_1d + 1U] -
        intermRes, frac[0]);
      intermRes = (int16_T)(intermRes + intermRes_0);
    }

    intermRes = (int16_T)mul_s32_s32_u32_sr31(intermRes - yL_0d0, frac[1]);
    y = (int16_T)(yL_0d0 + intermRes);
  }
```

```
/* Constant parameters (auto storage) */
const ConstParam rtConstP = {
  :
  /* Computed Parameter: maxIndex
   * Referenced by blocks:
   * '<S2>/Pumping Constant'
   */
  { 17U, 18U },
  :
  /* Computed Parameter: BreakpointsForDimension1
   * Referenced by blocks:
   * '<S2>/Pumping Constant'
   */
  :
  { 400, 600, 800, 1000, 1200, 1400, 1600, 2000, 2400, 2800, 3200, 3600, 4000,
    4800, 5600, 6400, 7200, 8000 },
  :
  /* Computed Parameter: BreakpointsForDimension2
   * Referenced by blocks:
   * '<S2>/Pumping Constant'
   */
  :
  { 6U, 13U, 19U, 26U, 32U, 38U, 45U, 51U, 58U, 64U, 70U, 77U, 83U, 90U, 96U,
    102U, 109U, 115U, 122U }
};
```

Figure 5: Generated code for Pumping Constant lookup (contains unevenly spaced breakpoints)

Optimize Code with Evenly Spaced Power of Two Breakpoints

You can optimize the generated code performance by using evenly spaced power of two breakpoints. In this example, remap the lookup table data in the air-fuel ratio control system based on the existing measured data.

When you loaded the model, the model `PostLoadFcn` created the lookup table data in the model workspace. Retrieve the original table data via `sldemo_fuelsys_data`, modify it for evenly spaced power of two, and reassign it in the model workspace.

```
td = sldemo_fuelsys_data('sldemo_fuelsys', 'get_table_data');
```

Compute new table data for evenly spaced power of two breakpoints.

```
ntd.SpeedVect   = 64 : 2^5 : 640;             % 32 rad/sec
ntd.PressVect   = 2*2^-5 : 2^-5 : 1-(2*2^-5); % 0.03 bar
ntd.ThrotVect   = 0:2^1:88;                   % 2 deg
ntd.RampRateKiX = 128:2^7:640;                % 128 rad/sec
ntd.RampRateKiY = 0:2^-2:1;                   % 0.25 bar
```

Remap table data.

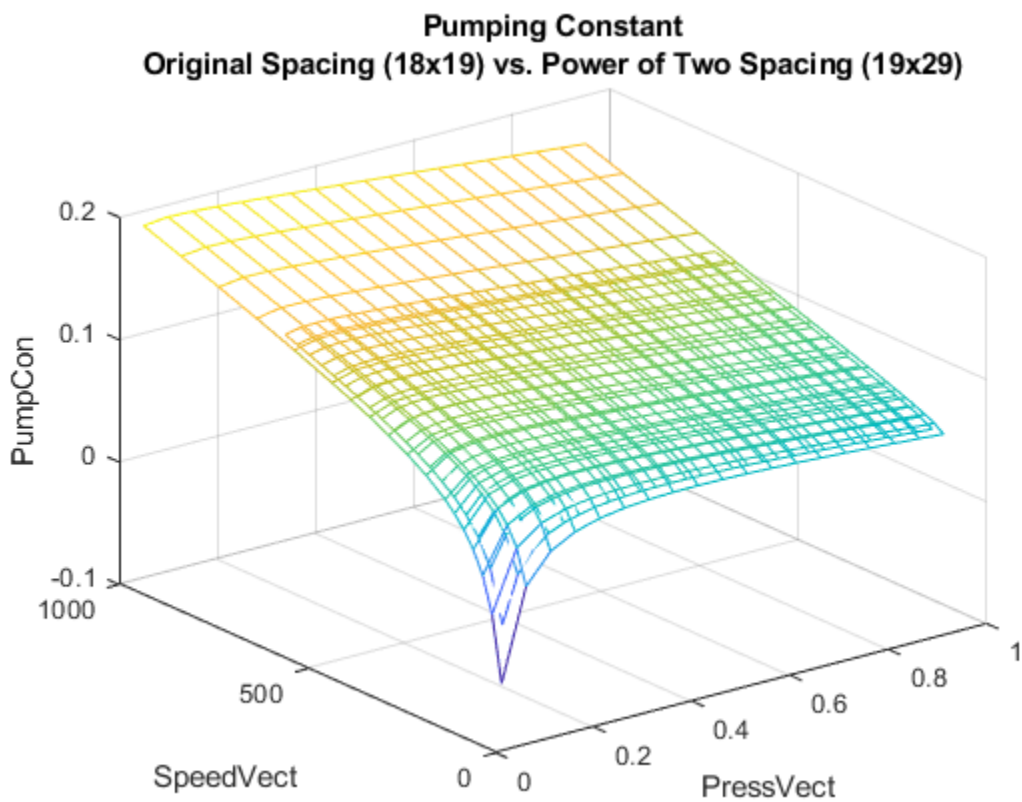
```
ntd.PumpCon  = interp2(td.PressVect,td.SpeedVect,td.PumpCon, ntd.PressVect',ntd.SpeedVect);
ntd.PressEst = interp2(td.ThrotVect,td.SpeedVect,td.PressEst,ntd.ThrotVect',ntd.SpeedVect);
ntd.SpeedEst = interp2(td.PressVect,td.ThrotVect,td.SpeedEst,ntd.PressVect',ntd.ThrotVect);
ntd.ThrotEst = interp2(td.PressVect,td.SpeedVect,td.ThrotEst,ntd.PressVect',ntd.SpeedVect);
```

Recompute Ramp Rate table data.

```
ntd.RampRateKiZ = (1:length(ntd.RampRateKiX))' * (1:length(ntd.RampRateKiY)) * td.Ki;
```

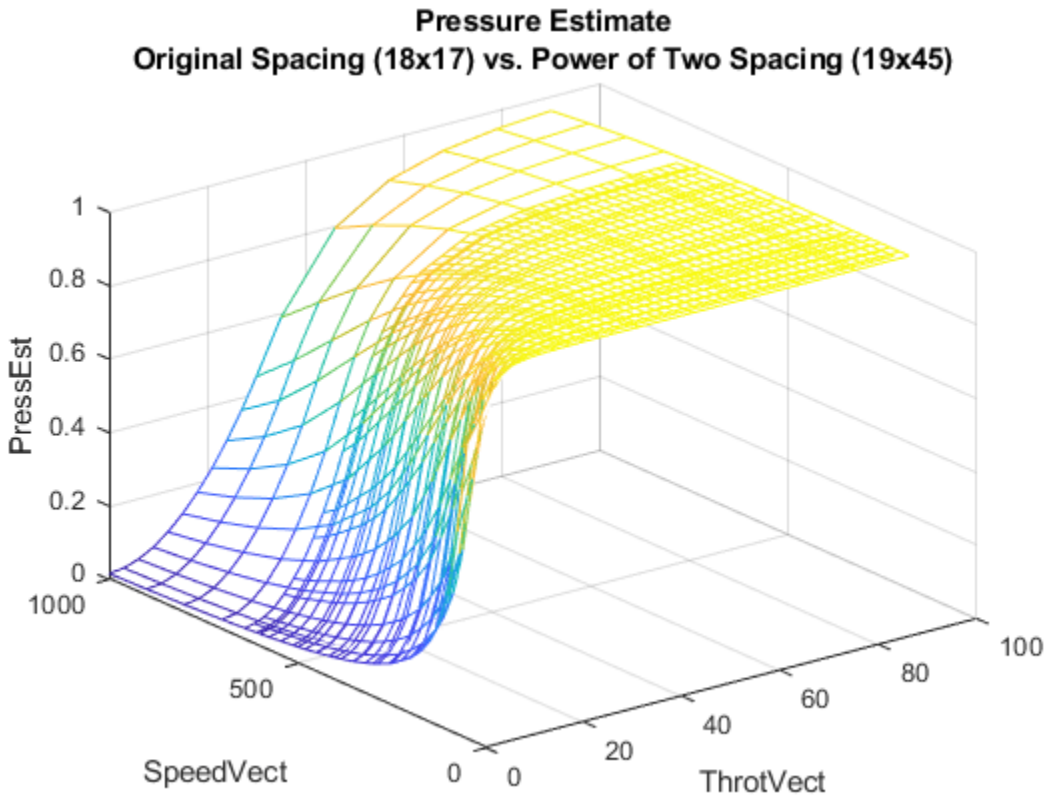
Pumping Constant Power of Two Spacing

```
figure('Tag','CloseMe');
mesh(td.PressVect,td.SpeedVect,td.PumpCon), hold on
mesh(ntd.PressVect,ntd.SpeedVect,ntd.PumpCon)
xlabel('PressVect'), ylabel('SpeedVect'), zlabel('PumpCon')
title(sprintf('Pumping Constant\nOriginal Spacing (%dx%d) vs. Power of Two Spacing (%dx%d)',...
    size(td.PumpCon,1),size(td.PumpCon,2),size(ntd.PumpCon,1),size(ntd.PumpCon,2)));
```



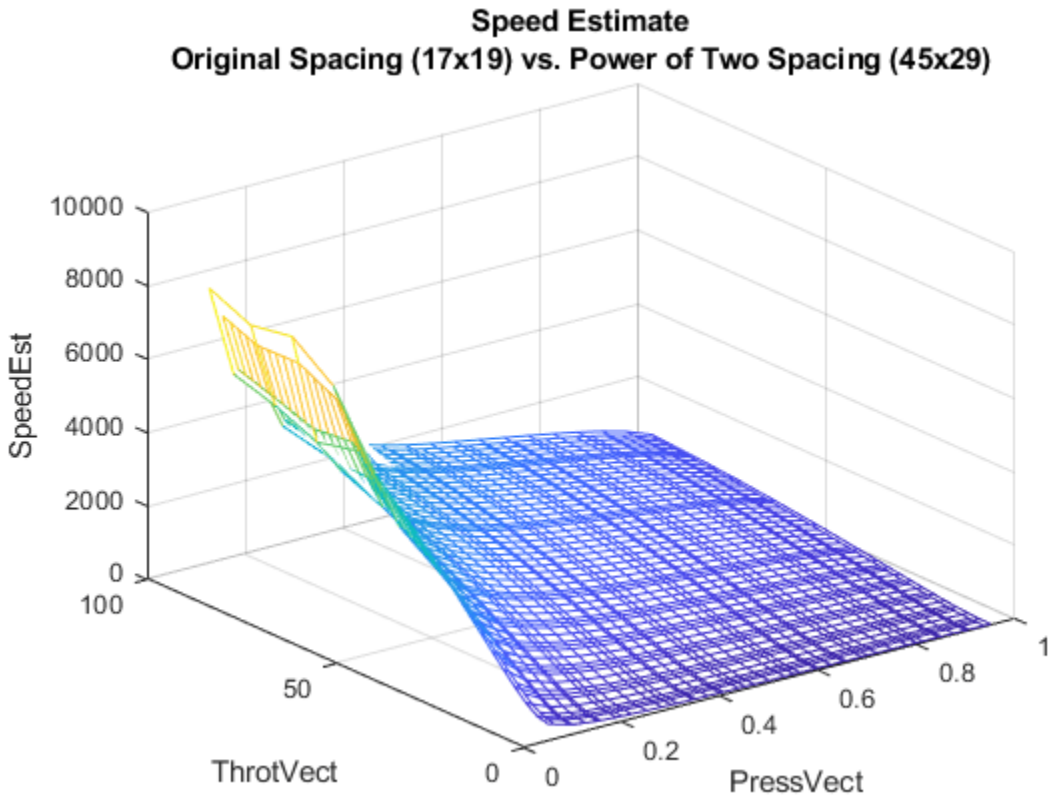
Pressure Estimate Power of Two Spacing

```
clf
mesh(td.ThrotVect,td.SpeedVect,td.PressEst), hold on
mesh(ntd.ThrotVect,ntd.SpeedVect,ntd.PressEst)
xlabel('ThrotVect'), ylabel('SpeedVect'), zlabel('PressEst')
title(sprintf('Pressure Estimate\nOriginal Spacing (%dx%d) vs. Power of Two Spacing (%dx%d)',...
    size(td.PressEst,1),size(td.PressEst,2),size(ntd.PressEst,1),size(ntd.PressEst,2)));
```



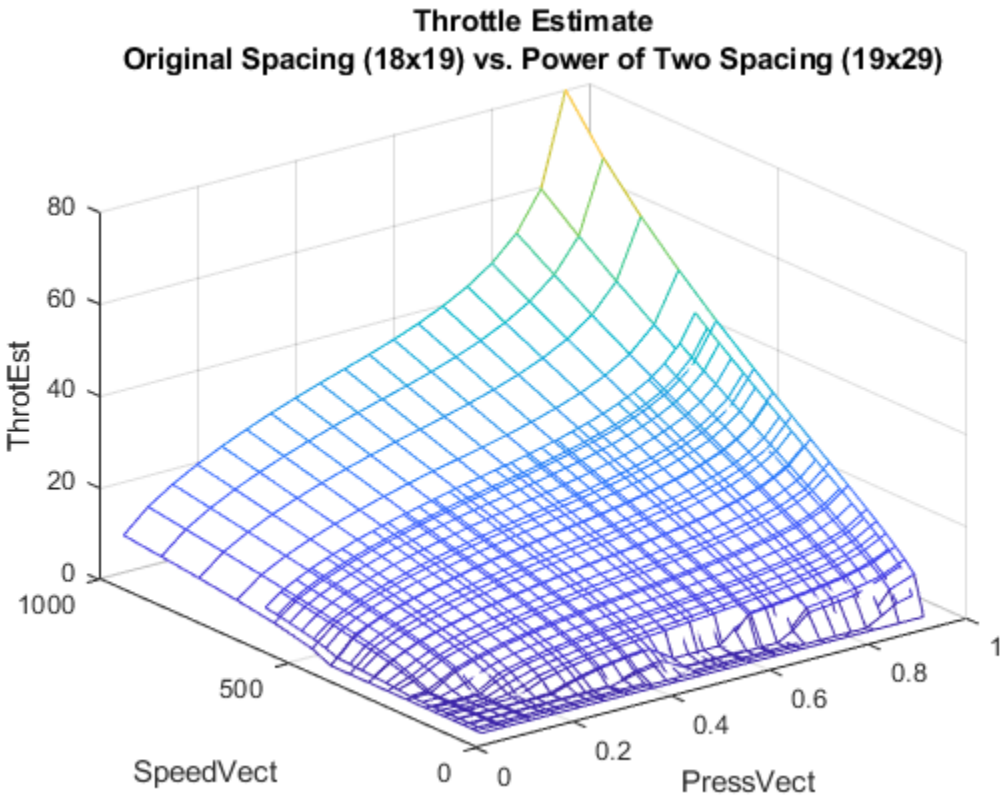
Speed Estimate Power of Two Spacing

```
clf
mesh(td.PressVect,td.ThrotVect,td.SpeedEst), hold on,
mesh(ntd.PressVect,ntd.ThrotVect,ntd.SpeedEst)
xlabel('PressVect'), ylabel('ThrotVect'), zlabel('SpeedEst')
title(sprintf('Speed Estimate\nOriginal Spacing (%dx%d) vs. Power of Two Spacing (%dx%d)',...
    size(td.SpeedEst,1),size(td.SpeedEst,2),size(ntd.SpeedEst,1),size(ntd.SpeedEst,2)));
```



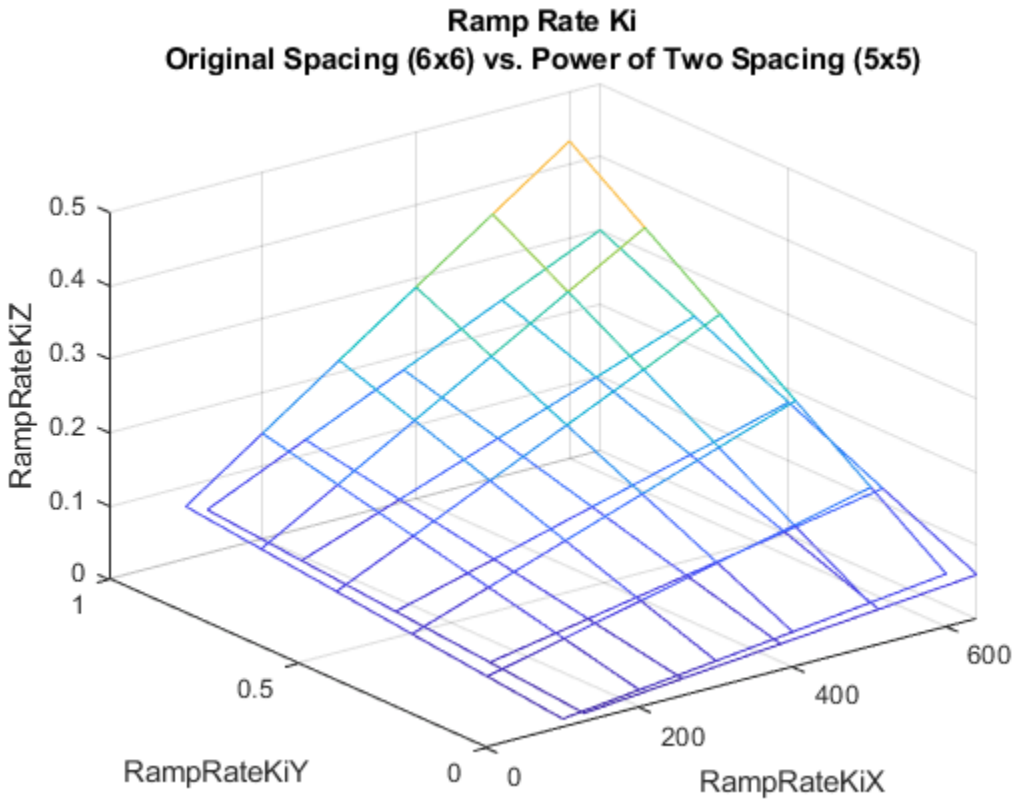
Throttle Estimate Power of Two Spacing

```
clf
mesh(td.PressVect,td.SpeedVect,td.ThrotEst), hold on
mesh(ntd.PressVect,ntd.SpeedVect,ntd.ThrotEst)
xlabel('PressVect'), ylabel('SpeedVect'), zlabel('ThrotEst')
title(sprintf('Throttle Estimate\nOriginal Spacing (%dx%d) vs. Power of Two Spacing (%dx%d)',...
    size(td.ThrotEst,1),size(td.ThrotEst,2),size(ntd.ThrotEst,1),size(ntd.ThrotEst,2)));
```



Ramp Rate Ki Power of Two Spacing

```
clf
mesh(td.RampRateKiX,td.RampRateKiY,td.RampRateKiZ'), hold on
mesh(ntd.RampRateKiX,ntd.RampRateKiY,ntd.RampRateKiZ'), hidden off
xlabel('RampRateKiX'), ylabel('RampRateKiY'), zlabel('RampRateKiZ')
title(sprintf('Ramp Rate Ki\nOriginal Spacing (%dx%d) vs. Power of Two Spacing (%dx%d)',...
    size(td.RampRateKiZ,1),size(td.RampRateKiZ,2),size(ntd.RampRateKiZ,1),size(ntd.RampRateKiZ,2
```



The default configuration causes the model to log simulation data for the top-level signals. These simulation results are stored in the workspace variable `sldemo_fuelsys_output`. Before updating the model workspace with the new data, save the result of the simulation in `hDemo.orig_data` for later comparison with the evenly spaced power of two table simulation.

```
set_param('sldemo_fuelsys','StopTime','8')
sim('sldemo_fuelsys')
hDemo.orig_data = sldemo_fuelsys_output;
```

Reassign the new table data in the model workspace.

```
hDemo.hWS = get_param('sldemo_fuelsys', 'ModelWorkspace');
hDemo.hWS.assignin('PressEst',   ntd.PressEst);
hDemo.hWS.assignin('PressVect',  ntd.PressVect);
hDemo.hWS.assignin('PumpCon',    ntd.PumpCon);
hDemo.hWS.assignin('SpeedEst',   ntd.SpeedEst);
hDemo.hWS.assignin('SpeedVect',  ntd.SpeedVect);
hDemo.hWS.assignin('ThrotEst',   ntd.ThrotEst);
hDemo.hWS.assignin('ThrotVect',  ntd.ThrotVect);
hDemo.hWS.assignin('RampRateKiX',ntd.RampRateKiX);
hDemo.hWS.assignin('RampRateKiY',ntd.RampRateKiY);
hDemo.hWS.assignin('RampRateKiZ',ntd.RampRateKiZ);
```

Reconfigure lookup tables for evenly spaced data.

```
hDemo.lookupTables = find_system(get_param('sldemo_fuelsys','Handle'),...
    'BlockType','Lookup_n-D');
```

```
for hDemo_blkIdx = 1 : length(hDemo.lookupTables)
    hDemo.blkH = hDemo.lookupTables(hDemo_blkIdx);
    set_param(hDemo.blkH,'IndexSearchMethod','Evenly spaced points')
    set_param(hDemo.blkH,'InterpMethod','None - Flat')
    set_param(hDemo.blkH,'ProcessOutOfRangeInput','None')
end
```

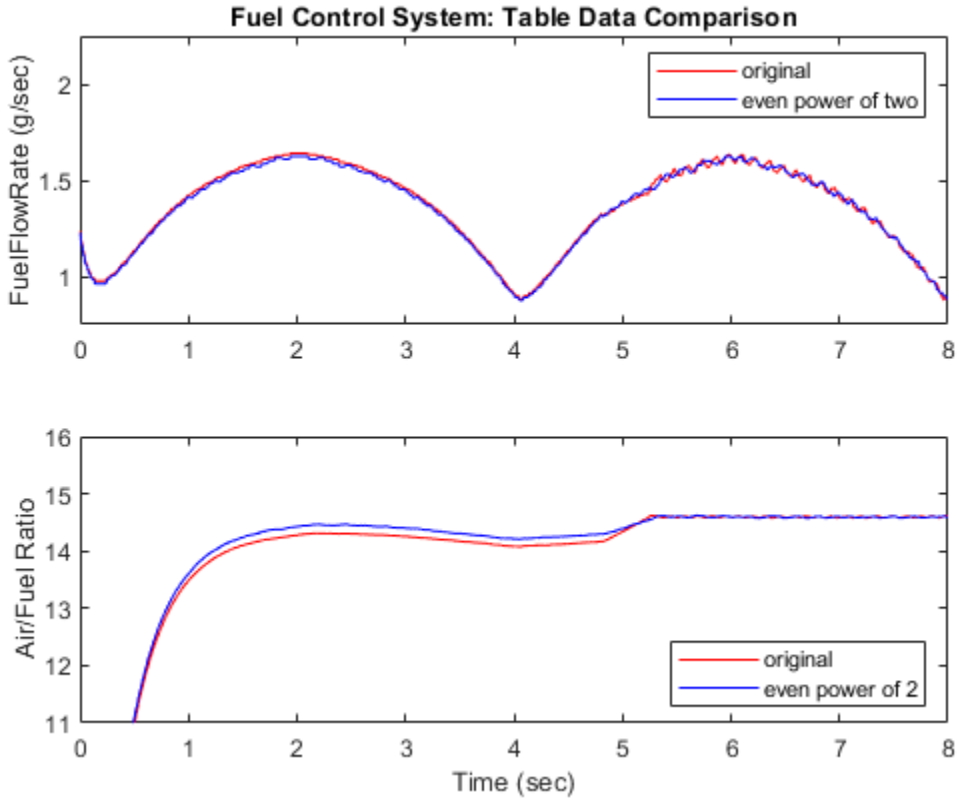
Rerun the simulation for the evenly spaced power of two implementation, and store the result of the simulation in `hDemo.pow2_data`.

```
sim('sldemo_fuelsys')
hDemo.pow2_data = sldemo_fuelsys_output;
```

Compare the result of the simulation for the fuel flow rate and the air fuel ratio. The simulation exercised the Pumping Constant and Ramp Rate Ki lookup tables, and shows an excellent match for the evenly spaced power of two breakpoints relative to the original table data.

```
figure('Tag','CloseMe');
subplot(2,1,1);
plot(hDemo.orig_data.get('fuel').Values.Time, ...
     hDemo.orig_data.get('fuel').Values.Data,'r-');
hold
plot(hDemo.pow2_data.get('fuel').Values.Time, ...
     hDemo.pow2_data.get('fuel').Values.Data,'b-');
ylabel('FuelFlowRate (g/sec)');
title('Fuel Control System: Table Data Comparison');
legend('original','even power of two');
axis([0 8 .75 2.25]);
subplot(2,1,2);
plot(hDemo.orig_data.get('air_fuel_ratio').Values.Time, ...
     hDemo.orig_data.get('air_fuel_ratio').Values.Data,'r-');
hold
plot(hDemo.pow2_data.get('air_fuel_ratio').Values.Time, ...
     hDemo.pow2_data.get('air_fuel_ratio').Values.Data,'b-');
ylabel('Air/Fuel Ratio');
xlabel('Time (sec)');
legend('original','even power of 2','Location','SouthEast');
axis([0 8 11 16]);
```

```
Current plot held
Current plot held
```



Rebuild the air-fuel ratio control system and compare the difference in the generated lookup table code.

```
slbuild('sldemo_fuelsys/fuel_rate_control');
```

```
### Starting build procedure for: fuel_rate_control
### Successful completion of build procedure for: fuel_rate_control

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================
fuel_rate_control  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 23.575s
```

Figure 6 shows the same snippets of the generated code for the 'Pumping Constant' lookup table. The generated code for evenly spaced power of two breakpoints is significantly more efficient than the unevenly spaced breakpoint case. The code consists of two simple breakpoint calculations and a direct index into the 2D lookup table data. The expensive division is avoided and the breakpoint data is not required in memory.

```
rtwtrace('sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant');
```

```
/* Lookup_n-D: '<S2>/Pumping Constant' */
bpIdx = plook_u32s16_even8cka(rtB.es_o.speed, 512, 18U);
bpIndices_idx = bpIdx;
bpIdx = plook_u32u8_even2cka(rtb_map, 8U, 28U);

/* DiscreteFilter: '<S2>/Throttle Transient' */
rtDWork.ThrottleTransient_tmp = (int16_T)(((rtb_throttle << 14) - -13107 *
  rtDWork.ThrottleTransient_DSTATE) >> 14);

/* Sum: '<S2>/Sum' incorporates:
 *  Product: '<S2>/Product'
 *  Product: '<S2>/Product2'
 */
rtb_Product2 = (int16_T)(((rtConstP.PumpingConstant_tab[bpIdx * 19U + bpIndices_idx] *
  rtb_speed >> 11) * rtb_map >> 7) + ((20972 *
  rtDWork.ThrottleTransient_tmp + -20972 * rtDWork.ThrottleTransient_DSTATE) >>
  17));
```

Simple index calculation

Direct index into table data

```
uint32_T plook_u32s16_even8cka(int16_T u, int16_T bp0, uint32_T maxIndex)
{
  uint32_T bpIndex;
  uint16_T fbpIndex;

  /* Prelookup - Index only
     Index Search method: 'even'
     Process out of range input: 'Clip to range'
     Use previous index: 'off'
     Use last breakpoint for index at or above upper limit: 'on'
   */
  if (u < bp0) {
    bpIndex = 0U;
  } else {
    fbpIndex = (uint16_T)((uint32_T)u - (uint32_T)bp0) >> 8U;
    if ((uint32_T)fbpIndex >= maxIndex) {
      bpIndex = maxIndex;
    } else {
      bpIndex = (uint32_T)fbpIndex;
    }
  }

  return bpIndex;
}
```

Figure 6: Generated code for Pumping Constant lookup (evenly spaced power of two breakpoints)

Close the example.

```
close(findobj(0,'Tag','CloseMe'));
clear hDemo* td ntd
close_system('sldemo_fuelsys',0);
rtwdemoclean;
```

Model Advisor for Code Efficiency

Improving code efficiency by using evenly spaced power of two breakpoints is one of several important optimizations for fixed-point code generation. The Simulink Model Advisor is a great tool for identifying other methods of improving code efficiency for a Simulink and Stateflow model. Make sure to run the checks under the Embedded Coder folder or Simulink Coder™.

See Also

Related Examples

- “Optimize Generated Code Using Fixed-Point Data with Simulink, Stateflow, and MATLAB” on page 36-196
- “Fixed-Point Code Generation Support” (Fixed-Point Designer)

Data Definition and Declaration Management in Embedded Coder

- “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2
- “Establish Data Ownership in a Model Hierarchy” on page 37-10

Control Placement of Global Data Definitions and Declarations in Generated Files

The generated code can create standard, global structure variables whose fields represent the signal, state, and parameter data in a model. With storage classes, you can configure data to appear in the code as separate global variables or custom global structures. For information about the standard data structures, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49 and “How Generated Code Exchanges Data with an Environment” on page 36-35. For information about storage classes, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

You can control the file placement of the variable definitions and declarations. Use the information in the table.

Goal	Details and Techniques
Generate a variable that external code can use	Apply a storage class with exported data scope, such as `ExportToFile`. The generated code declares the variable in a file that your code can include (`#include`). To generate an `extern` declaration in *`model`*`.h`, consider using the storage class `ExportedGlobal`. Alternatively, to place the declaration in a file whose name you can specify, choose one of these techniques: • To aggregate multiple variable declarations into one or more specific header files by default, apply a storage class such as `ExportToFile` to a category of data by using the Code Mappings editor. As you add blocks and signals to the model, new data items are declared in the files that you specify. • To explicitly specify the placement for an individual data item, directly apply a storage class such as `ExportToFile` and specify a header file name by using the **Header file** property. See “Exchange Data Between External Calling Code and Generated Code” on page 58-69.

Goal	Details and Techniques
Generate code that uses a variable defined by external code	Apply a storage class with imported data scope, such as `ImportFromFile`, to a data element in a model that represents the variable. Then, the generated code does not define the variable, but instead includes (`#include`) an external header file whose name you can specify. The generated algorithm (for example, the model `step` function) reads and writes to the variable. Alternatively, if you do not have an external declaration header file or do not want the generated code to include a header file, apply the storage class `ImportedExtern`. Then, the generated code declares the variable in *`model`*`_private.h`. See “Exchange Data Between External Calling Code and Generated Code” on page 58-69.
• Reduce the size of a generated file by creating multiple smaller files • Organize data into different files to make generated code easier to understand	Depending on the setting of **Configuration Parameters** > **File packaging format**, generated files such as *`model`*`.c` and *`model`*`_data.c` can contain many definitions of global variables. Corresponding header files can contain many declarations. • To make *`model`*`.c` smaller, consider setting **File packaging format** to `Modular` or `Compact (with separate data file)`. Then, the generated code defines the standard structures that represent tunable and constant parameter data in *`model`*`_data.c` instead of *`model`*`.c`. • To place different categories of data (for example, external parameters, block states, and internal signals) in specific files by default, use the Code Mappings editor. For each category of data, use a storage class such as `ExportToFile` and specify the property **Header file**. See “Configure Default Code Generation for Data” on page 33-16. • For precise control over the file placement of data, apply storage classes such as `ExportToFile` to individual data items. Use the properties of the storage class, such as **Definition file** and **Header file**, to specify file placement for each data item. For an example, see “Definition and Declaration of Signal Data” on page 26-7.

Goal	Details and Techniques
Combine multiple files into a single file	• Consider setting the model configuration parameter **File packaging format** to `Compact`. Then, the code generator does not create the *`model`*`_data.c` file. Instead, the definitions of the standard structures that store tunable and constant parameter data appear in *`model`*`.c`. • When you apply a storage class such as `ExportToFile` to data elements, leave the properties **Definition file** and **Header file** blank, the default value. Also, set the model configuration parameter **Data definition** to `Data defined in source file`. Then, the data definitions appear in *`model`*`.c` with the definitions of other global data.
Separate data definitions from function definitions	By default, the generated file that defines the entry-point functions for a model or subsystem also defines the data for that model or subsystem. When you frequently make changes to data, especially initial values for tunable parameters, the changing source code files can impede verification and make change management more difficult. To separate the data from the functions: • Consider setting the model configuration parameter **File packaging format** to a value that results in the generation of the separate data file *`model`*`_data.c`. Then, the definitions of the standard structures that store tunable and constant parameter data appear in this file instead of in *`model`*`.c`. • Apply storage classes such as `ExportToFile` to data elements by using the Code Mappings editor. Use the properties of the storage class, such as **Definition file** and **Header file**, to specify file placement. For more information, see “Configure Default Code Generation for Data” on page 33-16 and “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.
Isolate data definitions in separate files for component-based, team-oriented model development	See “Organize Data to Support Component-Based, Team-Oriented Model Development” on page 37-5.

Goal	Details and Techniques
Aggregate definitions of variant control parameters (`#define` macros) into a single header file	As described in “Generate Preprocessor Conditionals for Variant Systems” on page 27-32, a variant control parameter is a parameter object such as `Simulink.Parameter`. You apply a storage class that makes the object appear in the generated code as a macro. To control the file placement, choose one of these techniques: • For each variant control parameter that uses the storage class `Define`, set the **Header file** property to the same value. To configure these objects in a list that you can search, sort, and filter, you can use the Model Data Editor **Parameters** tab. For an example, see “Generate Variant Control Macros in Same Header File” on page 27-34. • Create one or more storage classes that represent variant control parameters. In the Custom Storage Class Designer, you can set **Header file** to a value that applies to all of the variant control parameters that use the storage class. With this technique, you do not need to manually specify a header file for each variant control parameter. For more information, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.

Organize Data to Support Component-Based, Team-Oriented Model Development

Depending on your model configuration, data definitions can appear aggregated in large generated files. In a system of components (subsystems or referenced models), you can separate and organize the data definitions into manageable, meaningful files based on the component hierarchy.

Goal	Technique
Establish ownership of global data by placing the data definitions with the code generated for specific components	When you divide a system into components by using referenced models and atomic subsystems, by default, global data is typically defined by the code generated for the top component in the hierarchy. Global data includes parameters, signals, and states to which you apply storage classes (see “C Code Generation Configuration for Model Interface Elements” on page 33-2). Instead, you can place global data definitions with other components in the hierarchy. For an example involving referenced models, see “Establish Data Ownership in a Model Hierarchy” on page 37-10. • To place a data definition with the code generated for a referenced model, use a built-in storage class such as `ExportToFile`, or a storage class that you create in a package, and set the **Owner** property to the name of the referenced model. To use **Owner** in a model, you must select the model configuration parameter **Use owner from data object for data definition placement**. The default setting for this parameter, cleared, means the code generator ignores the setting that you specify for **Owner**. If only one referenced model uses a parameter object, consider storing the object in the model workspace of that model. Then, the code generated for that model defines the data. You do not need to specify an owner (**Owner**) for the object. • In an atomic subsystem, use a storage class such as `ExportToFile` and explicitly specify the name of the definition file by using the **Definition file** property. However, you cannot place the definition file with the source files that belong to the subsystem. Instead, the definition file that you specify appears in the folder generated for the model.

Goal	Technique
For an atomic subsystem, place the standard data structures for the subsystem with the subsystem code	By default, the standard data structures for an atomic subsystem appear as substructures of the data structures for the entire model. To generate separate data structures for a subsystem, select the **Function with separate data** parameter in the subsystem block. Then, the subsystem data appear in separate structure variables that the subsystem code defines. See “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2.

Specify Default Placement

When you create data in a model by adding blocks and signal lines, by default, the data definitions appear in `model.c`. To specify a different default placement, use these tools and parameters:

- The model configuration parameter **File packaging format**. The setting that you choose determines whether the code generator places the standard structures that store tunable and constant parameter data in `model_data.c` instead of `model.c`. See File packaging format.
- The Code Mappings editor. For each category of model data, you can specify a default storage class, which controls file placement. For example, apply the storage class `ExportToFile` to a data category and, in the Property Inspector, use the **DefinitionFile**, **HeaderFile**, and **Owner** properties to control file placement.
- The model configuration parameters **Data definition** and **Data declaration**. These configuration parameters specify the default file placement for data items that you apply storage classes to.

  - These configuration parameters do not affect data items that you apply these storage classes to:

    - `ExportedGlobal`
    - `ImportedExtern`
    - `ImportedExternPointer`
    - `BitField`
    - `FileScope`
    - `Localizable`
    - `Struct`
    - `CompilerFlag`

    In the Embedded Coder Dictionary, the **Header File** value for unsupported storage classes is empty. You cannot specify default file placement for unsupported storage classes.
  - If you apply a storage class that explicitly specifies definition and declaration files for a data element, those specifications override the **Data definition** and **Data declaration** configuration parameters.

  For more information, see Data definition and Data declaration.
- The subsystem block parameter **Function with separate data**. When you select this parameter, the standard structures that store the subsystem data, such as the DWork structure, appear as

separate structure variables. By default, the variable definitions appear in the source file that defines the subsystem execution function. See “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2.

Override Default Placement for Individual Data Items

For an individual data item, to override the default file placement, use the Code Mappings editor to apply a storage class directly. For example, to make a data item appear in the generated code as a global variable, apply the storage class `ExportToFile`. Then, use the **Definition file** and **Header file** properties to control file placement. To apply storage classes directly, see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.

Prevent Name Clashes by Configuring Data Item as static

To apply the C keyword `static` to a global variable, which can help you avoid name clashes by limiting the scope of the variable name to the file that defines the variable, choose one of these techniques:

- Apply the built-in storage class `FileScope`. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.
- If `FileScope` does not satisfy your requirements, create your own storage class by using the Custom Storage Class Designer. In the Designer, set **Data scope** to `File` or `Auto`.
  - With `File`, the data element appears in the code as a `static` global variable.
  - With `Auto`, the code generator first attempts to represent the data element with a local variable in a function. If this attempt fails, the code generator uses a `static` global variable.

  For an example that shows how to create your own storage class by using the Designer, see “Create and Apply a Storage Class” on page 40-14.

You cannot apply `static` to the standard data structures such as the DWork structure.

To access `static` data, you can configure the generated code to include an interface such as an `a2l` (ASAP2) file. For more information, see “Export ASAP2 File for Data Measurement and Calibration” on page 63-2. To place the data in a specific memory location by including pragmas or other decorations in the generated code, create your own memory section and storage class. See “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Code Generation Impact of Storage Location for Parameter Objects

You can create a parameter object (such as `Simulink.Parameter`) in the base workspace, a model workspace, or a data dictionary. However, when you apply a storage class to the object, the location of the object can impact the file placement of the corresponding data definition in the generated code. See “Code Generation Impact of Storage Location for Parameter Objects” on page 36-119.

Specify Default #include Syntax for Data Header Files

To control the file placement of a data item, such as a signal line or block state, in the generated code, you can apply a storage class to the data item (see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67). You then use the **Header file** property to specify the generated or external header file that contains the declaration of the data.

To reduce maintenance effort and data entry, when you specify **Header file**, you can omit delimiters (" or <>) and use only the file name. You can then control the default delimiters that the generated code uses for the corresponding `#include` directives. To use angle brackets by default, set **Configuration Parameters** > **Code Generation** > **Code Placement** > **#include file delimiters** to `#include <header.h>`.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Simulink Modeling Components”
- “Manage File Packaging of Generated Code Modules” on page 53-12
- “How Generated Code Exchanges Data with an Environment” on page 36-35
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Establish Data Ownership in a Model Hierarchy

This example shows how to establish ownership of global data in the code generated from referenced models.

Create a global variable in the generated code by applying a storage class to a data element in a referenced model. For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2).

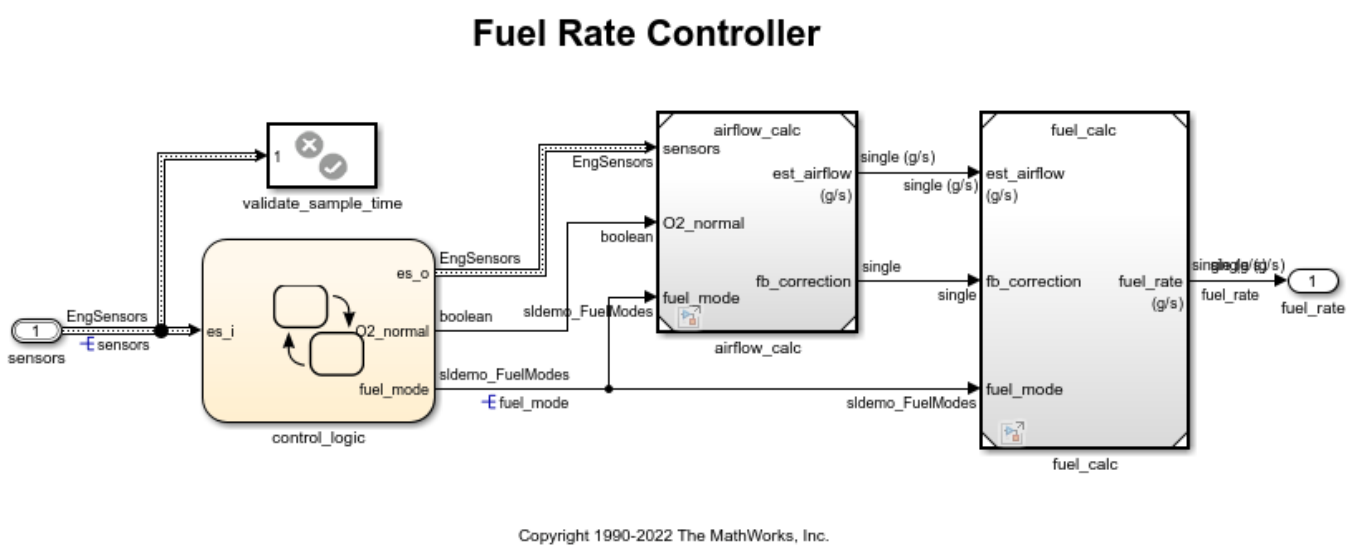
Under certain conditions, the code generator places the variable definition with the code generated from the top model in the hierarchy. This default placement can make it difficult to determine which model is responsible for the data and to manage code changes in a team-based development environment.

To place the definition with the code generated from the relevant model, apply a storage class such as `ExportToFile` and specify the **Owner** custom attribute. Alternatively, if you use a `Simulink.Parameter` in only one model, you can establish ownership by storing the object in a model workspace instead of the base workspace or a data dictionary.

Explore Example Model

2. Run the script `prepare_sldemo_fuelsys_dd_ctrl.m`. The script opens the model `sldemo_fuelsys_dd_controller` and prepares it for this example.

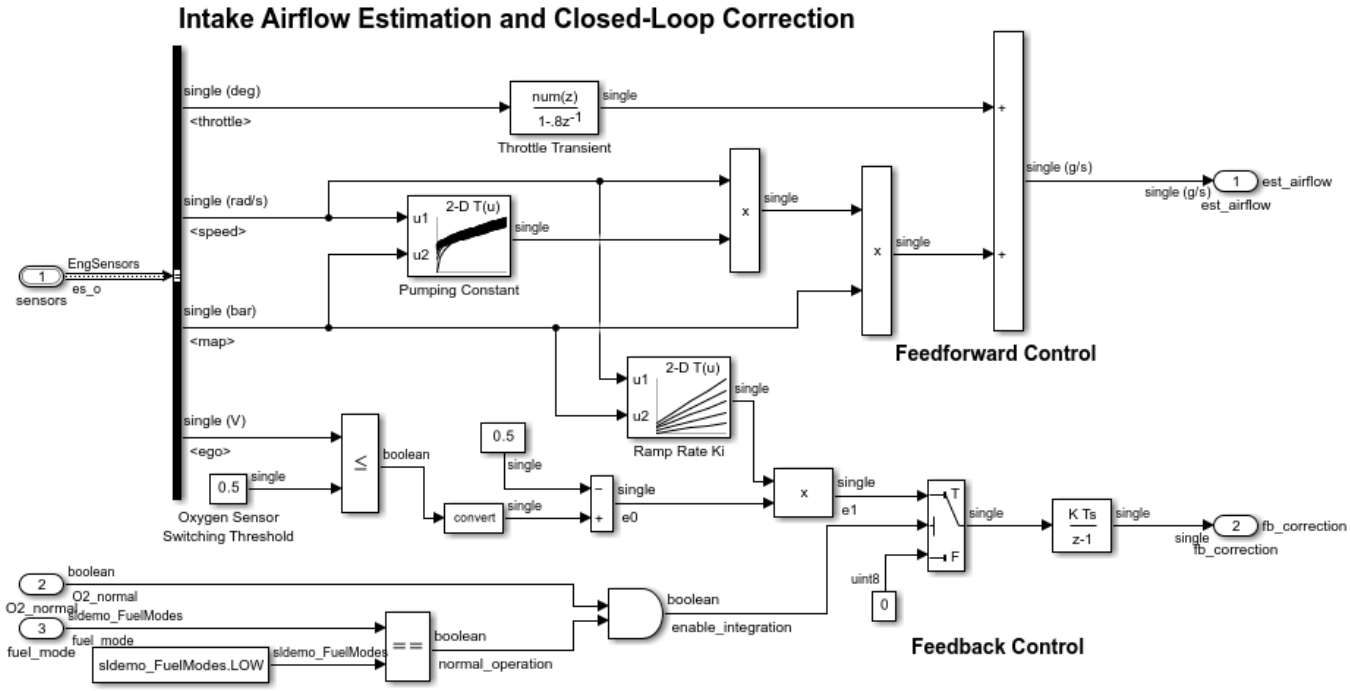
```
prepare_sldemo_fuelsys_dd_ctrl
open_system('sldemo_fuelsys_dd_controller')
```



This controller model contains two models, `airflow_calc` and `fuel_calc`. The two output signals of `airflow_calc`, `est_airflow` and `fb_correction`, are inputs of `fuel_calc`.

3. Open the `airflow_calc` model.

```
open_system('airflow_calc')
```



4. On the **C Code** tab, under **Code Interface**, click **Default Code Mappings**.

5. In the Code Mappings Editor, open the **Outports** tab.

6. In the `airflow_calc` model, select the `est_airflow` Outport block.

7. In the Code Mappings Editor, inspect the value in the **Storage Class** column. The signal that this block represents, `est_airflow`, uses the storage class `ExportToFile`. With this setting, the signal appears in the generated code as a global variable. The Outport block labeled `fb_correction` also uses this setting.

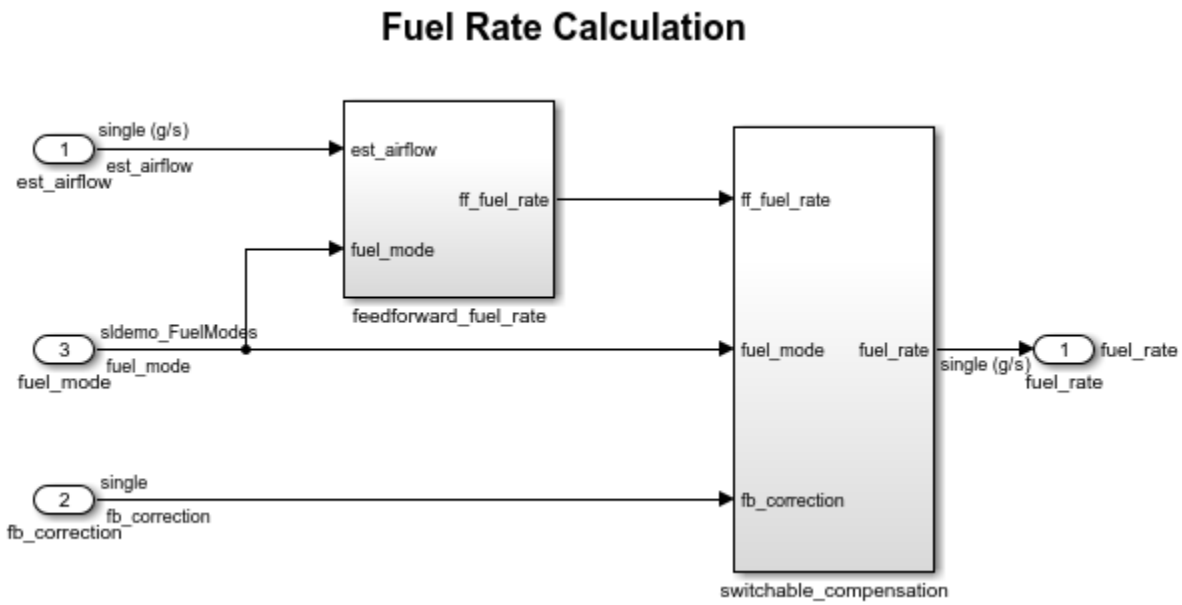
8. In the Code Mappings Editor, select the **Parameters** tab and click the **Refresh** button. The Code Mappings Editor now shows information about workspace variables and objects, such as `Simulink.Parameter` objects, that the model uses to set block parameter values.

9. In the **Filter contents** box, enter `numerator`. The `Simulink.Parameter` object `numerator_param`, which is in the base workspace, sets the value of the **Numerator** parameter in the Discrete Filter block labeled `Throttle Transient`. The object uses the storage class `ExportToFile`.

10. Select the **Signals/States** tab. In the **Filter contents** box, enter `e0`. The signal `e0`, which is internal to the `airflow_calc` referenced model, also uses `ExportToFile`. The signal is internal because it is not a root-level input or output of the model.

11. Open the `fuel_calc` model.

```
open_system('fuel_calc')
```



12. In the Code Mappings Editor for this model, open the **Inports** tab. The Inport blocks `est_airflow` and `fb_correction` use the same storage class, `ExportToFile`, as the corresponding Outport blocks in `airflow_calc`. In the **Outports** tab, the Outport block labeled `fuel_rate` also uses `ExportToFile`.

Generate and Inspect Code

1. Generate code from the controller model, `sldemo_fuelsys_dd_controller`.

```
evalc('slbuild(''sldemo_fuelsys_dd_controller'')');
```

2. In the dropdown menu at the top of the Code Generation Report window, change `sldemo_fuelsys_dd_controller` to `airflow_calc`. From the left pane, click `airflow_calc.c`.

The file `airflow_calc.c` defines the global variable that represents the signal `e0`. Because the blocks that write to and read from `e0` exist only in `airflow_calc`, the code generator assumes that this model owns the signal.

```
file = fullfile('slprj','ert','airflow_calc','airflow_calc.c');
rtwdemodbtype(file,'/* Definition for custom storage class: ExportToFile */',...
    'real32_T e0;',1,1)
```

```
/* Definition for custom storage class: ExportToFile */
real32_T e0;                            /* '<Root>/Sum1' */
```

3. In the code generation report, return to the code generated for `sldemo_fuelsys_dd_controller`.

The file `sldemo_fuelsys_dd_controller.c` defines the other global variables.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller.c');
rtwdemodbtype(file,...
```

```
    '/* Definition for custom storage class: ExportToFile */',...
    'real32_T numerator_param[2] = { 0.01F, -0.01F } ;',1,1);
```

```
/* Definition for custom storage class: ExportToFile */
real32_T est_airflow;                  /* '<Root>/airflow_calc' */
real32_T fb_correction;                /* '<Root>/airflow_calc' */
real32_T fuel_rate;                    /* '<Root>/fuel_calc' */
real32_T numerator_param[2] = { 0.01F, -0.01F } ;
```

Because the signals `est_airflow` and `fb_correction` pass between the two models and through the top-level controller model, the code generator assumes that `sldemo_fuelsys_dd_controller` owns the signals. The code generator makes a similar assumption about `fuel_rate`. Because the parameter object `numerator_param` exists in the base workspace, any model in the hierarchy can use the object. Therefore, the code generator assumes that `sldemo_fuelsys_dd_controller` owns the parameter.

You can configure each shared signal and the parameter object so that the corresponding variable definition appears in the code generated for the relevant model.

Configure Data Ownership

In this example, configure code generation settings so that:

- The code generated for the `airflow_calc` model defines the signals that pass between the two models, `est_airflow` and `fb_correction`.
- The code generated for `airflow_calc` defines the parameter data, `numerator_param`.
- The code generated for `fuel_calc` defines the output signal that `fuel_calc` calculates, `fuel_rate`.

1. In the `airflow_calc` model, on the **Modeling** tab, under **Design**, click **Property Inspector**.

2. In the Code Mappings Editor, select the **Outports** tab.

3. Select the row that corresponds to the Outport block `est_airflow`. The Property Inspector shows the properties of the Outport block.

4. In the Property Inspector, under **Code**, set **Owner** to `airflow_calc`. The code generator places the definition of the global variable with the code generated for `airflow_calc`.

5. For `fb_correction`, use the Code Mappings Editor and the Property Inspector to set **Owner** to `airflow_calc`.

6. From the Code Mappings Editor, select the **Parameters** tab.

7. Find the row that corresponds to the parameter object, `numerator_param`. In the row, double-click the icon in the left column. The Model Explorer opens and displays the properties of `numerator_param`.

8. In the **Model Hierarchy** pane, expand the **airflow_calc** node to see the subordinate **Model Workspace** node.

9. Use the Model Explorer to move `numerator_param` from the base workspace to the `airflow_calc` model workspace. For example, in the **Model Hierarchy** pane, select **Base Workspace**. Then, drag `numerator_param` from the **Contents** pane to the **Model Workspace** node

in the **Model Hierarchy** pane. With this change, the code generator assumes that `airflow_calc` owns `numerator_param`, and places the variable definition in the code generated for `airflow_calc`.

10. In the `fuel_calc` model, on the **Modeling** tab, under **Design**, click **Property Inspector**.

11. For the Inport blocks `est_airflow` and `fb_correction`, use the Code Mappings Editor and the Property Inspector to set **Owner** to `airflow_calc`. The `fuel_calc` code does not define the variables with this configuration.

12. For the Outport block, use the Code Mappings Editor and the Property Inspector to set **Owner** to `fuel_calc`.

Alternatively, to configure the data in the models, enter this code at the command prompt:

```
temp = Simulink.Signal;
temp.CoderInfo.StorageClass = 'Custom';
temp.CoderInfo.CustomStorageClass = 'ExportToFile';
temp.CoderInfo.CustomAttributes.Owner = 'airflow_calc';

set_param('airflow_calc/est_airflow','SignalName','est_airflow')
set_param('airflow_calc/est_airflow','SignalObject',copy(temp))

set_param('airflow_calc/fb_correction','SignalName','fb_correction')
set_param('airflow_calc/fb_correction','SignalObject',copy(temp))

mdlwks = get_param('airflow_calc','ModelWorkspace');
assignin(mdlwks,'numerator_param',copy(numerator_param));

portHandles = get_param('fuel_calc/est_airflow','portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle,'Name','est_airflow')
set_param(outportHandle,'SignalObject',copy(temp))

portHandles = get_param('fuel_calc/fb_correction','portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle,'Name','fb_correction')
set_param(outportHandle,'SignalObject',copy(temp))

temp.CoderInfo.CustomAttributes.Owner = 'fuel_calc';

set_param('fuel_calc/fuel_rate','SignalName','fuel_rate')
set_param('fuel_calc/fuel_rate','SignalObject',copy(temp))

clear temp portHandles outportHandle numerator_param
```

13. In each of the three models, specify the owner of the data object. In the Configuration Parameters window, click **Code Generation > Code Placement**, then select Use owner from data object for data definition placement*. When this setting is cleared, the code generator ignores the values specified for **Owner**.

```
set_param('fuel_calc','EnableDataOwnership','on')
set_param('airflow_calc','EnableDataOwnership','on')
set_param('sldemo_fuelsys_dd_controller','EnableDataOwnership','on')
```

14. Save the referenced models.

```
save_system('fuel_calc')
save_system('airflow_calc')
```

Generate Improved Code

1. Generate code from the controller model, `sldemo_fuelsys_dd_controller`.

```
evalc('slbuild(''sldemo_fuelsys_dd_controller'')');
```

Now, the file `sldemo_fuelsys_dd_controller.c` does not define any of the global variables.

2. In the code generation report, inspect the code generated for `airflow_calc`.

The file `airflow_calc.c` now defines the global variables that belong to `airflow_calc`.

```
file = fullfile('slprj','ert','airflow_calc','airflow_calc.c');
rtwdemodbtype(file,'/* Definition for custom storage class: ExportToFile */',...
    'real32_T numerator_param[2] = { 0.01F, -0.01F } ;',1,1)
```

```
/* Definition for custom storage class: ExportToFile */
real32_T e0;                            /* '<Root>/Sum1' */
real32_T est_airflow;                   /* '<Root>/Sum' */
real32_T fb_correction;                 /* '<Root>/Discrete Integrator' */
real32_T numerator_param[2] = { 0.01F, -0.01F } ;
```

3. Inspect the code generated for `fuel_calc`. The file `fuel_calc.c` defines the global variable `fuel_rate`.

```
file = fullfile('slprj','ert','fuel_calc','fuel_calc.c');
rtwdemodbtype(file,'/* Definition for custom storage class: ExportToFile */',...
    'real32_T fuel_rate;',1,1)
```

```
/* Definition for custom storage class: ExportToFile */
real32_T fuel_rate;                     /* '<S2>/Merge' */
```

Create Single Point of Specification for Signals That Pass Between Models

The Outport blocks in `airflow_calc` and Inport blocks in `fuel_calc` represent the signal data elements `est_airflow` and `fb_correction`. To change the configuration of one of these signals, you must make the change in both models. For example, to change the storage class of `est_airflow` from `ExportToFile` to `Volatile`, you must change the storage class for the Outport block in `airflow_calc` and the Inport block in `fuel_calc`.

To make maintenance of each signal easier, store the code generation settings in a `Simulink.Signal` object, which can exist in the base workspace or a data dictionary.

1. Open the Property Inspector, then click the `est_airflow` outport block.

2. In the Property Inspector, next to the **Signal name** property, click the button with the three vertical dots next to the **Signal name** property. Select **est_airflow: Create and Resolve...**.

3. In the **Create New Data** dialog box, set **Value** to `Simulink.Signal` and click **Create**. A `Simulink.Signal` object named `est_airflow` appears in the base workspace. In the Model Data Editor, for `est_airflow`, the check box in the **Resolve** check box is selected, which means the Outport block acquires code generation settings from the signal object in the base workspace.

4. In the `est_airflow` property dialog box, set **Storage class** to `ExportToFile`.

5. Set **Owner** to `airflow_calc`.

6. Use the Model Data Editor to create a similar signal object for `fb_correction`.

In the Model Data Editor for `fuel_calc`, on the **Inports/Outports** tab, in the **Resolve** column, select the check boxes for `est_airflow` and `fb_correction`. Now, each Inport block acquires code generation settings from the corresponding signal object.

Alternatively, to create the signal object and configure the blocks and lines in the model, enter these commands at the command prompt:

```
est_airflow = Simulink.Signal;
est_airflow.CoderInfo.StorageClass = 'Custom';
est_airflow.CoderInfo.CustomStorageClass = 'ExportToFile';
est_airflow.CoderInfo.CustomAttributes.Owner = 'airflow_calc';

fb_correction = Simulink.Signal;
fb_correction.CoderInfo.StorageClass = 'Custom';
fb_correction.CoderInfo.CustomStorageClass = 'ExportToFile';
fb_correction.CoderInfo.CustomAttributes.Owner = 'airflow_calc';

set_param('airflow_calc/est_airflow', 'StorageClass', 'Auto')
set_param('airflow_calc/est_airflow','MustResolveToSignalObject','on')

set_param('airflow_calc/fb_correction', 'StorageClass', 'Auto')
set_param('airflow_calc/fb_correction','MustResolveToSignalObject','on')

portHandles = get_param('fuel_calc/est_airflow','portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle, 'StorageClass', 'Auto')
set_param(outportHandle,'MustResolveToSignalObject','on')

portHandles = get_param('fuel_calc/fb_correction','portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle, 'StorageClass', 'Auto')
set_param(outportHandle,'MustResolveToSignalObject','on')

clear portHandles outportHandle
```

7. Save the models and generate code from `sldemo_fuelsys_dd_controller`. The code is the same as it was before you created the `Simulink.Signal` objects. Now, you can make changes to the signal objects instead of the corresponding blocks and lines in the models.

```
save_system('airflow_calc')
save_system('fuel_calc')
evalc('slbuild(''sldemo_fuelsys_dd_controller'')');
```

See Also

Related Examples

- “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2

- “Generate Code for Model Reference Hierarchy” on page 6-2

Data Types in Embedded Coder

- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “Control File Placement of Custom Data Types” on page 38-21
- “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24
- “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30
- “Specify Boolean and Data Type Limit Identifiers” on page 38-37

Manage Replacement of Simulink Data Types in Generated Code

You can use the **Data type replacement** configuration parameter to control the replacement of Simulink data types in generated code:

- If you set the parameter to `Use C data types with fixed-width integers` (default), the generated code uses data types from the C99 language standard, which includes definitions from `stdint.h` and `stdbool.h`. See “C99 Data Types in Generated Code” on page 38-2.

  The code generator does not create the “rtwtypes.h” on page 52-40 header file. If you use custom code that requires Simulink Coder data type definitions, you can force the generation of `rtwtypes.h` by selecting the **Coder typedefs compatibility** check box.
- If you set the parameter to `Use coder typedefs`, the code generator creates the header file `rtwtypes.h` and generates data types that are based on the C89 language standard. The `rtwtypes.h` file defines the data types through `typedef` statements that build on C primitive types, for example, `float` and `short`. If you also select the **Specify custom data type names** check box, you can specify replacement names for the Simulink Coder data types. See “Control Names of Simulink Coder Primitive Data Types” on page 38-3.

By default, the generated code aggregates signal, block parameter, and state data into structures (see “How Generated Code Exchanges Data with an Environment” on page 36-35 and “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49). You can specify a naming rule for the structure types. You can also place data items into separate custom structures and specify names for the custom structure types. See “Control Names of Structure Types” on page 38-10.

Use the naming control to:

- Conform to coding standards. For an example, see “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24.
- Integrate the generated code with your external code. For an example, see “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30.
- Make the generated code more readable and meaningful.

You can also reuse data type definitions in generated code, for example, `typedef` statements from external code. See “Generate Code That Reuses Data Types From External Code” on page 38-10.

C99 Data Types in Generated Code

When the **Data type replacement** configuration parameter is set to `Use C data types with fixed-width integers`, the code generator replaces Simulink data types with C99 data types as shown in this table.

Simulink Name	Code Generation Name
`double`	`double`
`single`	`float`
`int32`	`int32_t`
`int16`	`int16_t`

Simulink Name	Code Generation Name
int8	int8_t
uint32	uint32_t
uint16	uint16_t
uint8	uint8_t
boolean	bool
int	int
uint	unsigned int
char	char
uint64	uint64_t
int64	int64_t

You cannot specify replacements for the generated data type names.

Control Names of Simulink Coder Primitive Data Types

To rename a Simulink Coder primitive data type such as `int8_T`, create a `Simulink.AliasType` object whose name matches the type name that you want the generated code to use. For example, to create a type named `myType`, at the command prompt, enter:

```
myType = Simulink.AliasType;
```

Set the `BaseType` property to the name of the Simulink data type that corresponds to the target Simulink Coder data type. For example, if the target type is `int8_T`, specify `int8`. To identify the Simulink data type, use the information in the table.

Simulink Coder Type Name	Corresponding Simulink Type Name
real_T	double
real32_T	single
int32_T	int32
int16_T	int16
int8_T	int8
uint32_T	uint32
uint16_T	uint16
uint8_T	uint8

Simulink Coder Type Name	Corresponding Simulink Type Name
`boolean_T`	`boolean` For configuring data type replacement throughout an entire model (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**), you can alternatively use one of these types: • `uint8` • `int8` • `int`*n**
`int_T`	`int`*n**
`uint_T`	`uint`*n**
`char_T`	`int`*n**
`uint64_T`	`uint64`
`int64_T`	`int64`

* *n* must be 8, 16, 32, or 64.

Note The `boolean_T` `BaseType` must promote to a signed `int`.

For example, to replace the type name `uint8_T` with `myType`, set the `BaseType` property of the `Simulink.AliasType` object to `'int8'`.

```
myType.BaseType = 'int8';
```

Then, use one or both of these techniques to apply the type to data items in a model:

- Configure data type replacements. Throughout the code that you generate from a model, you can replace a Simulink Coder data type name with the name of the `Simulink.AliasType` object. You configure data type replacements through model configuration parameters (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**.

  For an example that shows how to use data type replacement, see “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24.
- Use the name of the `Simulink.AliasType` object to specify the data type of an individual signal, that is, a block output, or block parameter. By default, due to data type propagation and inheritance (see “Data Type Inheritance Rules”), the signals, states, and parameters of other downstream blocks typically inherit the same data type. Optionally, you can configure data items in upstream blocks to inherit the type (`Inherit: Inherit via back propagation`) or stop the propagation at an arbitrary block by specifying a noninherited data type setting.

  To specify a data type for an individual data item, use the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). You can use the name of the same `Simulink.AliasType` object to specify the data types of multiple data items.

  For an example that shows how to use a `Simulink.AliasType` object directly in a model, see “Create Data Type Alias in the Generated Code” on page 38-10.

Control Names of Primitive Types	Technique
Change the name of a primitive type that the generated code uses to define variables by default (for example, `int8_T`).	Configure a data type replacement for an entire model. To replace a default Simulink Coder data type name with the corresponding Simulink type name, for example, to replace `int8_T` with `int8`, you do not need to create a `Simulink.AliasType` object. See “Use Simulink Data Type Names Instead of Simulink Coder Data Type Names” on page 38-6. To share data type replacements throughout a model reference hierarchy, use referenced configuration sets (see “Share a Configuration Across Referenced Models”).
Configure the generated code to define a particular data item, such as a variable, by using a specific, meaningful type name.	In the model, locate the data item that corresponds to the variable. For example, trace from the code generation report to the model. Use the name of the `Simulink.AliasType` object to set the data type of the item. If necessary, to prevent other data items in upstream and downstream blocks from using the same type name, configure those items to use a data type setting that is not inherited. By default, most signals use the inherited type `Inherit: Inherit via internal rule`.
Use the same type name for multiple signals and other data items in a data path, which is a series of connected blocks.	In the model, use the name of a `Simulink.AliasType` object to set the data type of one of the signals in the path, for example, a root-level Inport block or a Constant block. For example, if the path begins with an Inport block at the root level of the model, you can specify the type in that block. By default, due to data type propagation, the data items in the other blocks typically inherit the same type. Usually, no matter where on the data path you specify the type, downstream data items inherit the type. You can configure upstream data items to inherit the type, too. Consider specifying the type in a block that you do not expect to remove or change frequently.
Configure the generated code to replace an implementation-dependent type, such as `char_T` or `boolean_T`, and another type of equivalent bit length with a single type name that you specify.	Use the name of the `Simulink.AliasType` object to configure multiple data type replacements simultaneously. See “Replace Implementation-Dependent Types with the Same Type Name” on page 38-7.

Use Simulink Data Type Names Instead of Simulink Coder Data Type Names

For consistency between the data type names that you see in a model and in the generated code, you can configure data type replacements so the code uses Simulink type names instead of Simulink Coder names. For each Simulink Coder type that you want to rename, use the Simulink data type name to specify the replacement name. You do not need to create `Simulink.AliasType` objects.

Simulink Name	Code Generation Name	Replacement Name
double	real_T	<empty>
single	real32_T	<empty>
int32	int32_T	<empty>
int16	int16_T	<empty>
int8	int8_T	int8
uint32	uint32_T	<empty>
uint16	uint16_T	<empty>
uint8	uint8_T	<empty>
boolean	boolean_T	<empty>
int	int_T	<empty>
uint	uint_T	<empty>
char	char_T	<empty>
uint64	uint64_T	<empty>
int64	int64_T	<empty>

To replace `boolean_T`, `int_T`, or `uint_T`, use the information in the table.

Simulink Coder Type Name	Replacement Names to Use
`boolean_T`	One of these names: • `boolean` • `uint8` • `int8` • `int`*n*`*`
`int_T`	`int`*n*`*`
`uint_T`	`uint`*n*`*`
`char_T`	`int`*n*`*`

* Replace *n* with the number of bits displayed in **Configuration Parameters** > **Hardware Implementation** for either **Number of bits: int** or **Number of bits: char**. Use the appropriate number of bits for the data type that you want to replace.

You cannot use this technique to replace `real_T` with `double` or `real32_T` with `single`.

Replace Implementation-Dependent Types with the Same Type Name

Some Simulink Coder type names map to C primitives that are implementation-dependent. For example, the Simulink Coder type `int_T` maps to the C type `int`, whose bit length depends on the native integer size of your target hardware. Other implementation-dependent types include `boolean_T`, `char_T`, and `uint_T`.

For more readable, simpler code, you can use the same type name to replace multiple Simulink Coder types of the same size. For example, if the native integer size of your hardware is 32 bits, you can replace `int_T` and `int32_T` with the same type name, say, `myInt`.

1. Configure your target hardware settings in **Configuration Parameters** > **Hardware Implementation**.
2. Create a `Simulink.AliasType` object named `myInt`. In this case, because `int_T` and `int32_T` represent a 32-bit signed integer, set `BaseType` to `int32`.

   ```
   myInt = Simulink.AliasType('int32')
   ```
3. Configure the same data type replacement for `int32_T` and `int_T`.

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	<empty>
single	real32_T	<empty>
int32	int32_T	myInt
int16	int16_T	<empty>
int8	int8_T	<empty>
uint32	uint32_T	<empty>
uint16	uint16_T	<empty>
uint8	uint8_T	<empty>
boolean	boolean_T	<empty>
int	int_T	myInt
uint	uint_T	<empty>
char	char_T	<empty>
uint64	uint64_T	<empty>
int64	int64_T	<empty>

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

Many-to-one data type replacement is not supported for Simulink Coder data types of different sizes.

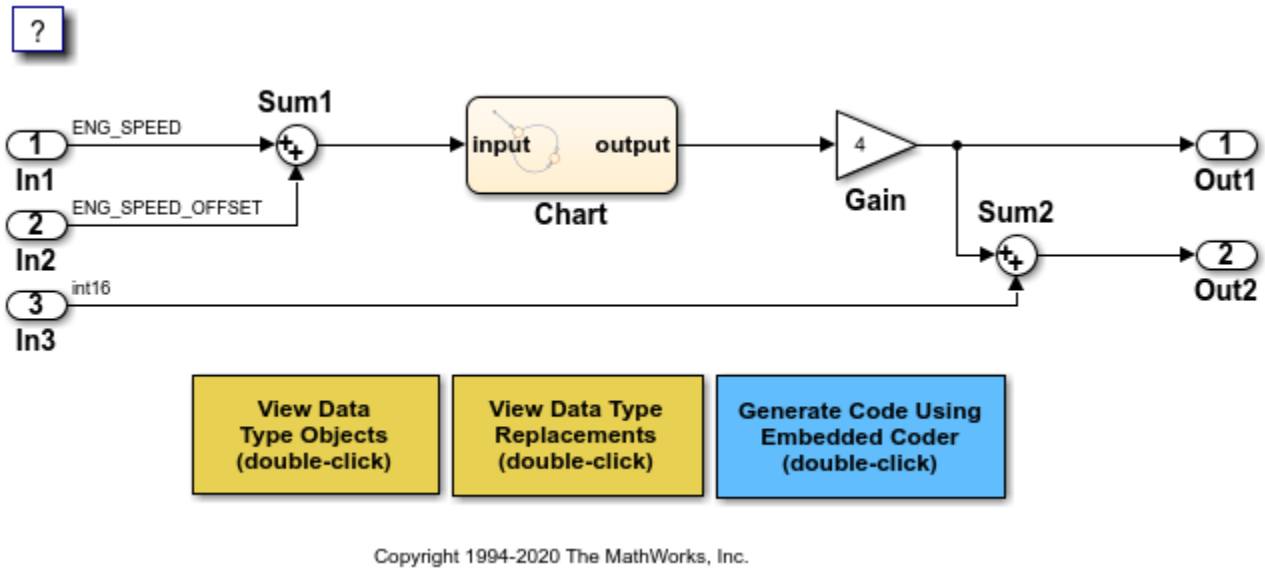
Define Abstract Numeric Types and Rename Types

This model shows user-defined types, consisting of numeric and alias types. Numeric types allow you to define abstract numeric types, which is particularly useful for fixed-point types. Alias types allow you to rename types, which allows you create a relationship for types.

Explore Example Model

Open the example model and configure it to show the generated names of blocks.

```
load_system('UserDefinedDataTypes')
set_param('UserDefinedDataTypes','HideAutomaticNames','off')
open_system('UserDefinedDataTypes')
```



Key Features of User-Defined Types

- Displayed and propagated on signal lines
- Used to parameterize a model by type (e.g., In1 specifies its **Output data type** as `ENG_SPEED`)
- Types with a common ancestor can be mixed, whereby the common ancestor is propagated (e.g., output of Sum1)
- Intrinsically supported by the Simulink Model Explorer
- Include an optional header file attribute that is ideal for importing legacy types (ignored for GRT targets)
- Types used in the generated code (ignored for GRT targets)

Instructions

1. Inspect the user-defined types in the Model Explorer by double-clicking the first yellow button below.
2. Inspect the replacement data type mapping by double-clicking the second yellow button below.
3. Compile the diagram to display the types in this model (Debug > Update Model > Update Model or **Ctrl+D**).
4. Generate code with the blue button below and inspect model files to see how user-defined types appear in the generated code.
5. Modify the attributes of `ENG_SPEED` and `ENG_SPEED_OFFSET` and repeat steps 1-4.

Notes

- User-defined types are a feature of Simulink that facilitate parameterization of the data types in a model. Embedded Coder preserves alias data type names (e.g., `ENG_SPEED`) in the generated code, whereas Simulink Coder implements user-defined types as their base type (e.g., `real32_T`).

- Embedded Coder also enables you to replace the built-in data types with user-defined data types in the generated code.

Control Names of Structure Types

To control the names of the standard structures that Simulink Coder creates by default to store data (*model*_P for parameter data, for example), use **Configuration Parameters** > **Code Generation** > **Identifiers** > **Global types** to specify a naming rule. For more information, see “Identifier Format Control” on page 55-17.

When you use nonvirtual buses and parameter structures to aggregate signals and block parameters into a custom structure in the generated code, control the name of the structure type by creating a `Simulink.Bus` object. For more information, see “Organize Data into Structures in Generated Code” on page 36-147.

Generate Code That Reuses Data Types From External Code

To generate code that reuses a data type definition from your external C code, specify the data scope of the corresponding data type object or enumeration in Simulink as `Imported`. With this setting, the generated code includes (`#include`) the definition from your code. For more information about controlling the file placement of a custom data type, see “Control File Placement of Custom Data Types” on page 38-21.

Instead of creating individual data type objects and enumerated types, and then configuring them, consider creating the objects and types by using the `Simulink.importExternalCTypes` function. By default, the function configures the new objects and types so that the generated code imports (reuses) the type definitions from your code. You can then use the objects and types to set data types in a model and to configure data type replacements. For more information, see `Simulink.importExternalCTypes` and “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30.

Create Data Type Alias in the Generated Code

This example shows how to configure the generated code to use a data type name (`typedef`) that you specify.

Export Type Definition

When you integrate code generated from a model with code from other sources, your model code can create an exported `typedef` statement. Therefore, all of the integrated code can use the type. This example shows how to export the definition of a data type to a generated header file.

Create a `Simulink.AliasType` object named `mySingleAlias` that acts as an alias for the built-in data type `single`.

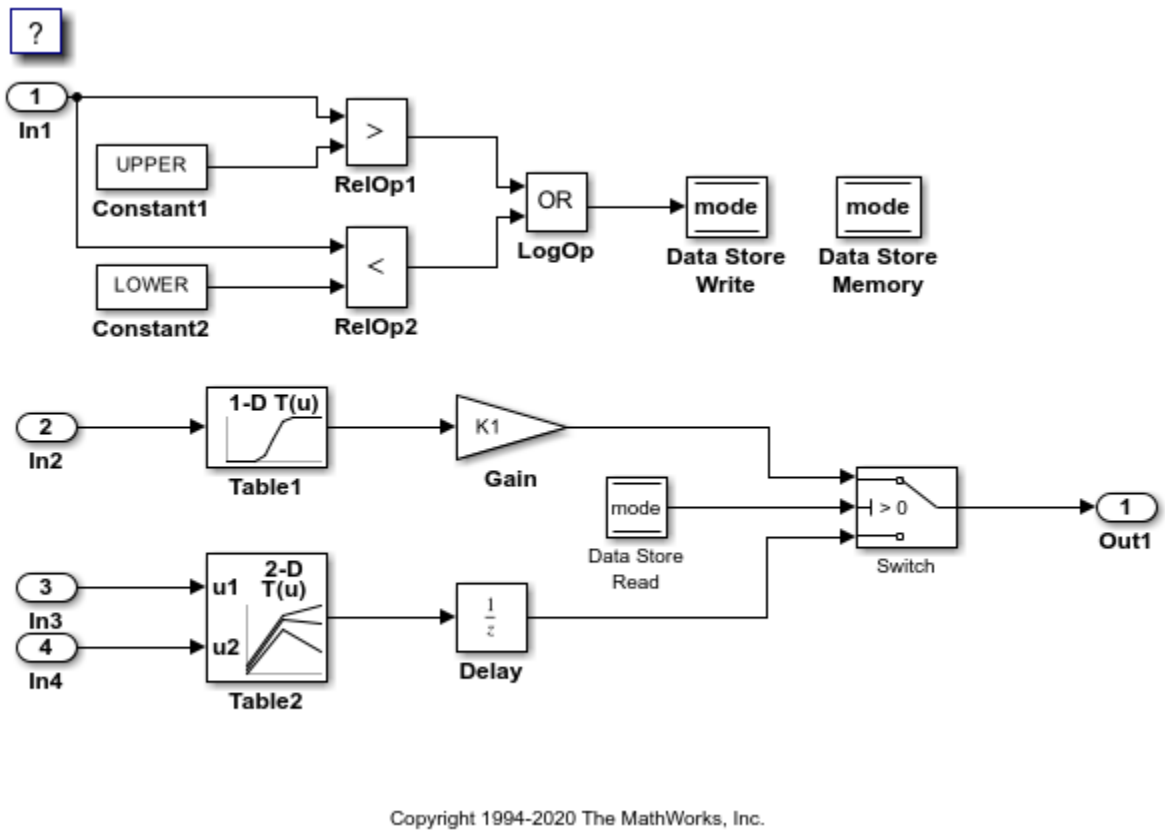
```
mySingleAlias = Simulink.AliasType('single');
```

Configure the object to export its definition to a header file called `myHdrFile.h`.

```
mySingleAlias.DataScope = 'Exported';
mySingleAlias.HeaderFile = 'myHdrFile.h';
```

Open the model `rtwdemo_configinterface`.

```
open_system('rtwdemo_configinterface')
```



Configure the model to show the generated names of blocks.

```
set_param('rtwdemo_configinterface','HideAutomaticNames','off')
```

On the **Modeling** tab, click **Model Data Editor**.

In the model, select the Inport block labeled `In1`.

Use the **Data Type** column to set the data type to `mySingleAlias`.

```
set_param('rtwdemo_configinterface/In1','OutDataTypeStr','mySingleAlias')
```

Configure `In1` to use default storage.

In the **C Code** tab, select **Code Interface > Default Code Mappings**.

In the Code Mappings editor, under **Inports and Outports**, select category **Inports**. Set the default storage class to `Default`.

On the **Inports** tab, set **Storage Class** to `Model default`.

```
cm = coder.mapping.api.get('rtwdemo_configinterface');
setDataDefault(cm,'Inports','StorageClass','Default');
setInport(cm,'In1','StorageClass','Model default');
```

Generate code from the model.

```
slbuild('rtwdemo_configinterface')
```

```
### Starting build procedure for: rtwdemo_configinterface
### Successful completion of code generation for: rtwdemo_configinterface

Build Summary

Top model targets built:

Model                     Action           Rebuild Reason
========================================================================================
rtwdemo_configinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.081s
```

In the code generation report, view the file `rtwdemo_configinterface_types.h`. The code creates a `#include` directive for the generated file `myHdrFile.h`.

```
file = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface_types.h');
rtwdemodbtype(file,'#include "myHdrFile.h"',...
    '#include "myHdrFile.h"',1,1)
```

```
#include "myHdrFile.h"
```

View the file `myHdrFile.h`. The code uses the identifier `mySingleAlias` as an alias for the data type `real32_T`. By default, generated code represents the Simulink data type `single` by using the identifier `real32_T`.

The code also provides a macro guard of the form `RTW_HEADER_filename_h_`. When you export a data type definition to integrate generated code with code from other sources, you can use macro guards of this form to prevent unintentional identifier clashes.

```
file = fullfile('slprj','ert','_sharedutils','myHdrFile.h');
rtwdemodbtype(file,'#ifndef RTW_HEADER_myHdrFile_h_',...
    ' * File trailer for generated code.',1,0)
```

```
#ifndef RTW_HEADER_myHdrFile_h_
#define RTW_HEADER_myHdrFile_h_
#include "rtwtypes.h"

typedef real32_T mySingleAlias;
typedef creal32_T cmySingleAlias;

#endif                                /* RTW_HEADER_myHdrFile_h_ */

/*
```

View the file `rtwdemo_configinterface.h`. The code uses the data type alias `mySingleAlias` to define the structure field `input1`, which corresponds to the Inport block labeled `In1`.

```
file = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface.h');
rtwdemodbtype(file,...
    '/* External inputs (root inport signals with default storage) */',...
    '} ExtU_rtwdemo_configinterface_T;',1,1)
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  mySingleAlias input1;                /* '<Root>/In1' */
  MYTYPE input2;                       /* '<Root>/In2' */
  MYTYPE input3;                       /* '<Root>/In3' */
  MYTYPE input4;                       /* '<Root>/In4' */
} ExtU_rtwdemo_configinterface_T;
```

Import Type Definition

When you integrate code generated from a model with code from other sources, to avoid redundant `typedef` statements, you can import a data type definition from the external code. This example shows how to import your own definition of a data type from a header file that you create.

Use a text editor to create a header file to import. Name the file `ex_myImportedHdrFile.h`. Place it in your working folder. Copy the following code into the file.

```
#ifndef HEADER_myImportedHdrFile_h_
#define HEADER_myImportedHdrFile_h_

typedef float myTypeAlias;

#endif
```

The code uses the identifier `myTypeAlias` to create an alias for the data type `float`. The code also uses a macro guard of the form `HEADER_filename_h`. When you import a data type definition to integrate generated code with code from other sources, you can use macro guards of this form to prevent unintentional identifier clashes.

At the command prompt, create a `Simulink.AliasType` object named `myTypeAlias` that creates an alias for the built-in type `single`. The Simulink data type `single` corresponds to the C data type `float`.

```
myTypeAlias = Simulink.AliasType('single')
```

```
myTypeAlias =

  AliasType with properties:

    Description: ''
      DataScope: 'Auto'
     HeaderFile: ''
       BaseType: 'single'
```

Configure the object so that generated code imports the type definition from the header file `ex_myImportedHdrFile.h`.

```
myTypeAlias.DataScope = 'Imported';
myTypeAlias.HeaderFile = 'ex_myImportedHdrFile.h';
```

Open the model `rtwdemo_configinterface`.

```
open_system('rtwdemo_configinterface')
```

On the **Modeling** tab, click **Model Data Editor**.

In the model, select the Inport block labeled `In1`.

Use the **Data Type** column to set the data type to `myTypeAlias`.

```
set_param('rtwdemo_configinterface/In1','OutDataTypeStr','myTypeAlias')
```

Generate code from the model.

```
slbuild('rtwdemo_configinterface')
```

```
### Starting build procedure for: rtwdemo_configinterface
### Successful completion of code generation for: rtwdemo_configinterface

Build Summary

Top model targets built:

Model                     Action           Rebuild Reason
========================================================================
rtwdemo_configinterface  Code generated.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.409s
```

In the code generation report, view the file `rtwdemo_configinterface_types.h`. The code creates a `#include` directive for your header file `ex_myImportedHdrFile.h`.

```
file = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface_types.h');
rtwdemodbtype(file,'#include "ex_myImportedHdrFile.h',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#include "ex_myImportedHdrFile.h"
#include "MYTYPE.h"
#ifndef DEFINED_TYPEDEF_FOR_Table1_Type_
#define DEFINED_TYPEDEF_FOR_Table1_Type_

typedef struct {
  MYTYPE BP[11];
  MYTYPE Table[11];
} Table1_Type;

#endif

#ifndef DEFINED_TYPEDEF_FOR_Table2_Type_
#define DEFINED_TYPEDEF_FOR_Table2_Type_

typedef struct {
  MYTYPE BP1[3];
  MYTYPE BP2[3];
  MYTYPE Table[9];
} Table2_Type;
```

```
#endif
#endif                          /* RTW_HEADER_rtwdemo_configinterface_types_h_ */

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

View the file `rtwdemo_configinterface.h`. The code uses the data type alias `myTypeAlias` to define the structure field `input1`, which corresponds to the Inport block labeled `In1`.

```
file = fullfile('rtwdemo_configinterface_ert_rtw','rtwdemo_configinterface.h');
rtwdemodbtype(file,...
    '/* External inputs (root inport signals with default storage) */',...
    '} ExtU_rtwdemo_configinterface_T;',1,1)
```

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  myTypeAlias input1;                  /* '<Root>/In1' */
  MYTYPE input2;                       /* '<Root>/In2' */
  MYTYPE input3;                       /* '<Root>/In3' */
  MYTYPE input4;                       /* '<Root>/In4' */
} ExtU_rtwdemo_configinterface_T;
```

Display Base Data Types and Aliases on Model Diagram

When you display signal data types on the model diagram, you can choose to display aliases (such as `myTypeAlias`) and base data types (such as `int16`). To display aliases, on the **Debug** tab, select **Information Overlays** > **Alias Data Types**. To display the base types, select **Information Overlays** > **Base Data Types**. For more information, see “Port Data Types”.

Create a Named Fixed-Point Data Type in the Generated Code

This example shows how to create and name a fixed-point data type in generated code. You can use the name of the type to specify parameter and signal data types throughout a model and in generated code.

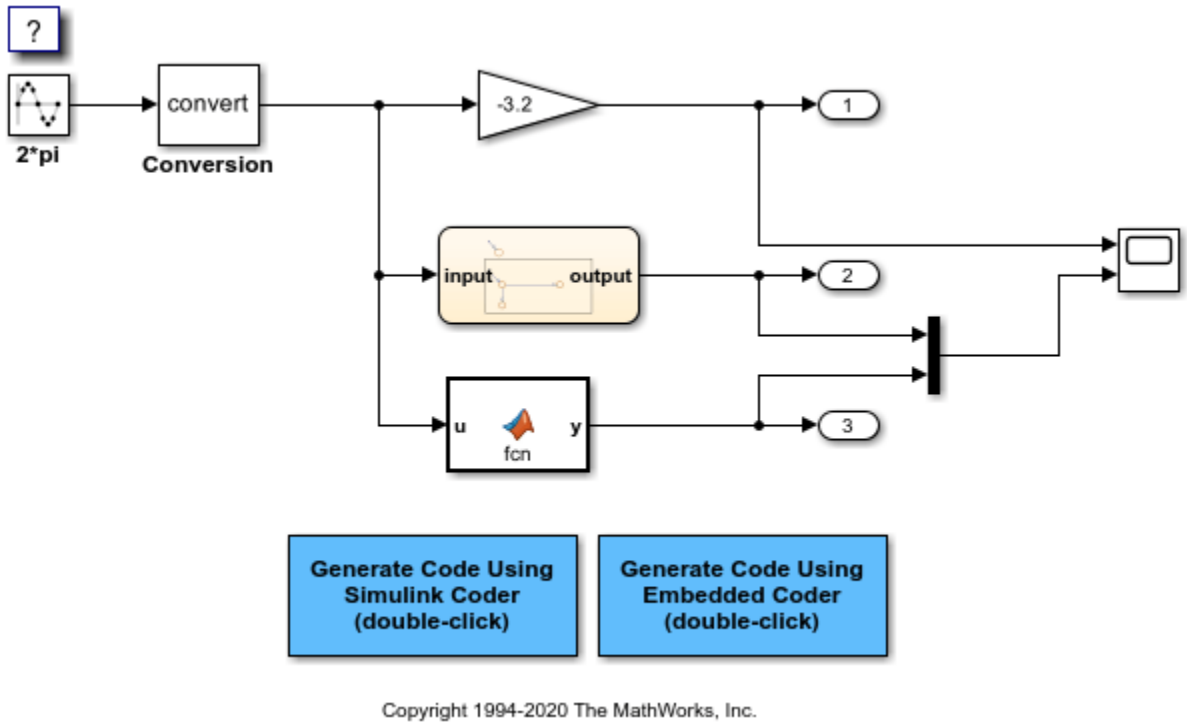
The example model `rtwdemo_fixpt1` uses fixed-point data types. So that you can more easily see the fixed-point data type in the generated code, in this example, you create a `Simulink.Parameter` object that appears in the code as a global variable.

Create a `Simulink.AliasType` object that defines a fixed-point data type. Name the object `myFixType`. The generated code uses the name of the object as a data type.

```
myFixType = Simulink.AliasType('fixdt(1,16,4)');
```

Open the model `rtwdemo_fixpt1`.

```
open_system('rtwdemo_fixpt1')
```



Configure the model to show the generated names of the blocks.

```
set_param('rtwdemo_fixpt1','HideAutomaticNames','off')
```

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, select the **Parameters** tab.

In the model, select the Gain block.

In the Model Data Editor, for the row that represents the **Gain** parameter of the Gain block, in the **Value** column, specify `myParam`.

Click the action button (with three vertical dots) next to the parameter value. Select **Create**.

In the **Create New Data** dialog box, set `Value` to `Simulink.Parameter(8)`. In this example, for more easily readable code, you set the parameter value to 8 instead of -3.2. Click **Create**. A `Simulink.Parameter` object named `myParam` appears in the base workspace. The object stores the real-world value `8`, which the Gain block uses for the value of the **Gain** parameter.

In the **Simulink.Parameter** property dialog box, set **Storage class** to `ExportedGlobal`. Click **OK**. With this setting, `myParam` appears in the generated code as a separate global variable.

In the Model Data Editor, use the **Data Type** column to set the data type of the **Gain** parameter of the Gain block to `myFixType`.

On the **Signals** tab, use the **Data Type** column to set the data type of the Gain block output to `myFixType`.

Use the **Data Type** column to set the data type of the `Conversion` block output to `myFixType`.

Alternatively, you can use these commands at the command prompt to configure the blocks and create the object:

```
set_param('rtwdemo_fixpt1/Gain','Gain','myParam','OutDataTypeStr','myFixType',...
    'ParamDataTypeStr','myFixType')
myParam = Simulink.Parameter(8);
myParam.StorageClass = 'ExportedGlobal';
set_param('rtwdemo_fixpt1/Conversion','OutDataTypeStr','myFixType')
```

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors data type aliases such as `myFixType`.

```
set_param('rtwdemo_fixpt1','SystemTargetFile','ert.tlc')
```

Select the configuration parameter **Generate code only**.

```
set_param('rtwdemo_fixpt1','GenCodeOnly','on')
```

Generate code from the model.

```
slbuild('rtwdemo_fixpt1')
```

```
### Starting build procedure for: rtwdemo_fixpt1
### Successful completion of code generation for: rtwdemo_fixpt1

Build Summary

Top model targets built:

Model           Action           Rebuild Reason
================================================================================
rtwdemo_fixpt1  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.941s
```

In the code generation report, view the file `rtwdemo_fixpt1_types.h`. The code defines the type `myFixType` based on an integer type of the specified word length (16).

```
file = fullfile('rtwdemo_fixpt1_ert_rtw','rtwdemo_fixpt1_types.h');
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_myFixType_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_myFixType_
#define DEFINED_TYPEDEF_FOR_myFixType_

typedef int16_T myFixType;
typedef cint16_T cmyFixType;

#endif
```

View the file `rtwdemo_fixpt1.c`. The code uses the type `myFixType`, which is an alias of the integer type `int16`, to define the variable `myParam`.

```
file = fullfile('rtwdemo_fixpt1_ert_rtw','rtwdemo_fixpt1.c');
rtwdemodbtype(file,'myFixType myParam = 128;','myFixType myParam = 128;',1,1)
```

```
myFixType myParam = 128;               /* Variable: myParam
```

The stored integer value `128` of `myParam` is not the same as the real-world value `8` because of the scaling that the fixed-point data type `myFixType` specifies. For more information, see “Scaling” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

The line of code that represents the Gain block applies a right bit shift corresponding to the fraction length specified by `myFixType`.

```
rtwdemodbtype(file,...
    'rtwdemo_fixpt1_Y.Out1 = (myFixType)((myParam * rtb_Conversion) >> 4);',...
    'rtwdemo_fixpt1_Y.Out1 = (myFixType)((myParam * rtb_Conversion) >> 4);',1,1)
```

```
  rtwdemo_fixpt1_Y.Out1 = (myFixType)((myParam * rtb_Conversion) >> 4);
```

Rename Data Type Object

To rename a data type object such as `Simulink.AliasType` or `Simulink.Bus` after you create it (for example, to rename an alias when coding standards change or when you encounter a naming conflict), you can allow Simulink to rename the object and correct all of the references to the object that appear in a model or models. In the Model Explorer, right-click the variable and select **Rename All**. For more information, see “Rename Variables”.

Display Signal Data Types on Block Diagram

For readability, you can display signal data types directly on a block diagram. When you use custom names for primitive data types, you can choose to display the custom name (the alias), the underlying primitive, or both. See “Port Data Types”.

Data Type Replacement Limitations

- You cannot configure the generated code to use these custom C data types:
  - Array types
  - Pointer types
  - `const` or `volatile` types
- You cannot configure the generated code to use generic C primitive types such as `int` and `short`.

When you set **Data type replacement** to `Use C data types with fixed-width integers`, these limitations apply:

- The **Specify custom data type names** configuration parameter is not supported.
- Word size values for production integer data types must include the values 8, 16, and 32 bits and each value can only be 8, 16, 32, or 64 bits. If `ProdEqTarget` is set to `'off'`, the same requirement applies to test hardware integer data types.
- You cannot generate AUTOSAR code.
- You cannot run external mode simulations that use TCP/IP or serial communication. Run external mode simulations that use XCP communication.

- You cannot generate code for third-party hardware workflows that use Embedded Coder support packages.
- In generated C++ code, statements that include the header files `stdint.h` and `stdbool.h` violate MISRA C++:2008 rule 18-0-1.
- If the model includes Simscape blocks, some of the generated source files might contain `#include "rtwtypes.h"`. Generated code from Simscape blocks is not intended for production deployment. The build process can compile the generated code because the code generator creates a `rtwtypes.h` file for continuous-time models.

When you set **Data type replacement** to `Use coder typedefs` and select the **Specify custom data type names** configuration parameter, these limitations apply:

- Data type replacement does not support multiple levels of mapping. Each replacement data type name maps directly to one or more built-in data types.
- Data type replacement is not supported for simulation target code generation for referenced models.
- If you select the **Classic call interface** configuration parameter for your model, data type replacement is not supported.
- Code generation performs data type replacements while generating `.c`, `.cpp`, and `.h` files. Code generation places these files in build folders (including top and referenced model build folders) and in the `_sharedutils` folder. *Exceptions* are as follows:
  `rtwtypes.h`
  `multiword_types.h`
  `model_reference_types.h`
  `builtin_typeid_types.h`
  `model_sf.c` or `.cpp` (ERT S-function wrapper)
  `model_dt.h` (C header file supporting external mode)
  `model_capi.c` or `.cpp`
  `model_capi.h`
- Data type replacement is not supported for complex data types.
- Many-to-one data type replacement is not supported for the `char` data type. Attempting to use `char` as part of a many-to-one mapping to a custom data type represents a violation of the MISRA C specification. For example, if you map `char` (`char_T`) and either `int8` (`int8_T`) or `uint8` (`uint8_T`) to the same replacement type, the result is a MISRA C violation. If you try to generate C++ code, the code generator makes invalid implicit type casts, resulting in compile-time errors. Use `char` only in one-to-one data type replacements.

  Many-to-one data type replacement is not supported for Simulink Coder data types of different sizes. For more information, see “Replace Implementation-Dependent Types with the Same Type Name” on page 38-7.
- For ERT S-functions, replace the `boolean` data type with only an 8-bit integer, `int8`, or `uint8`.
- Set the `DataScope` property of a `Simulink.AliasType` object to `Auto` (default) or `Imported`.

See Also

Related Examples

- “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30

- “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24

Control File Placement of Custom Data Types

By default, when you use data type objects, such as `Simulink.AliasType` and `Simulink.Bus`, and custom enumerations to specify data types for signals and block parameters, the code generated from the model defines the types (for example, with `typedef` statements). To ease integration of the generated code with other existing code and to modularize the generated code, you can control the file placement of the type definitions by adjusting the properties of the objects and the enumerations.

Data Scope and Header File

To control the file placement of a custom type definition in generated code, set the `DataScope` and `HeaderFile` properties of the data type object according to the table. Similarly, for an enumeration that you define in a MATLAB file, set the return arguments of the `getDataScope` and `getHeaderFile` methods.

- *`typename`* is the name of the custom data type.
- *`filename`* is the name of a header file.
- *`model`* is the name of the model.

Goal	Specify `DataScope` as	Specify `HeaderFile` as
Export type definition to *`model`*`_types.h`	`Auto`	Empty
Import type definition from a header file that you create, *`filename`*	`Auto` or `Imported`	*`filename`* (if necessary, include a `.h` extension)
Export type definition to a generated header file, *`filename`*`.h`	`Exported`	*`filename`* or *`filename`*`.h`
Export type definition to a generated header file, *`typename`*`.h`	`Exported`	Empty

When you import a data type definition (for example, by setting `DataScope` to `Imported`), the generated model code creates an `#include` directive for your header file in place of a type definition. You supply the header file that contains the definition. To avoid linker errors, you must add include guards, such as `#pragma once`, to the beginning of an imported header file.

Considerations for Data Type Replacement

- If you use the **Specify custom data type names** configuration parameter to replace a built-in Simulink Coder data type with your own data type in generated code (see “Model Configuration Parameters: Code Generation Data Type Replacement”), `typedef` statements and `#include` directives appear in `rtwtypes.h` instead of *`model`*`_types.h`.
- When you use a `Simulink.AliasType` or `Simulink.NumericType` object in data type replacement, you cannot set the `DataScope` property of the object to `Exported`. Therefore, if you want the code generator to generate the corresponding `typedef` statement, you cannot control the file placement of the statement. However, you can set `DataScope` to `Imported`, which means that you can configure the code to reuse the `typedef` statement that your external code provides.

  As a workaround, instead of using the data type object as a data type replacement, use the object to set the data types of individual data items in a model. To configure many data items, you can

use the Model Data Editor and take advantage of data type propagation and inheritance. For more information, see “Control Names of Simulink Coder Primitive Data Types” on page 38-3.

Import Definition of Numerically Complex Data Type

You can use a `Simulink.AliasType` object with numerically complex data (`i`). In this case, if you configure the generated code to import the type definition from your external code (for example, by setting the `DataScope` property to `Imported`), your code must provide two complementary `typedef` statements.

Suppose your external header file `myAliasTypes.h` defines the data type `IAT_int32` as an alias of a 32-bit integer. The file must define two types: `IAT_int32` and `cIAT_int32`:

```
#ifndef myAliasTypes_H_
#define myAliasTypes_H_

#include "rtwtypes.h"

typedef int32_T IAT_int32;
typedef cint32_T cIAT_int32;

#endif
```

You do not need to create two `Simulink.AliasType` objects. In this example, you create one object, `IAT_int32`. The generated code then creates complex data (variables) by using both `IAT_int32` and `cIAT_int32`.

Macro Guards

When you export one or more data type definitions to a generated header file, the file contains a file-level macro guard of the form `RTW_HEADER_`*`filename`*`_h`.

Suppose that you use several `Simulink.AliasType` objects: `mySingleAlias`, `myDoubleAlias`, and `myIntAlias` with these properties:

- `DataScope` set to `Exported`
- `HeaderFile` set to `myTypes.h`

When you generate code, the guarded file `myTypes.h` contains the `typedef` statements:

```
#ifndef RTW_HEADER_myTypes_h_
#define RTW_HEADER_myTypes_h_
#include "rtwtypes.h"

typedef real_T myDoubleAlias;
typedef real32_T mySingleAlias;
typedef int16_T myIntAlias;

#endif
```

When you export data type definitions to *`model`*`_types.h`, the file contains a macro guard of the form `_DEFINED_TYPEDEF_FOR_`*`typename`*`_` for each `typedef` statement. Suppose that you use a `Simulink.AliasType` object `mySingleAlias` with these properties:

- `DataScope` set to `Auto`

- `HeaderFile` not specified

When you generate code, the file *`model`*`_types.h` contains the guarded `typedef` statement:

```
#ifndef _DEFINED_TYPEDEF_FOR_mySingleAlias_
#define _DEFINED_TYPEDEF_FOR_mySingleAlias_

typedef real32_T mySingleAlias;

#endif
```

See Also

`Simulink.AliasType` | `Simulink.NumericType` | `Simulink.Bus`

Related Examples

- "Manage Replacement of Simulink Data Types in Generated Code" on page 38-2

Replace and Rename Simulink Coder Data Types to Conform to Coding Standards

If the **Data type replacement** configuration parameter is set to `Use coder typedefs`, the generated code uses Simulink Coder data type aliases such as `real_T` and `int32_T`. The code uses these aliases to define global and local variables. If your coding standards require that you use other data type aliases, including aliases that your existing code defines, you can:

- Configure data type replacement for the entire model.
- Configure individual data items (such as signals, parameters, and states) to use specific data type aliases.

For information about controlling data types in a model, see “Control Data Types of Signals”.

Inspect External C Code

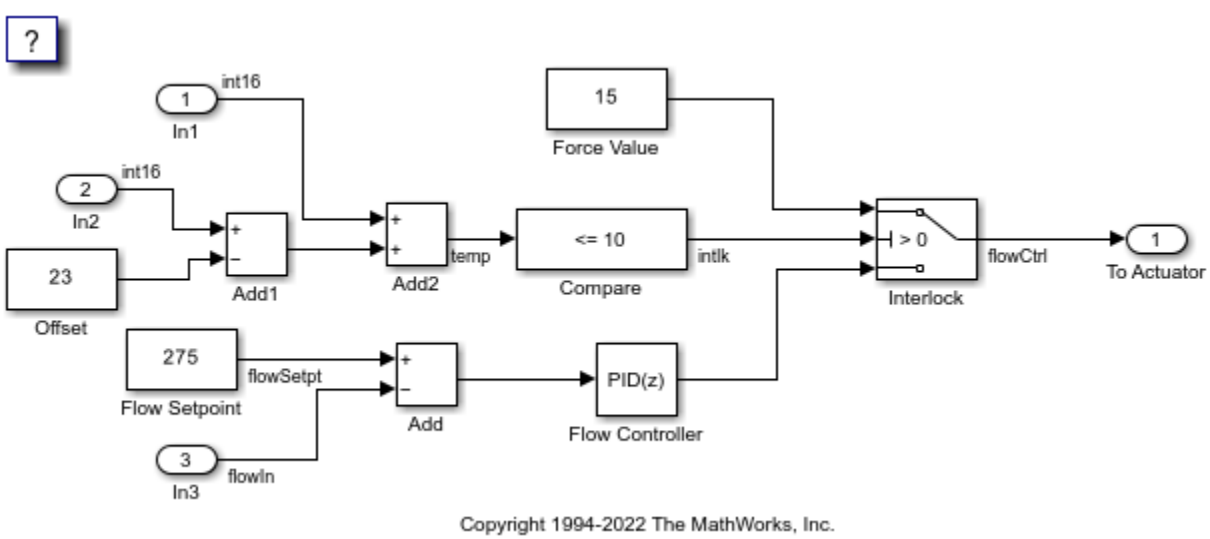
Save this C code into a file named `my_types.h` in your current folder. This file represents a header file in your existing code that defines custom data type aliases by using `typedef` statements.

```
#include <stdbool.h>

typedef double my_dblPrecision;
typedef short my_int16;
typedef bool my_bool;
```

Explore Example Model and Default Generated Code

Open the example model `ex_data_type_replacement`

```
open_system('ex_data_type_replacement')
```



1. Open the **Embedded Coder** app.
2. Select **Code Interface** > **Default Code Mappings**.
3. On the **Signals/States** tab, the **Storage Class** column shows that named signal lines such as `temp` use the storage class `ExportedGlobal`. With this setting, the signal lines appear in the generated code as separate global variables.
4. On the **Modeling** tab, click **Model Data Editor**.
5. In the Model Data Editor, inspect the **Signals** tab.
6. Set the **Change view** drop-down list to `Design`.
7. Update the block diagram. Simulink assigns a specific data type to each signal that uses an inherited data type setting such as `Inherit: Inherit via internal rule`.
8. In the data table, expand the width of the **Data Type** column. The column shows that the signals in the model use a mix of the data types `int16`, `double`, and `boolean`. You can also see these data types in the model.
9. Generate code for the model.

In the code generation report, inspect the shared utility file `rtwtypes.h`. The code uses `typedef` statements to rename the primitive C data types by using standard Simulink Coder™ aliases. For example, the code renames the primitive type `double` by using the alias `real_T`.

```
typedef double real_T;
```

Inspect the file `ex_data_type_replacement.c`. The code uses the default data type aliases to declare and define variables. For example, the code uses the data types `real_T`, `int16_T`, and `boolean_T` to define the global variables `flowIn`, `temp`, and `intlk`.

```
real_T flowIn;       /* '<Root>/In3' */
int16_T temp;        /* '<Root>/Add2' */
boloean_T intlk;     /* '<S1>/Compare' */
```

The model `step` function defines local variables by using the same data type aliases.

```
real_T rtb_Add;
real_T rtb_FilterCoefficient;
```

Reuse External Data Type Definitions

Configure the generated code to define variables by using the custom data types defined in `my_types.h` instead of the default type names.

1. At the command prompt, create a `Simulink.AliasType` object for each data type alias that your external code defines.

   ```
   Simulink.importExternalCTypes('my_types.h');
   ```

   In the base workspace, the `Simulink.importExternalCTypes` function creates the objects `my_dblPrecision`, `my_int16`, and `my_bool`.

   For each object, the function sets the `DataScope` property to `'Imported'` and the `HeaderFile` property to `'my_types.h'`. With these settings, the code generator does not create a `typedef` statement for each object, but instead the generated code reuses the statements from `my_types.h` by including (`#include`) the file.

2. In the model's Configuration Parameters dialog box, from the **Data type replacement** drop-down list, select `Use coder typedefs`.
3. Select the **Specify custom data type** names check box.
4. In the **Replacement Name** column, specify the values provided in this table.

Simulink Name	Replacement Name
`double`	`my_dblPrecision`
`int16`	`my_int16`
`boolean`	`my_bool`

5. Generate code from the model.
6. In the code generation report, inspect the file `rtwtypes.h`. Now, the code uses an `#include` directive to import the contents of the external header file `my_types.h`, which contains the custom data type aliases.

```
#include "my_types.h"     /* User defined replacement datatype for int16_T real_T boolean_T  */
```

7. Inspect the file `ex_data_type_replacement.c`. The code uses the custom data type aliases `my_dblPrecision`, `my_int16`, and `my_bool` to define the global variables such as `flowIn`, `temp`, and `intlk`.

```
my_dblPrecision flowIn;                /* '<Root>/In3' */
my_int16 temp;                         /* '<Root>/Add2' */
my_bool intlk;                         /* '<S1>/Compare' */
```

The model `step` function defines local variables by using the custom data type aliases.

```
my_dblPrecision rtb_Add;
my_dblPrecision rtb_FilterCoefficient;
```

Create Meaningful Data Type Aliases for Individual Data Items

Suppose that your coding standards require that important data items use a data type whose name indicates the real-world significance. You can create more `Simulink.AliasType` objects to represent these custom data type aliases. Use the objects to set the data types of data items in the model.

1. In the Model Data Editor **Data Type** column, use the `Simulink.AliasType` objects to set the data types of these block outputs. You can select a signal line in the model, which highlights the corresponding row in the Model Data Editor, or search for the signals in the Model Data Editor by using the **Filter contents** box.

Block	Output Data Type
Flow Setpoint	`flow_T`
Add2 (which feeds the Compare block)	`diffTemp_T`

2. Use the Model Data Editor **Inports/Outports** tab to set the data type of the third Inport block to `flow_T`.
3. On the **Modeling** tab, under **Design**, click **Property Inspector**.
4. Select the block labeled Flow Controller. In the Property Inspector, click **Open** to open the block dialog box.

5. On the **Data Types** tab, set **Sum output** to `ctrl_T`. Click **OK**.

   With these data type settings, some of the named signals, such as `temp` and `flowIn`, use data types that evoke real-world quantities, such as liquid flow rate and temperature.

6. At the command prompt, create `Simulink.AliasType` objects to represent these new custom data type aliases.

   ```
   flow_T = Simulink.AliasType('double');
   diffTemp_T = Simulink.AliasType('int16');
   ctrl_T = Simulink.AliasType('double');
   ```

   In the model, the signals `flowIn` and `flowSetPt` use the primitive data type `double`, so the data type alias `flow_T` maps to `double`.

7. Update the block diagram.

   Due to data type inheritance, other signals also use the custom data type aliases. For example, the output data type of the Add block fed by the third Inport block is set to `Inherit: Inherit via internal rule`. The internal rule chooses the same data type that the block inputs use, `flow_T`.

8. Generate code from the model.

9. The file `ex_data_type_replacement_types.h` defines the new data types `flow_T`, `diffTemp_T`, and `ctrl_T` as aliases of `my_dblPrecision` and `my_int16`.

   ```
   typedef my_dblPrecision flow_T;
   typedef my_int16 diffTemp_T;
   typedef my_dblPrecision ctrl_T;
   ```

10. In the file `ex_data_type_replacement.c`, the code defines global variables by using the new type names.

    ```
    flow_T flowIn;                        /* '<Root>/In3' */
    flow_T flowSetPt;                     /* '<Root>/Flow Setpoint' */
    ctrl_T flowCtrl;                      /* '<Root>/Interlock' */
    diffTemp_T temp;                          /* '<Root>/Add2' */
    ```

11. For blocks that do not use the new data types, the corresponding generated code continues to use the replacement types that you specified earlier. For example, in the file `ex_data_type_replacement.h`, the outputs of the blocks In1 and In2 appear as structure fields that use the replacement type `my_int16`.

    ```
    /* External inputs (root inport signals with auto storage) */
    typedef struct {
      my_int16 In1;                       /* '<Root>/In1' */
      my_int16 In2;                       /* '<Root>/In2' */
    } ExtU_ex_data_type_replacement_T;
    ```

Create Single Point of Definition for Primitive Types

The custom data type aliases `flow_T` and `ctrl_T` map to the primitive data type `double`. If you want to change this underlying data type from `double` to `single` (`float`), you must remember to modify the `BaseType` property of both `Simulink.AliasType` objects.

To more easily make this change, you can create a `Simulink.NumericType` object and configure both `Simulink.AliasType` objects to refer to it. Then, you need to modify only the

`Simulink.NumericType` object. A `Simulink.NumericType` object enables you to share a data type without creating a data type alias.

1. At the command prompt, create a `Simulink.NumericType` object to represent the primitive data type `single`.

   ```
   sharedType = Simulink.NumericType;
   sharedType.DataTypeMode = 'Single';
   ```

2. Configure the `Simulink.AliasType` objects `flow_T` and `ctrl_T` to acquire an underlying data type from this new object.

   ```
   flow_T.BaseType = 'sharedType';
   ctrl_T.BaseType = 'sharedType';
   ```

3. On the **Debug** tab, select **Information Overlays** > **Base Data Types Information Overlays** > **Alias Data Types**and (see “Port Data Types”). Update the block diagram.

   The data type indicators in the model show that the aliases `flow_T` and `ctrl_T` map to the primitive type `single`. To change this underlying primitive type, you can modify the `DataTypeMode` property of the `Simulink.NumericType` object, `sharedType`.

By default, the `Simulink.NumericType` object does not cause another `typedef` statement to appear in the generated code.

If you generate code from the model while the `Simulink.NumericType` object represents the data type `single`, the generated code maps `flow_T` and `ctrl_T` to the default Simulink Coder data type alias `real32_T`, which maps to the C data type `float`. You can replace `real32_T` in the same way that you replaced `real_T`, `int16_T`, and `boolean_T` (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**).

Permanently Store Data Type Objects

The `Simulink.NumericType` and `Simulink.AliasType` objects in the base workspace do not persist if you end your current MATLAB session. To permanently store these objects, consider migrating your model to a data dictionary. See “Migrate Models to Use Simulink Data Dictionary”.

Create and Maintain Objects Corresponding to Multiple C typedef Statements

To create `Simulink.AliasType` objects for a large number of `typedef` statements in your external C code, consider using the `Simulink.importExternalCTypes` function.

See Also

Related Examples

- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “Model Configuration Parameters: Code Style”
- “Unit Specification in Simulink Models”
- “Design Data Interface by Configuring Inport and Outport Blocks” on page 36-169

- “Choose an External Code Integration Workflow” on page 58-2
- “Control File Placement of Custom Data Types” on page 38-21

Exchange Structured and Enumerated Data Between Generated and External Code

This example shows how to generate code that exchanges data with external, existing code. Construct and configure a model to match data types with the external code and to avoid duplicating type definitions and memory allocation (definition of global variables). Then, compile the generated code together with the external code into a single application.

Inspect External Code

Create the file `ex_cc_algorithm.c` in your current folder.

```
#include "ex_cc_algorithm.h"

inSigs_T inSigs;

float_32 my_alg(void)
{
    if (inSigs.err == TMP_HI) {
        return 27.5;
    }
    else if (inSigs.err == TMP_LO) {
        return inSigs.sig1 * calPrms.cal3;
    }
    else {
        return inSigs.sig2 * calPrms.cal3;
    }
}
```

The C code defines a global structure variable named `inSigs`. The code also defines a function, `my_alg`, that uses `inSigs` and another structure variable named `calPrms`.

Create the file `ex_cc_algorithm.h` in your current folder.

```
#ifndef ex_cc_algorithm_h
#define ex_cc_algorithm_h

typedef float float_32;

typedef enum {
    TMP_HI = 0,
    TMP_LO,
    NORM,
} err_T;

typedef struct inSigs_tag {
    err_T err;
    float_32 sig1;
    float_32 sig2;
} inSigs_T;

typedef struct calPrms_tag {
    float_32 cal1;
    float_32 cal2;
```

```
    float_32 cal3;
} calPrms_T;

extern calPrms_T calPrms;
extern inSigs_T inSigs;

float_32 my_alg(void);

#endif
```

The file defines `float_32` as an alias of the C data type `float`. The file also defines an enumerated data type, `err_T`, and two structure types, `inSigs_T` and `calPrms_T`.

The function `my_alg` is designed to calculate a return value by using the fields of `inSigs` and `calPrms`, which are global structure variables of the types `inSigs_T` and `calPrms_T`. The function requires another algorithm to supply the signal data that `inSigs` stores.

This code allocates memory for `inSigs`, but not for `calPrms`. Create a model whose generated code:

- Defines and initializes `calPrms`.
- Calculates values for the fields of `inSigs`.
- Reuses the type definitions (such as `err_T` and `float_32`) that the external code defines.

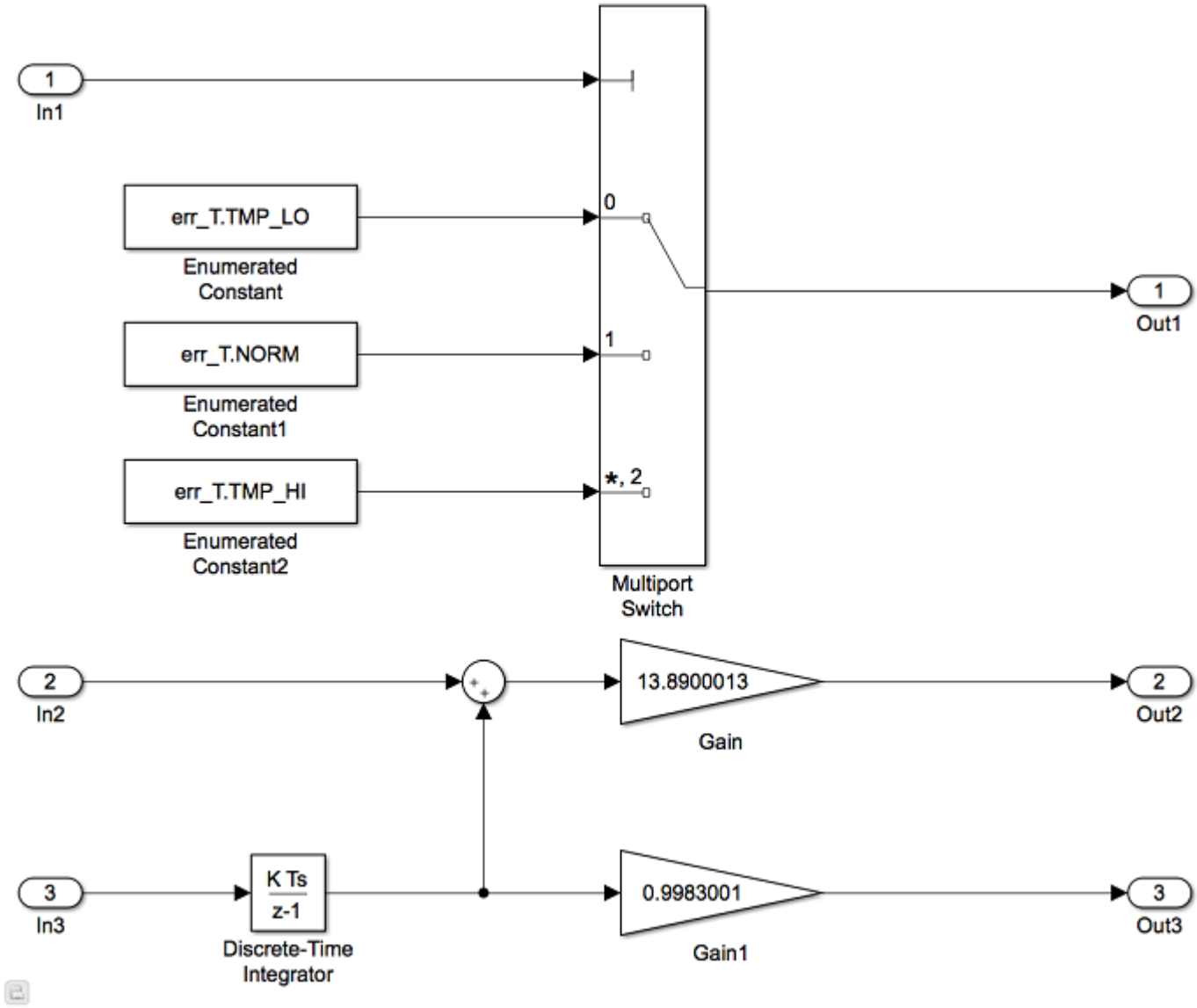
Create Simulink Model

1. So that you can create enumerated and structured data in the Simulink model, first create Simulink representations of the data types that the external code defines. Store the Simulink types in a new data dictionary named `ex_cc_integ.sldd`.

   ```
   Simulink.importExternalCTypes('ex_cc_algorithm.h',...
       'DataDictionary','ex_cc_integ.sldd');
   ```

   The data dictionary appears in your current folder.
2. To inspect the dictionary contents in the Model Explorer, in your current folder, double-click the file, `ex_cc_integ.sldd`.

   The `Simulink.importExternalCTypes` function creates `Simulink.Bus`, `Simulink.AliasType`, and `Simulink.data.dictionary.EnumTypeDefinition` objects that correspond to the custom C data types from `ex_cc_algorithm.h`.
3. Create a new model and save it in your current folder as `ex_struct_enum_integ`.
4. Link the model to the data dictionary. On the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
5. Add algorithmic blocks that calculate the fields of `inSigs`.



Now that you have the algorithm model, you must:

- Organize the output signals into a structure variable named `inSigs`.
- Create the structure variable `calPrms`.
- Include `ex_cc_algorithm.c` in the build process that compiles the code after code generation.

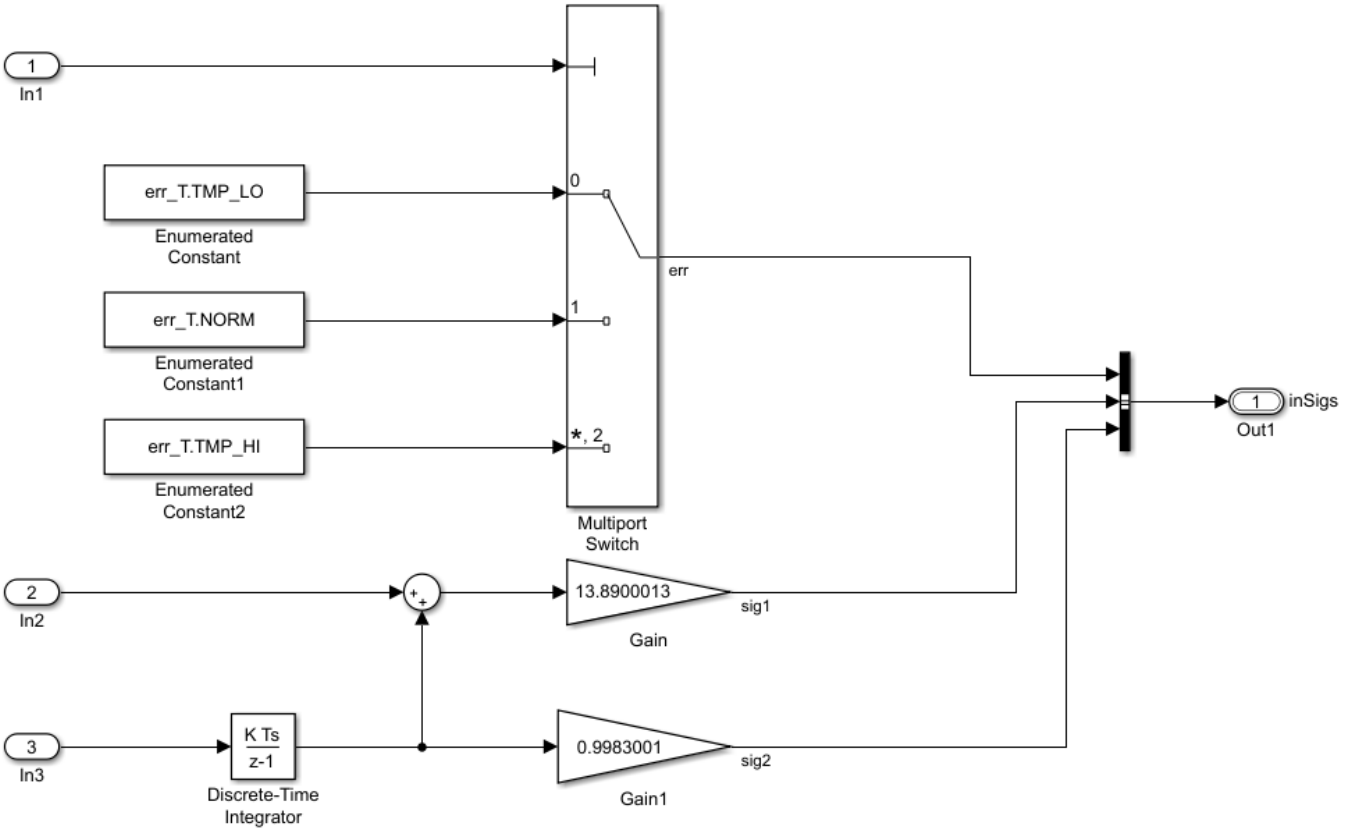
Configure Generated Code to Write Output Data to Existing Structure Variable

1. Add a Bus Creator block near the existing Outport blocks. The output of a Bus Creator block is a bus signal, which you can configure to appear in the generated code as a structure.
2. In the Bus Creator block, set these parameters:

- **Number of inputs** to 3
- **Output data type** to `Bus: inSigs_T`
- **Output as nonvirtual bus** to selected

3. Delete the three existing Outport blocks (but not the signals that enter the blocks).
4. Connect the three remaining signal lines to the inputs of the Bus Creator block.
5. Add an Outport block after the Bus Creator block. Connect the output of the Bus Creator to the Outport.
6. In the Outport block, set the **Data type** parameter to `Bus: inSigs_T`.
7. On the **Modeling** tab, click **Model Data Editor**.
8. On the **Inports/Outports** tab, for the Inport blocks labeled `In2` and `In3`, change **Data Type** from `Inherit: auto` to `float_32`.
9. For the Outport block, set **Signal Name** to `inSigs`.
10. Open the **Embedded Coder** app. in the **Apps** gallery, under **Code Generation**, click **Embedded Coder**. The **C Code** tab opens.
11. Open the Code Mappings editor. Select **Code Interface** > **Individual Element Code Mappings**
12. On the **Outports** tab, select the row for the outport. Set **Storage Class** to `ImportFromFile`.
13. Click the and set **Header File** to `ex_cc_algorithm.h`.
14. In the model, select the output signal of the Multiport Switch block.
15. In the Model Data Editor, for the selected signal, set **Name** to `err`.
16. Set the name of the output signal of the Gain block to `sig1`.
17. Set the name of the output signal of the Gain1 block to `sig2`.

When you finish, the model stores output signal data (such as the signals `err` and `sig1`) in the fields of a structure variable named `inSigs`.



Because you set **Storage Class** to `ImportFromFile`, the generated code does not allocate memory for `inSigs`.

Configure Generated Code to Define Parameter Data

Configure the generated code to define the global structure variable, `calPrms`, that the external code needs.

1. In the Model Explorer **Model Hierarchy** pane, under the dictionary node **ex_cc_integ**, select the **Design Data** node.
2. In the **Contents** pane, select the `Simulink.Bus` object named `calPrms_T`.
3. In the Dialog pane (the right pane), click **Launch Type Editor**.
4. In the Type Editor, select `calPrms_T`.
5. In the Type Editor toolstrip, click **Simulink Parameter**.
6. In the MATLAB Editor, copy the generated MATLAB code and run the code at the command prompt. The code creates a `Simulink.Parameter` object in the base workspace.
7. In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.
8. Use the Model Explorer to move the parameter object, `calPrms_T_Param`, from the base workspace to the Design Data section of the data dictionary.
9. With the data dictionary selected, in the **Contents** pane, rename the parameter object as `calPrms`.

10. In the Model Data Editor, select the **Parameters** tab.
11. For the Gain block, replace the value `13.8900013` with `calPrms.cal1`.
12. In the other Gain block, use `calPrms.cal2`.
13. While editing the value of the other Gain block, next to `calPrms.cal2`, click the action button ⋮ and select **calPrms** > **Open**.
14. In the `calPrms` property dialog box, next to the **Value** box, click the action button ⋮ and select **Open Variable Editor**.
15. Use the Variable Editor to set the field values in the parameter object.

    - For the fields `cal1` and `cal2`, use the numeric values that the Gain blocks in the model previously used.
    - For `cal3`, use a nonzero number such as `15.2299995`.

    When you finish, close the Variable Editor.
16. In the property dialog box, set **Storage class** to `ExportedGlobal`. Click **OK**.
17. Use the Model Explorer to save the changes that you made to the dictionary.

Generate, Compile, and Inspect Code

1. Configure the model to include `ex_cc_algorithm.c` in the build process. Set **Configuration Parameters** > **Code Generation** > **Custom Code** > **Code information** > **Source files** to `ex_cc_algorithm.c`.
2. Generate code from the model.
3. Inspect the generated file `ex_struct_enum_integ.c`. The file defines and initializes `calPrms`.

```
/* Exported block parameters */
calPrms_T calPrms = {
  13.8900013F,
  0.998300076F,
  15.23F
} ;                                   /* Variable: calPrms
```

The algorithm then calculates and stores data in the fields of `inSig`.

Replace Data Type Names Throughout Model

To generate code that uses `float_32` instead of the default, `real32_T`, instead of manually specifying the data types of block output signals and bus elements, you can use data type replacement (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**). For more information, see “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24.

See Also

`Simulink.importExternalCTypes`

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code”
- “Manage Replacement of Simulink Data Types in Generated Code” on page 38-2
- “Use Enumerated Data in Generated Code” on page 36-86
- “Organize Data into Structures in Generated Code” on page 36-147

Specify Boolean and Data Type Limit Identifiers

You can use command-line parameters to replace the default Boolean and data type limit identifiers. If you want to associate the data type limit identifiers with the data type names, consider replacing the default identifiers. You can also use command-line parameters to import a header file with the Boolean and data type limit identifier definitions.

Data Type Limit Identifiers

You can control the data type limit identifiers in the generated code by using the command-line parameters in this table.

Data Type Limit	Default Identifier	Command-Line Parameter
64-bit integer maximum	`MAX_int64_T`	`MaxIdInt64`
16-bit integer maximum	`MAX_int16_T`	`MaxIdInt16`
32-bit integer maximum	`MAX_int32_T`	`MaxIdInt32`
8-bit integer maximum	`MAX_int8_T`	`MaxIdInt8`
64-bit unsigned integer maximum	`MAX_uint64_T`	`MaxIdUint64`
16-bit unsigned integer maximum	`MAX_uint16_T`	`MaxIdUint16`
32-bit unsigned integer maximum	`MAX_uint32_T`	`MaxIdUint32`
8-bit unsigned integer maximum	`MAX_uint8_T`	`MaxIdUint8`
64-bit integer minimum identifier	`MIN_int64_T`	`MinIdInt64`
16-bit integer minimum	`MIN_int16_T`	`MinIdInt16`
32-bit integer minimum	`MIN_int32_T`	`MinIdInt32`
8-bit integer minimum	`MIN_int8_T`	`MinIdInt8`

For example, to change the default identifiers for the 8-bit integer data limit minimum and maximum to `s4g_S4MIN` and `s4g_S4MAX`, respectively:

```
set_param(gcs,'MinIdInt8','s4g_S4MIN');
set_param(gcs,'MaxIdInt8','s4g_S4MAX')
```

If you do not import a header file, the generated file `rtwtypes.h` defines the 8-bit integer data minimum and maximum identifiers:

```
#define s4g_S4MAX                       ((int8_T)(127))
#define s4g_S4MIN                       ((int8_T)(-128))
```

If you do import a header file defining the data type limit identifiers, the header file is included in `rtwtypes.h`.

Boolean Identifiers

You can control the Boolean identifiers in the generated code by using the command-line parameters in this table. When changing boolean identifiers, you must define `false` to be numerically equivalent to `0`, and `true` to be numerically equivalent to `1`.

Boolean	Default Identifier	Command-Line Parameter
True	`true`	`BooleanTrueId`
False	`false`	`BooleanFalseId`

For example, to change the default Boolean true and false identifiers:

```
set_param(gcs,'BooleanTrueId','bTrue');
set_param(gcs,'BooleanFalseId','bFalse')
```

If you do not import a header file, the generated file `rtwtypes.h` defines the Boolean identifiers:

```
#define bFalse                          (0U)
#define bTrue                           (1U)
```

If you do import a header file defining the Boolean identifiers, the header file is included in `rtwtypes.h`.

Note When changing boolean identifiers, you must define `false` to be numerically equivalent to `0`, and `true` to be numerically equivalent to `1`.

Boolean and Data Type Limit Identifier Header Files

You can import a header file that defines Boolean and data type limit identifiers by using the configuration parameter **Type limit identifier replacement header file** or command-line parameter `TypeLimitIdReplacementHeaderFile`. The header file is included in `rtwtypes.h`. You must use the command-line parameters to specify the Boolean and data type limit identifiers that are included in the imported header file.

For example, if you have a header file `myfile.h` with data type limit definitions, use `TypeLimitIdReplacementHeaderFile` to include the definitions in the generated code:

```
set_param(gcs,'TypeLimitIdReplacementHeaderFile','myfile.h');
```

The generated file `rtwtypes.h` includes `myfile.h`.

```
/* Import type limit identifier replacement definitions. */
#include "myfile.h"
```

See Also

Related Examples

- “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24
- “Model Configuration Parameters: Code Generation Data Type Replacement”

Module Packaging Tool (MPT) Data Objects in Embedded Coder

MPT Data Object Properties

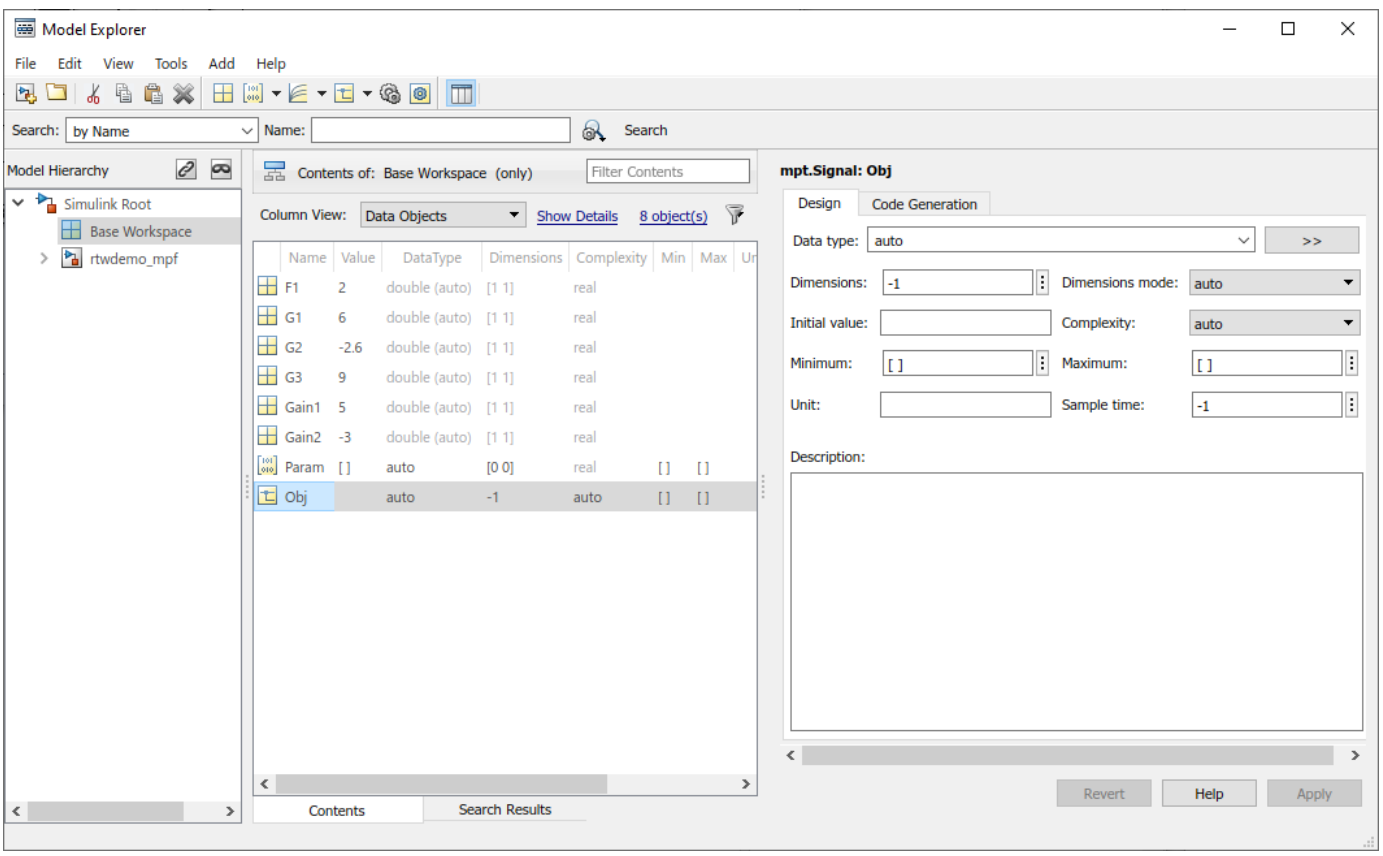
Note While `mpt.Signal` and `mpt.Parameter` objects continue to be supported for legacy applications, most of the advanced code generation capabilities that the objects provide are now also provided by `Simulink.Signal` and `Simulink.Parameter` objects.

The following table describes the properties and property values for `mpt.Parameter` and `mpt.Signal` data objects that appear in the Model Explorer.

Note You can create `mpt.Signal` and `mpt.Parameter` objects in the base MATLAB or model workspace. However, if you create a signal object in a model workspace, the object's storage class must be set to `Auto`.

The figure below shows an example of the Model Explorer. When you select an `mpt.Parameter` or `mpt.Signal` data object in the middle pane, its properties and property values display in the rightmost pane.

In the Properties column, the table lists the properties in the order in which they appear on the Model Explorer.



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Parameter	`Value`	`*0`	The data type and numeric value of the data object. For example, `int8(5)`. The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an `mpt.Signal` data object, but not for an `mpt.Parameter` data object. The data type for an `mpt.Parameter` data object is specified in the **Value** field above. See "About Data Types in Simulink".
Both	Unit	`*null`	Units of measurement of the signal or parameter. (Enter text in this field.)
Both	Dimensions	`*-1`	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.

Both	Complexity	`*auto`
		
		`real`
		
		`complex`
			Complexity specifies whether the signal or parameter is a real or complex number. Select `auto` for the code generator to decide. For a parameter, the complexity is derived from its value.

Signal	Sample time	`*-1`	Model or block execution rate.
Signal	Sample mode	`*auto`	Determines how the signal propagates through the model. Select `auto` for the code generator to decide.
		`Sample based`	The signal propagates through the model one sample at a time.
		`Frame based`	The signal propagates through the model in batches of samples.
Both	Minimum	`*0.0`	The minimum value to which the parameter or signal is expected to be bound.
		Number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
Both	Maximum	`*0.0`	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
	Storage class		Note that an `auto` selection for a storage class tells the build process to decide how to declare and store the selected parameter or signal.
Both	Global (Custom)	`Global` is the default storage class for `mpt` data objects.	Specifies that a code generator not place a qualifier in the data object's declaration.
Both	Memory section	`*Default`	**Memory section** allows you to specify storage directives for the data object. `Default` specifies that the code generator not place a type qualifier and `pragma` statement with the data object's declaration.
Parameter		`MemConst`	Places the `const` type qualifier in the declaration.
Both		`MemVolatile`	Places the `volatile` type qualifier in the declaration.
Parameter		`MemConstVolatile`	Places the `const volatile` type qualifier in the declaration.
Both	Header file		Name of the file used to import or export the data object. This file contains the declaration (`extern`) to the data object. Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the `.h` extension. For example, specify `"object.h"` or `"object"`. For the selected data object, this overrides the general delimiter selection in the **#include file delimiter** field on the Configuration Parameters dialog box.
Both	Owner	*Blank	The name of the module that owns this signal or parameter. This is used to help determine the ownership of a definition. For details, see “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2.
Both	Definition file	*Blank	Name of the file that defines the data object.
		Valid character vector	

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Persistence level		The number you specify is relative to **Signal display level** or **Parameter tune level** on the **Code Placement** pane of the Configuration Parameters dialog box. For a signal, allows you to specify whether or not the code generator declares the data object as global data. For a parameter, allows you to specify whether or not the code generator declares the data object as tunable global data. See **Signal display level** and **Parameter tune level** in "Model Configuration Parameters: Code Generation Code Placement".
Both	Bitfield (Custom)		Embeds Boolean data in a named bit field.
	Struct name		Name of the `struct` into which the object's data will be packed.
Parameter	Const (Custom)		Places the `const` type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Both	Volatile (Custom)		Places the `volatile` type qualifier in the declaration.
Both	Header file		See above.
Both	Owner		See above.
Both	Definition file		See above.
Both	Persistence level		See above.
Parameter	ConstVolatile (Custom)		Places the `const volatile` type qualifier in declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Parameter	Define (Custom)		Represents parameters with a `#define` macro.
Parameter	Header file		See above.
Both	ExportToFile (Custom)		Generates global variable definition, and generates a user-specified header (`.h`) file that contains the declaration (`extern`) to that variable.
Both	Memory section		See above.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Header file		See above.
Both	Definition file		See above.
Both	ImportFromFile (Custom)		Includes predefined header files containing global variable declarations, and places the `#include` in a corresponding file. Assumes external code defines (allocates memory) for the global variable.
Both	Data access	`*Direct`	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type (`Direct`) or stores the address of the data (a pointer).
Both		`Pointer`	If you select `Pointer`, the code generator places * before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named `struct` to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.
Both	Alias	`*null`	As explained in detail in “Apply Custom Naming Conventions to Identifiers” on page 55-11, for a Simulink or `mpt` data object (identifier), specifying a name in the **Identifier** field overrides the global naming rule selection you make on the Configuration Parameters dialog box.
		Valid ANSI[a] C/C++ variable name	
Both	`Description`	`*null`	Text description of the parameter or signal. Appears as a comment beside the signal or parameter's identifier in the generated code.
		Character vector	
Signal	Reusable		Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same storage class for both. The storage class is either `Reusable` or derived from `Reusable`.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Signal	Data Scope	`*Auto`	You can specify the scope of symbols code generation generates for a data object of this class by selecting a value for **DataScope**. When you take the default of `Auto`, code generation determines the symbol scope internally. If possible, symbols have `File` scope. Otherwise, they have `Exported` scope.
		`File`	Code generation defines the scope of each symbol as the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, code generation reports an error.
		`Exported`	Code generation exports symbols to external code in the header file specified by the **HeaderFile** field. If a **HeaderFile** is not specified, symbols are exported to external code in *`model`*`.h`.
		`Imported`	Code generation imports symbols from external code in the header file specified by the **HeaderFile** field. If you do not specify a header file, code generation generates an `extern` directive in *`model`*_private.h.
Signal	Header file		See above.
Signal	Owner		See above.
Signal	Definition file		See above.

a ANSI is a registered trademark of the American National Standards Institute, Inc.

`mpt` Package Storage Classes

CSC Name	Purpose	Signals?	Parameters?
`BitField`	Generate a `struct` declaration that embeds Boolean data in named bit fields.	Y	Y
`CompilerFlag`	Supports preprocessor conditionals defined via compiler flag. See "Generate Preprocessor Conditionals for Variant Systems" on page 27-32.	N	Y
`Const`	Generate a constant declaration with the `const` type qualifier.	N	Y
`ConstVolatile`	Generate declaration of volatile constant with the `const volatile` type qualifier.	N	Y
`Define`	Generate `#define` directive.	Y	Y
`ExportToFile`	Generate header (`.h`) file, with user-specified name, containing global variable declarations.	Y	Y
`FileScope`	Generate a static qualifier suffix for a variable declaration so that the scope of the variable is limited to the current file.	Y	Y
`GetSet`	Supports specialized function calls to read and write the memory associated with a Data Store Memory block. See "Access Data Through Functions with Storage Class GetSet" on page 40-25.	Y	Y
`Global`	The default storage class for the `mpt` package. Generate the declaration and definition of a data object in specified files, and use the specified memory section.	Y	Y
`ImportedDefine`	Supports preprocessor conditionals defined via legacy header file. See "Generate Preprocessor Conditionals for Variant Systems" on page 27-32.	N	Y
`ImportFromFile`	Generate directives to include predefined header files containing global variable declarations.	Y	Y
`Reusable`	Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same storage class for both. The storage class is either `Reusable` or derived from `Reusable`.	Y	N

CSC Name	Purpose	Signals?	Parameters?
`Struct`	Generate a `struct` declaration encapsulating parameter or signal object data.	Y	Y
`StructConst`	Generate a `struct` declaration, with a `const` type qualifier, encapsulating parameter object data.	N	Y
`StructVolatile`	Generate a `struct` declaration, with a `volatile` type qualifier, encapsulating parameter or signal object data.	Y	Y
`Volatile`	Use `volatile` type qualifier in declaration.	Y	Y

Examples of Property Value Changes on Generated Code

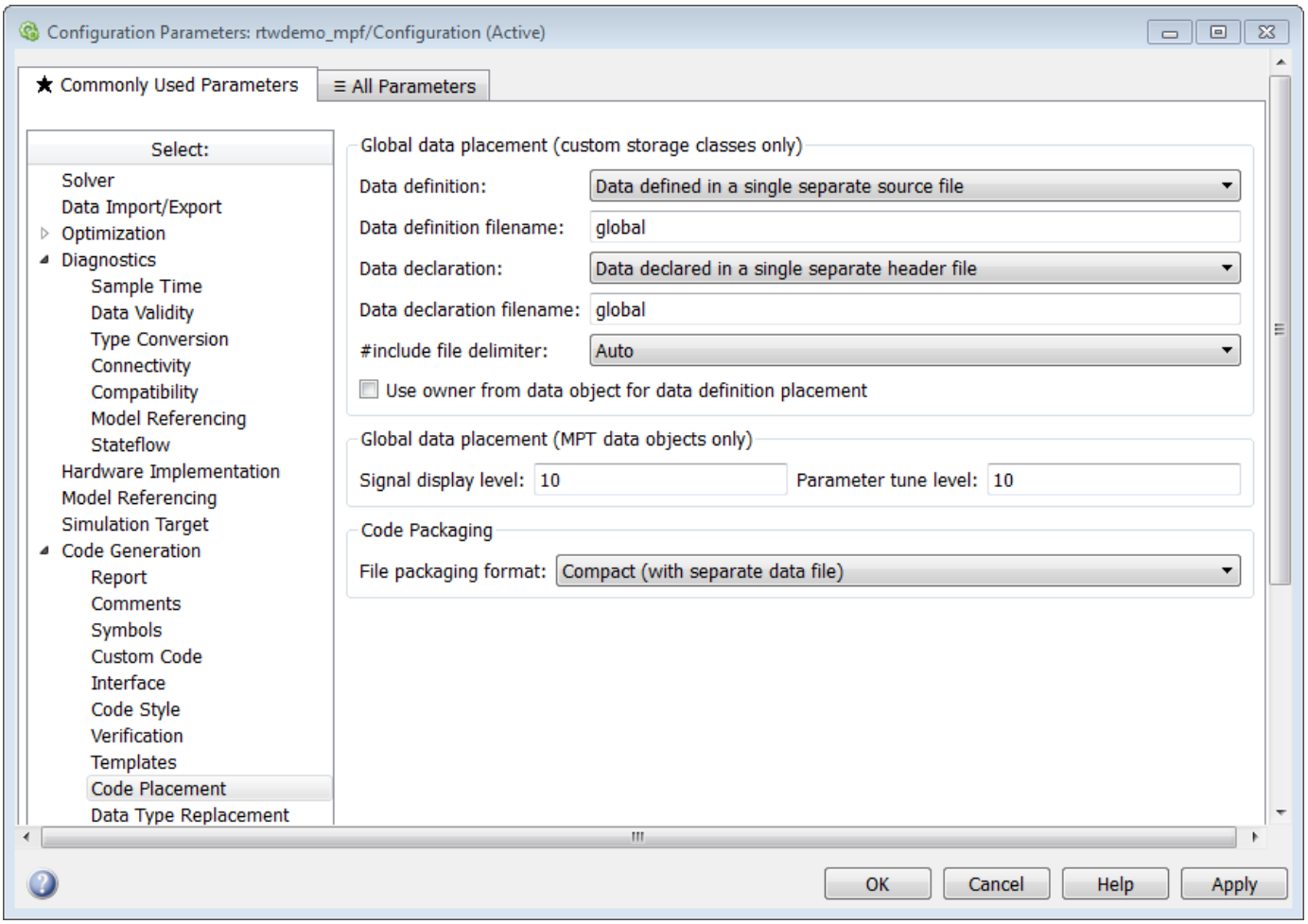
What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 1: Parameter data objects can be declared or defined as constants. I know that the data object `GAIN` is a parameter. I want this to be declared or defined in the `.c` file as a variable. But I notice that `GAIN` is declared as a constant by the statement `const real_T GAIN = 5.0;`. Also, this statement is in the constant section of the file.	In the Model Explorer, I clicked the data object `GAIN`. I noticed that the property value for its **Memory section** property is set at `MemConst`. I changed this to `Default`.	I notice two differences. One is that now `GAIN` is declared as a variable with the statement `real_T GAIN = 5.0;`. The second difference is that the declaration now is located in the `MemConst` memory section in the `.c` or `.cpp` file.
Example 2: I notice again the declaration of `GAIN` in the `.c` file mentioned in Example 1. It appears as `real_T GAIN = 5.0;`. But I have changed my mind. I want data object `GAIN` to be `#define`.	I changed the **Storage class** selection to `Define`.	`GAIN` is not declared in the `.c` file as a `MemConst` parameter. Rather, it is defined as a `#define` macro by the code `#define GAIN 5.0`, and this is located near the top of the `.c` file with the other preprocessor directives.
Example 3: I changed my mind again after doing Example 2. I do want `GAIN` defined using the `#define` preprocessor directive. But I do not want to include the `#define` in this file. I know it exists in another file and I want to reference that file.	On the Model Explorer, I notice that the property value for the **Header file** property is blank. I changed this to `filename.h`. (I chose the ANSI C/C++ double quote mechanism for the `#include`, but could have chosen the angle bracket mechanism.) Also, I must make the user-defined `filename.h` available to the compiler, placing it either in the system path or local directory.	`#define GAIN 5.0` is not present in this `.c` file. Instead, the `#include filename.h` code appears as a preprocessor directive at the top of the file.
Example 4: I have one more change I want to make. Let us say that we have declared the data object `data_in`, and that its declaration statement in the `.c` file reads `real_T data_in = 0.0;`. I want to replace this statement with an alias in the `.c` file.	In the Model Explorer, I selected the data object `data_in`. I noticed that the **Identifier** field is blank. I changed this to `data_in_alias`, which I know is a valid ANSI C/C++ variable name.	The identifier `data_in_alias` now appears in the `.c` file everywhere `data_in` appeared.

Specify Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global

during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Use the **Persistence Level** property of `mpt.Signal` and `mpt.Parameter` data objects. For descriptions of the properties on the Model Explorer, see “MPT Data Object Properties” on page 39-2.

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



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

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data

without the custom attributes you specified for that data object. Or, based on expression folding, the code generator could remove the data object so that it does not appear in the code.

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

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object. If the data object's **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears tunable in the generated code with the custom attributes that you specified. For example, this would occur if **Persistence level** is 2 and **Parameter tune level** is 5.

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

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

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

Storage Classes in Embedded Coder

- “Reuse Parameter Data from External Code in the Generated Code” on page 40-2
- “Import Parameter Data with Conditionally Compiled Dimension Length” on page 40-6
- “Access Structured Data Through a Pointer That External Code Defines” on page 40-10
- “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14
- “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23
- “Access Data Through Functions with Storage Class GetSet” on page 40-25
- “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary” on page 40-41
- “Integrate External Application Code with Code Generated from PID Controller” on page 40-45
- “Configure Generated Code According to Interface Control Document Specifications” on page 40-54
- “Generate Local Variables with Localizable Storage Class” on page 40-62

Reuse Parameter Data from External Code in the Generated Code

This example shows how to generate code that imports a parameter value from your external code.

Create External Code Files

Suppose your external code defines a vector parameter `myGains` with three elements. Save the definition in your current folder in a file `ex_vector_import_src.c`.

```
#include "ex_vector_import_decs.h"

my_int8 myGains[3] = {
    2,
    4,
    6
};
```

Save the declaration in your current folder in a file called `ex_vector_import_decs.h`.

```
#include "ex_vector_import_cust_types.h"

extern my_int8 myGains[3];
```

Save the data type definition `my_int8` in your current folder in a file called `ex_vector_import_cust_types.h`.

```
typedef signed char my_int8;
```

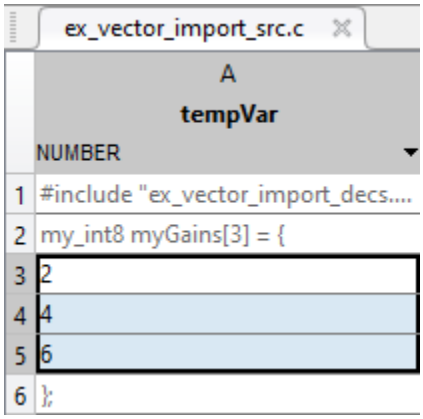
Import Parameter Value for Simulation

In your current folder, right-click the file `ex_vector_import_src.c` and select **Import Data**.

In the Import dialog box, set **Output Type** to `Numeric Matrix`.

Set the name of the generated MATLAB® variable to `tempVar`.

Select only the parameter values (2, 4, and 6) to import.



Import the data by clicking the green check mark. The MATLAB variable `tempVar` appears in the base workspace.

Alternatively, use the command prompt to manually create `tempVar`.

```
tempVar = [2;4;6];
```

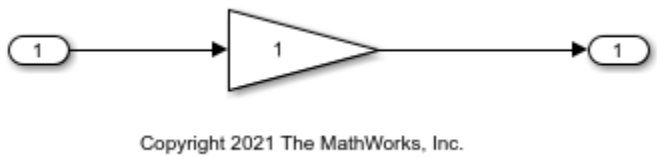
At the command prompt, create a `Simulink.Parameter` object that represents `myGains`.

```
myGains = Simulink.Parameter(tempVar);
```

Create and Configure Model

Create the model `ex_vector_import`.

```
open_system('ex_vector_import')
```



Open the **Embedded Coder** app.

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, inspect the **Parameters** tab.

Use the **Value** column to set the value of the **Gain** parameter in the Gain block to `myGains`.

Click the **Show/refresh additional information** button. The data table now contains a row that represents the parameter object, `myGains`.

Use the **Data Type** column to set the data type of `myGains` to `my_int8`.

For the other row (which represents the **Gain** parameter of the Gain block), set **Data Type** to `Inherit: Inherit from 'Gain'`. With this setting, the **Gain** parameter inherits the data type `my_int8` from `myGains`.

Set the **Change view** drop-down list to `Code`.

Set these properties for `myGains`:

- **Storage Class** to `ImportFromFile`
- **Header File** to `ex_vector_import_decs.h`

Alternatively, use these commands at the command prompt to configure the object and the block:

```
set_param('ex_vector_import/Gain','Gain','myGains',...
    'ParamDataTypeStr','Inherit: Inherit from ''Gain''')
myGains.DataType = 'my_int8';
myGains.CoderInfo.StorageClass = 'Custom';
myGains.CoderInfo.CustomStorageClass = 'ImportFromFile';
myGains.CoderInfo.CustomAttributes.HeaderFile = 'ex_vector_import_decs.h';
```

At the command prompt, create a `Simulink.AliasType` object to represent your custom data type `my_int8`. Set the `DataScope` and `HeaderFile` properties to import the type definition from your external code.

```
my_int8 = Simulink.AliasType('int8');
my_int8.DataScope = 'Imported';
my_int8.HeaderFile = 'ex_vector_import_cust_types.h';
```

Set model configuration parameter **Source files** to `ex_vector_import_src.c`.

```
set_param('ex_vector_import','CustomSource','ex_vector_import_src.c')
```

Generate and Inspect Code

Generate code.

```
slbuild('ex_vector_import')
```

```
### Starting build procedure for: ex_vector_import
### Successful completion of build procedure for: ex_vector_import

Build Summary

Top model targets built:

Model              Action                         Rebuild Reason
========================================================================================
ex_vector_import  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.384s
```

The generated file `ex_vector_import.h` includes the external header files `ex_vector_import_decs.h` and `ex_vector_import_cust_types.h`, which contain the parameter variable declaration (`myGains`) and custom type definition (`my_int8`).

```
file = fullfile('ex_vector_import_ert_rtw','ex_vector_import.h');
rtwdemodbtype(file,'#include "ex_vector_import_cust_types.h"',...
    '#include "ex_vector_import_cust_types.h"',1,1)
rtwdemodbtype(file,'/* Includes for objects with custom storage classes */',...
    '#include "ex_vector_import_decs.h"',1,1)
```

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

The generated code algorithm in the model `step` function in the generated file `ex_vector_import.c` uses `myGains` for calculations.

```
file = fullfile('ex_vector_import_ert_rtw','ex_vector_import.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)
```

```
/* Model step function */
void ex_vector_import_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
```

```
   *  Inport: '<Root>/In1'
   */
  rtY.Out1[0] = (real_T)myGains[0] * rtU.In1;
  rtY.Out1[1] = (real_T)myGains[1] * rtU.In1;
  rtY.Out1[2] = (real_T)myGains[2] * rtU.In1;
}
```

The generated code does not define (allocate memory for) or initialize the global variable `myGains` because the data scope of the corresponding parameter object is imported.

When you simulate the model in Simulink®, the model uses the value stored in the `Value` property of the parameter object. However, if you use external mode simulation, the external executable begins the simulation by using the value from your code. See “Considerations for Other Modeling Goals” on page 58-67.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Import Parameter Data with Conditionally Compiled Dimension Length” on page 40-6

Import Parameter Data with Conditionally Compiled Dimension Length

Suppose your external code conditionally allocates memory for and initializes lookup table and breakpoint set data based on a dimension length that you specify as a `#define` macro. This example shows how to generate code that imports this external global data.

Create External Code Files

Save the definition of the breakpoint set data `T1Break` and lookup table data `T1Data` in your current folder in a file called `ex_vec_symdim_src.c`. These global variables have either 9 or 11 elements depending on the value of the macro `bpLen`.

```
#include "ex_vec_symdim_decs.h"

#if bpLen == 11
double T1Break[bpLen] = {
    -5.0,
    -4.0,
    -3.0,
    -2.0,
    -1.0,
    0.0,
    1.0,
    2.0,
    3.0,
    4.0,
    5.0
} ;

double T1Data[bpLen] = {
    -1.0,
    -0.99,
    -0.98,
    -0.96,
    -0.76,
    0.0,
    0.76,
    0.96,
    0.98,
    0.99,
    1.0
} ;
#endif

#if bpLen == 9
double T1Break[bpLen] = {
    -4.0,
    -3.0,
    -2.0,
    -1.0,
    0.0,
    1.0,
    2.0,
```

```
    3.0,
    4.0
} ;

double T1Data[bpLen] = {
    -0.99,
    -0.98,
    -0.96,
    -0.76,
    0.0,
    0.76,
    0.96,
    0.98,
    0.99
} ;
#endif
```

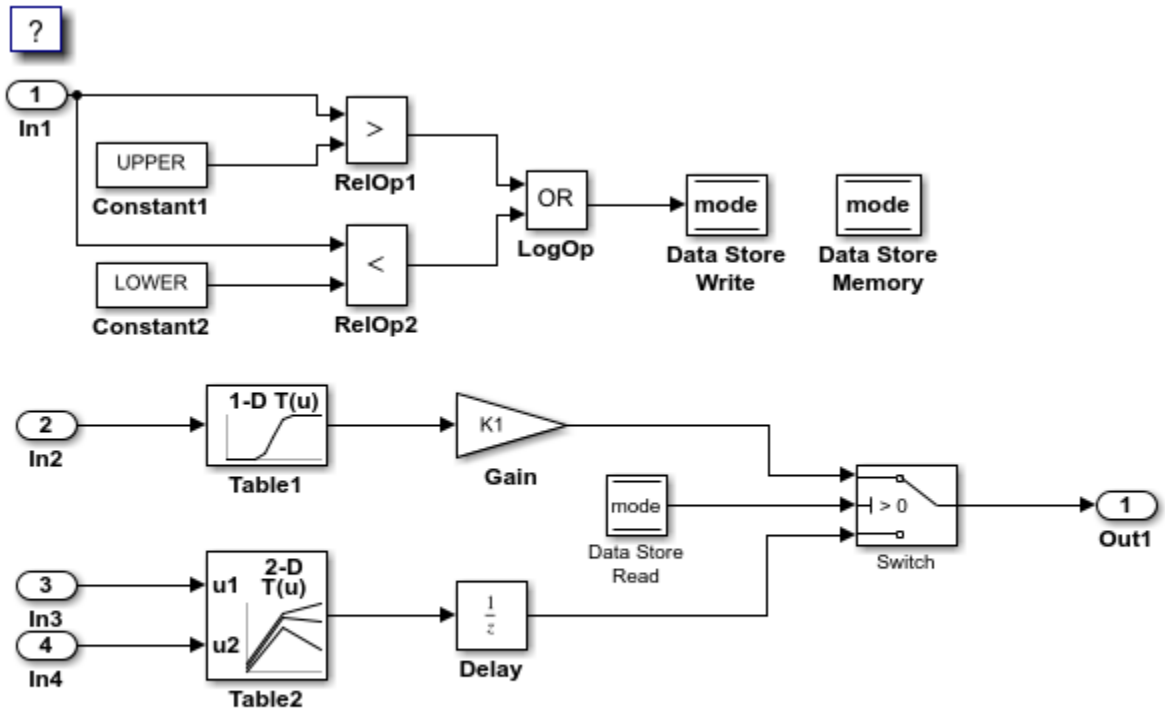
Save the declarations of the variables and the definition of the macro in your current folder in file `ex_vec_symdim_decs.h`.

```
#define bpLen 11

extern double T1Break[bpLen];
extern double T1Data[bpLen];
```

Explore and Configure Example Model

Open the example model `rtwdemo_configinterface`.

```
open_system('rtwdemo_configinterface')
set_param('rtwdemo_configinterface/Table1','DataSpecification','Table and breakpoints');
```



Copyright 1994-2020 The MathWorks, Inc.

Open the Table1 block dialog box. The block refers to variables, `T1Data` and `T1Break`, in the model workspace. These variables store the lookup table and breakpoint set data with 11 elements.

At the command prompt, convert the variables to `Simulink.Parameter` objects.

```
mws = get_param('rtwdemo_configinterface','modelworkspace');
T1Data = mws.getVariable('T1Data');
T1Break = mws.getVariable('T1Break');
T1Data = Simulink.Parameter(T1Data);
T1Break = Simulink.Parameter(T1Break);
```

At the command prompt, create a `Simulink.Parameter` object to represent the custom macro `bpLen`.

```
bpLen = Simulink.Parameter(11);
bpLen.Min = 9;
bpLen.Max = 11;
bpLen.DataType = 'int32';
bpLen.CoderInfo.StorageClass = 'Custom';
bpLen.CoderInfo.CustomStorageClass = 'ImportedDefine';
bpLen.CoderInfo.CustomAttributes.HeaderFile = 'ex_vec_symdim_decs.h';
```

Use `bpLen` to set the dimensions of the lookup table and breakpoint set data. Configure the model to enable symbolic dimensions by selecting the configuration parameter **Allow symbolic dimension specification**.

```
T1Data.Dimensions = '[1 bpLen]';
T1Break.Dimensions = '[1 bpLen]';
set_param('rtwdemo_configinterface','AllowSymbolicDim','on');
```

Set **Configuration Parameters > Code Generation > Custom Code > Code information > Source files** to `ex_vec_symdim_src.c`.

```
set_param('rtwdemo_configinterface','CustomSource','ex_vec_symdim_src.c')
```

Configure the objects to import the data definitions from your external code.

```
mws.assignin('T1Data',T1Data);
mws.assignin('T1Break',T1Break);
cm = coder.mapping.utils.create('rtwdemo_configinterface');
setDataDefault(cm,'ModelParameters','StorageClass','Default');
setModelParameter(cm,'Table1','StorageClass','Model default');
setModelParameter(cm,'Table2','StorageClass','Model default');
setModelParameter(cm,'T1Data','StorageClass','ImportFromFile','HeaderFile','ex_vec_symdim_decs.h
setModelParameter(cm,'T1Break','StorageClass','ImportFromFile','HeaderFile','ex_vec_symdim_decs.h
```

Generate and Inspect Code

Generate code from the model.

The generated code algorithm is in the model `step` function in the generated file `rtwdemo_configinterface.c`. The algorithm passes `T1Break`, `T1Data`, and `bpLen` as argument values to the function that performs the table lookup. In this case, `bpLen` controls the upper bound of the binary search that the function uses.

For more information about symbolic dimensions, see “Implement Symbolic Dimensions for Array Sizes in Generated Code” on page 27-2.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Reuse Parameter Data from External Code in the Generated Code” on page 40-2

Access Structured Data Through a Pointer That External Code Defines

This example shows how to generate code that uses global data that some handwritten code defines. In the handwritten code, a pointer variable points to one of three structure variables that contain parameter data. A handwritten function switches the pointer between the structures. The generated code accesses the parameter data by dereferencing the pointer variable.

Explore External Code

Open the example source file `rtwdemo_importstruct_user.c`. The code defines a default data structure variable `ReferenceStruct` as constant (`const`) data and statically initializes each field. This structure represents the reference data set.

```
/* Constant default data structure (reference data set) */
const DataStruct_type ReferenceStruct =
{
  11,   /* OFFSET */
  2     /* GAIN */
};
```

The code defines two other structure variables, `WorkingStruct1` and `WorkingStruct2`, as volatile (`volatile`) data.

```
/* Volatile data structures (working data sets) */
volatile DataStruct_type WorkingStruct1;
volatile DataStruct_type WorkingStruct2;
```

The code defines a function that can initialize:

- An inactive working structure from the active working structure.
- Both working structures from the reference structure.

```
/* Function to initialize inactive working structures from active structure */
void InitInactiveWorkingStructs(void)
{
    if (StructPointer == &WorkingStruct1) {
        /* Copy values from WorkingStruct1 to WorkingStruct2 */
        memcpy((void*)&WorkingStruct2, (void*)&WorkingStruct1, sizeof(ReferenceStruct));

    } else if (StructPointer == &WorkingStruct2) {
        /* Copy values from WorkingStruct2 to WorkingStruct1 */
        memcpy((void*)&WorkingStruct1, (void*)&WorkingStruct2, sizeof(ReferenceStruct));

    } else {
        /* Initialize both working structures from ReferenceStruct */
        memcpy((void*)&WorkingStruct1, &ReferenceStruct, sizeof(ReferenceStruct));
        memcpy((void*)&WorkingStruct2, &ReferenceStruct, sizeof(ReferenceStruct));
    }
}
```

The code defines `StructPointer`, which is a `const volatile` pointer to a structure. The code initializes the pointer to the address of `ReferenceStruct`.

```
/* Define structure pointer.  Point to reference structure by default */
const volatile DataStruct_type *StructPointer = &ReferenceStruct;
```

Finally, the code defines a function that can dynamically set `StructPointer` to point to either `ReferenceStruct`, `WorkingStruct1`, or `WorkingStruct2`.

```
/* Function to switch between structures */
void SwitchStructPointer(Dataset_T Dataset)
{
    switch (Dataset) {
    case Working1:
        StructPointer = &WorkingStruct1;
        break;
    case Working2:
        StructPointer = &WorkingStruct2;
        break;
    default:
        StructPointer = &ReferenceStruct;
    }
}
```

The example header file `rtwdemo_importstruct_user.h` defines the enumeration `Dataset_T` and the structure type `Datastruct_type`. The file includes (`#include`) the built-in Simulink® Coder™ header file `rtwtypes.h`, which defines (`typedef`) Simulink Coder data types such as `int16_T`.

```
#include "rtwtypes.h"

typedef enum {
    Reference = 0,
    Working1,
    Working2
} Dataset_T;

typedef struct DataStruct_tag {
  int16_T   OFFSET; /* OFFSET */
  int16_T   GAIN;   /* GAIN */
} DataStruct_type;
```

The file also declares the global variables and the functions.

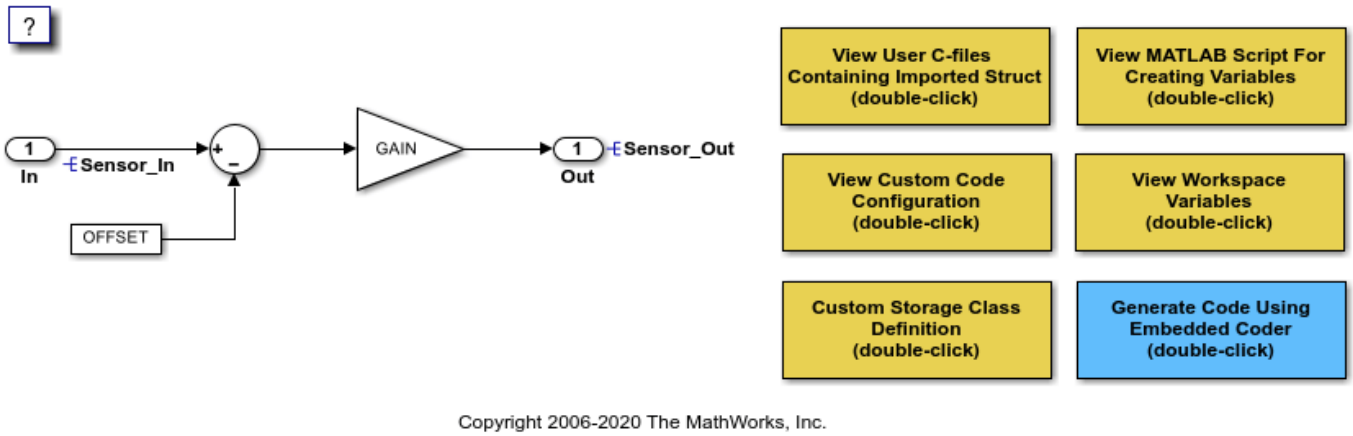
Purpose of External Code

The code is designed so that the source code of a control algorithm (whether generated or handwritten) can read data from `ReferenceStruct`, `WorkingStruct1`, or `WorkingStruct2` by dereferencing (`->`) `StructPointer`. The two working structures (`WorkingStruct1` and `WorkingStruct2`) are located in volatile memory and are intended to be changed during runtime by an external calibration tool. The calibration engineer does not change the active working structure. Instead, the engineer changes the parameter values in the inactive working structure and then activates it by switching working structures.

If necessary for safety or in preparation for shutting down the application, the calibration tool can point `StructPointer` to `ReferenceStruct` instead. `ReferenceStruct` stores default parameter values that do not change during execution.

Explore Example Model

Open the example model, `rtwdemo_importstruct`.



The model creates variables and objects in the base workspace. The Constant block and the Gain block use the `ECoderDemos.Parameter` objects `GAIN` and `OFFSET` to set the **Constant value** and **Gain** block parameters. `ECoderDemos` is an example custom package that defines two classes, `Parameter` and `Signal`, and some custom storage classes.

On the **Modeling** tab, click **Model Data Editor**.

In the Model Data Editor, inspect the **Parameters** tab.

Set the **Change view** drop-down list to `Code`.

Click the **Show/refresh additional information** button.

The Model Data Editor shows rows that correspond to the **Constant value** and **Gain** parameters of the blocks and rows that corresponds to `OFFSET` and `GAIN`, which set the parameter values. In the **Storage Class** column, `OFFSET` and `GAIN` use the custom storage class `StructPointer`, which the `ECoderDemos` package defines.

Open the Custom Storage Class Designer and inspect the custom storage classes in the `ECoderDemos` package. At the command prompt, use this command:

```
cscdesigner('ECoderDemos')
```

This example package defines multiple custom storage classes, including `StructPointer`. You cannot edit the definitions. However, you can create your own packages and custom storage classes later. For an example that shows how to create a package and a custom storage class, see “Create and Apply a Storage Class” on page 40-14.

Under **Custom storage class definitions**, click `StructPointer`. The settings for this custom storage class enable the generated code to interact with the pointer variable, `StructPointer`, from the external code. For example, the custom storage class uses these settings:

- **Data scope** is set to `Imported` because the example external code defines (allocates memory for) `StructPointer`. With this setting, the code generator avoids generating unnecessary, duplicate definitions for the data items, such as the `ECoderDemos.Parameter` objects, that use the custom storage class.
- **Data access** is set to `Pointer` because in the example external code, `StructPointer` is a pointer.
- **Memory section** is set to `ConstVolatile` because the example external code defines `StructPointer` as constant, volatile data (`const volatile`).
- **Type** is set to `FlatStructure` because in the example external code, `StructPointer` points to a structure. With this setting, the generated code treats each data item (`ECoderDemos.Parameter` object) as a field of a flat structure whose variable name and type name you can specify.
- On the **Structure Attributes** tab, **Struct name** is set to `StructPointer`. For a `FlatStructure` custom storage class, **Struct name** specifies the name of the structure variable in the generated code. In this example, `StructPointer` is the name of the variable that the external code defines.
- **Type name** is set to `DataStruct_type`, which is the name of the structure type that the example external code defines.

In the model, in the **Configuration Parameters** dialog box, inspect the **Code Generation > Custom Code** pane.

On the **Additional source code** tab, select **Initialize code**. This parameter value specifies the code to include in the generated model initialize function. In this model, the configuration parameter is set so that the generated code calls the `InitInactiveWorkingStructs` function. `InitInactiveWorkingStructs` initializes the working structures with the values from `ReferenceStruct`. The initialization code then sets the pointer to the first working structure.

On the **Code information** tab, select **Source files**. This configuration parameter identifies the example external code file `rtwdemo_importstruct_user.c` for inclusion in the build process after code generation.

Generate and Inspect Code

Generate code from the model.

In the generated file `rtwdemo_importstruct.c`, the model initialization function calls `InitInactiveWorkingStructs`. The algorithm in the model execution (`step`) function dereferences the pointer variable `StructPointer`.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 36-161
- “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14

Create Storage Classes by Using the Custom Storage Class Designer

To control the appearance of data in the generated code, you can use built-in storage classes such as `ExportToFile` (see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67). If the built-in storage classes do not meet your requirements, you can create your own storage classes. To create a storage class that you can apply to data objects, create the storage class in a package by using the Custom Storage Class Designer. See “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

To define a storage class that you can apply in a model by using the Code Mappings editor, use the **Embedded Coder Dictionary**. Or, create the storage class in a package and refer to the package from the Embedded Coder Dictionary.

A package also defines `Signal` and `Parameter` data classes. See “Data Objects”. A data object from a package such as `Simulink` can use storage classes defined in that package but not storage classes that are defined in other packages.

For information about using the Custom Storage Class Designer to create memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Create and Apply a Storage Class

This example shows how to control code generated from a model by creating and applying your own storage class for data objects.

Explore Example Model

Open the model rtwdemo_cscpredef. You can control code generated from this model by defining your own data classes and creating your own storage classes.

In this example, you export the declarations and definitions of multiple signals and parameters in the model to one declaration header file and one definition file.

Create Data Class Package

To create storage classes, you first create a data class package to contain the storage class definitions. Data objects that you create from your package can use the storage classes that the package defines.

1. Create your own data class package by copying the example package folder `+SimulinkDemos`. Navigate to the example package folder.

   ```
   % Remember the current folder path
   currentPath = pwd;

   % Navigate to the example package folder
   demoPath = '\toolbox\simulink\simdemos\dataclasses';
   cd([matlabroot,demoPath])
   ```

2. Copy the `+SimulinkDemos` folder to your clipboard.
3. Return to your working folder.

```
cd(currentPath)
```

4. Paste the `+SimulinkDemos` folder from your clipboard into your working folder. Rename the copied folder to `+myPackage`.
5. Navigate inside the `+myPackage` folder to the file `Signal.m` to edit the definition of the `Signal` class.
6. Uncomment the `methods` section that defines the method `setupCoderInfo`. In the call to the function `useLocalCustomStorageClasses`, replace `'packageName'` with `'myPackage'`. When you finish, the section appears as follows:

   ```
   methods
     function setupCoderInfo(h)
       % Use custom storage classes from this package
       useLocalCustomStorageClasses(h, 'myPackage');
     end
   end % methods
   ```

   The function `useLocalCustomStorageClasses` allows you to apply the storage classes that `myPackage` defines to data objects that you create from `myPackage`.
7. Save and close the file.
8. Navigate inside the `+myPackage` folder to the file `Parameter.m` to edit the definition of the `Parameter` class. Uncomment the `methods` section that defines the method `setupCoderInfo` and replace `'packageName'` with `'myPackage'`.
9. Save and close the file.

For more information on creating a data class, see “Define Data Classes”.

Create Storage Class

You can use the Custom Storage Class Designer to create or to edit the storage classes that a data class package defines.

1. Set your current folder to the folder that contains the package `myPackage`.
2. Open the Custom Storage Class Designer.

   ```
   cscdesigner('myPackage')
   ```
3. Select the storage class `ExportToFile`.
4. In the **Name** field, rename the storage class to `ExportToGlobal`.
5. In the **Header file** drop-down list, change the selection from `Instance specific` to `Specify`. In the new field, provide the header file name `global.h`.
6. In the **Definition file** drop-down list, change the selection from `Instance specific` to `Specify`. In the new field, provide the definition file name `global.c`.
7. Click **OK**. Click **Yes** to save changes to the data class package `myPackage`.

Apply Storage Class

To apply your own storage class, you create data objects from your package and configure the objects to use your storage class.

1. Create data objects to represent some of the parameters and signals in the example model. Create the objects using the data class package `myPackage`.

```
% Parameters
templimit = myPackage.Parameter(202);
pressurelimit = myPackage.Parameter(45.2);
O2limit = myPackage.Parameter(0.96);
rpmlimit = myPackage.Parameter(7400);

% Signals
tempalarm = myPackage.Signal;
pressurealarm = myPackage.Signal;
O2alarm = myPackage.Signal;
rpmalarm = myPackage.Signal;
```

2 Set the storage class of each object to `ExportToGlobal`.

```
% Parameters
templimit.CoderInfo.StorageClass = 'Custom';
templimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
pressurelimit.CoderInfo.StorageClass = 'Custom';
pressurelimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
O2limit.CoderInfo.StorageClass = 'Custom';
O2limit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
rpmlimit.CoderInfo.StorageClass = 'Custom';
rpmlimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';

% Signals
tempalarm.CoderInfo.StorageClass = 'Custom';
tempalarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
pressurealarm.CoderInfo.StorageClass = 'Custom';
pressurealarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
O2alarm.CoderInfo.StorageClass = 'Custom';
O2alarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
rpmalarm.CoderInfo.StorageClass = 'Custom';
rpmalarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
```

3 Select the **Signal name must resolve to Simulink signal object** option for each of the target signals in the model. You can select the option by using the Signal Properties dialog box, the **Resolve** column in the Model Data Editor, or by using the command line.

```
% Signal tempalarm
portHandles = get_param('rtwdemo_cscpredef/RelOp1','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal pressurealarm
portHandles = get_param('rtwdemo_cscpredef/RelOp2','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal O2alarm
portHandles = get_param('rtwdemo_cscpredef/RelOp3','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal rpmalarm
portHandles = get_param('rtwdemo_cscpredef/RelOp4','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')
```

Generate Code

1. Generate code for the example model.

```
slbuild('rtwdemo_cscpredef')
```

2. In the Code view for the model, view the generated header file `global.h`. The file contains the `extern` declarations of the model signals and parameters that use the storage class `ExportToGlobal`.

```
/* Declaration for custom storage class: ExportToGlobal */
extern boolean_T O2alarm;
extern real_T O2limit;
extern boolean_T pressurealarm;
extern real_T pressurelimit;
extern boolean_T rpmalarm;
extern real_T rpmlimit;
extern boolean_T tempalarm;
extern real_T templimit;
```

3. View the generated file `global.c`. The file contains the definitions of the model signals and parameters that use the storage class `ExportToGlobal`.

```
/* Definition for custom storage class: ExportToGlobal */
boolean_T O2alarm;
real_T O2limit = 0.96;
boolean_T pressurealarm;
real_T pressurelimit = 45.2;
boolean_T rpmalarm;
real_T rpmlimit = 7400.0;
boolean_T tempalarm;
real_T templimit = 202.0;
```

Modify a Built-In Storage Class

You cannot directly modify a built-in storage class, such as `ExportToFile` from the `Simulink` package, but you can create a copy, and then modify the copy. When you create a new package with no `csc_registration.m` file, and then open the Custom Storage Class Designer for the first time in that package, Simulink copies the definitions of the built-in storage classes into the package. Then, in the Custom Storage Class Designer, you can modify the copied definitions. Alternatively, to keep the built-in storage classes available in your package, copy one of them by clicking **Copy**, then modify the resulting copy.

Control Data Representation by Configuring Storage Class Properties

The Custom Storage Class Designer is a tool for creating and managing storage classes and memory sections. To open the Custom Storage Class Designer for a specific package, for example, `+mypkg`, at the command prompt, use the `cscdesigner` function:

```
cscdesigner('mypkg')
```

The name of the folder that represents a package begins with a `+`. When you use the function `cscdesigner`, omit the `+` from the name of the target package.

To control the effect that a storage class has on data in a model, you specify properties of the storage class in the Custom Storage Class Designer. To determine how to generate a particular kind of code, use the information in the table.

Kind of Data in Generated Code	Technique
Structure	Set **Type** to `FlatStructure`. See “Generate Structured Data” on page 40-19.
Macro	Set **Data initialization** to `Macro` and clear **For signals**. See “Generate a Macro” on page 40-18.
Pointer	Set **Data access** to `Pointer` and **Data scope** to `Auto`, `Imported`, or `Instance specific`. Your external code must define the pointer variable.
`static` data (file-scoped data)	Set **Data scope** to `File`.
`const` or `volatile` data	Set **Memory section** to a memory section that specifies `const`, `volatile`, or both. For example, use one of the built-in memory sections `MemConst`, `MemVolatile`, and `MemConstVolatile`. To create your own memory section, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.
Calls to external functions to read and write to data	See “Call Custom Accessor Functions or Macros Instead of Reading and Writing to Variables” on page 40-19.

Allow Users of Storage Class to Specify Property Value

For some properties of a storage class, such as **Data scope**, instead of specifying a value in the Custom Storage Class Designer, you can allow users of the storage class to specify the value. You can create a single, flexible storage class instead of multiple storage classes that vary only by a few property values.

For example, suppose you create a storage class that yields a global variable in the generated code. You can configure the storage class so that a user can set the **Data scope** property to `Exported` for a signal in a model and set the property to `Imported` for a different signal.

In the Custom Storage Class Designer, for each property that you want to configure in this way, set the property value to `Instance specific`. Then, when a user applies the storage class to a data item, the property appears to the user as a custom attribute for that data item. For information about instance-specific custom attributes, see “Storage Class Properties” on page 36-75.

Generate a Macro

To create a storage class that yields a macro in the generate code, set **Data initialization** to `Macro` and clear **For signals**.

When you set **Data initialization** to `Macro`, **Definition file** has no meaning. To prevent users of the storage class from specifying a meaningless definition file, in the Custom Storage Class Designer, set **Definition file** to `Specify` and leave the text box empty.

- To define the macro in a header file (`#define`), specify the name of the header file with the **Header file** property.
- To provide the macro definition as a compiler option or compiler flag, set **Data access** to `Imported` and **Header file** to `Specify`. Leave the **Header file** text box empty. Then, the generated code does not define the macro and does not include (`#include`) a header file.

To specify the compiler option or flag, use the model configuration parameter **Configuration Parameters** > **Code Generation** > **Custom Code** > **Code information** > **Defines**. See Code Generation Pane: Custom Code: Additional Build Information: Defines.

Generate Structured Data

To create a storage class that aggregates data items into a flat structure (similar to the built-in storage class `Struct`), set **Type** to `FlatStructure`. The **Structure Attributes** tab appears.

- To control the name of the global structure variable that appears in the generated code, use the **Struct name** property and text box.
- If you set **Struct name** to `Instance specific`, users of the storage class specify the name of the structure variable. In this case, the code generator derives the name of the structure type from the name of the variable.

  To more precisely control the name and other characteristics of the structure type, set **Struct name** to `Specify`. Control the type characteristics by using the additional properties that appear.
- To generate a structure with bit fields (similar to the built-in storage class `BitField`), select **Bit-pack booleans**. Data items that use the data type `boolean` appear in the generated code as bit fields of the structure.

Control Assignment of Initial Value in Generated Code

- To use the default Simulink Coder initialization strategy, set **Data initialization** to the default value, `Auto`.
  - The generated code statically initializes parameter data with an assignment statement at the top of a `.c` or `.cpp` source file, outside of any function.
  - The generated code dynamically initializes signal and state data as part of the initialization function of the model (named *`model`*`_initialize` by default).
- To statically initialize the data, set **Data initialization** to `Static`.
- To dynamically initialize the data, set **Data initialization** to `Dynamic`.
- To prevent the generated code from initializing the data, set **Data initialization** to `None`. When your external code initializes the data, use this setting to prevent the generated code from overwriting your initialization. This setting does not apply to data that you specify in an Initialize Function block.

Call Custom Accessor Functions or Macros Instead of Reading and Writing to Variables

Using the built-in storage class `GetSet`, you can generate code that calls your external, custom functions instead of directly interacting with variables. For general information about `GetSet`, see “Access Data Through Functions with Storage Class GetSet” on page 40-25.

To create a similar storage class, set **Type** to `AccessFunction`.

- Specify the names of the external functions.
  - To apply the same `get` or `set` function naming scheme to data items that use the storage class, set **Get function** and **Set function** to `Specify`. Then, in the text boxes, specify the function naming scheme, for example, `get_myData_$N`. Use the token `$N` in a naming scheme to represent the name of each data item. If you do not use the token, each data item uses the same specified `get` or `set` function name, so the model generates an error when you generate code.

  - To specify a different `get` or `set` function name for each data item, set **Get function** or **Set function** to `Instance specific`. Later, when you create a data item and apply the storage class, specify the function name by configuring the custom attributes of the data item.
- Specify the name of the header file that declares the `get` and `set` functions with the **Header file** property. In this case, **Definition file** has no meaning. To prevent users of the storage class from specifying a meaningless definition file, set **Definition file** to `Specify` and leave the text box empty.
- If you implement the `get` mechanism for scalar or array data as a macro instead of a function, you can generate code that omits parentheses when reading that data. On the **Access Function Attributes** tab, select **Get data through macro (omit parentheses)**.

Generate Code That Uses Data From External Code

When your external code defines data, to avoid generating a duplicate definition, you must configure the generated code to read and write to the existing data. In the Custom Storage Class Designer, set **Data scope** to `Imported`. The code generator does not generate a definition for the global variable or macro.

- If your code defines a pointer variable, to generate code that interacts with the pointer, set **Data access** to `Pointer`.
- If you generate structured data by setting **Type** to `FlatStructure`, your code must define the structure type and the global structure variable.
- If your code initializes state or signal data, you can prevent the generated code from overwriting your initialization by setting **Data initialization** to `None`. See “Prevent Duplicate Initialization Code for Global Variables” on page 58-70

Generate Code Comments with Data Definitions and Declarations

To customize the comments that appear in the generated code, on the **Comments** tab, set **Comment rules** to `Specify`. To specify a comment that appears with the declaration of each data item, use the **Declaration comment** box. To specify a comment that appears with the definition, use the **Definition comment** box.

For structured data (you set **Type** to `FlatStructure`), to specify a comment that appears with the structure type definition, use the **Type comment** box.

Avoid Errors During Code Generation by Validating Storage Class Configuration

As you configure a storage class in the Custom Storage Class Designer, to check for possible errors in the configuration, click **Validate**.

For example, validating the storage class warns you if you accidentally set **Data initialization** to `Macro` with **For signals** selected. A signal cannot appear in the generated code as a macro.

Make Storage Classes Available Outside the Current Folder

To use a storage class that you create, you must make the defining package available. Either set your current folder to the folder that contains the package or add the folder containing the package folder to the MATLAB path (see “What Is the MATLAB Search Path?”). Adding the package to the MATLAB path enables you to use the storage classes no matter where you set your current folder.

Make Storage Classes Available in the Code Mappings Editor

To use package-based storage classes in the Code Mappings editor, refer to the package from an Embedded Coder Dictionary. To configure an Embedded Coder Dictionary to refer to the package, see “Refer to Code Generation Definitions in a Package”.

Share Storage Class Between Packages

When you create a package, you can refer to a storage class defined in another package such as the built-in package `Simulink` or a different package that you created. Then, the defining package and any referencing packages can use the storage class. When you want to make a change to the storage class, you make the change only once, in the defining package.

To configure a package to refer to a storage class that another package defines:

1. In the Custom Storage Class Designer, select the **Custom Storage Class** tab.
2. Use **Select Package** to select the referencing package. The package must be writable.
3. In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference.
4. Click **New Reference**.

   A new reference with a default name and properties appears below the previously selected definition. The new reference is selected and a **Reference** tab appears that shows the initial properties.
5. Use **Name** to specify a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package. The name cannot be a TLC keyword.
6. Set **Refer to custom storage class in package** to specify the package that contains the storage class that you want to reference.
7. Set **Custom storage class to reference** to specify the storage class to be referenced.
8. Click **OK** or **Apply**. To save the changes permanently, click **Save**.

Control Appearance of Storage Class Drop-Down List

When you apply a storage class to a data item, you select the storage class from a drop-down list.

- To control the order of the custom storage classes in the list, in the Custom Storage Class Designer, use the **Up** and **Down** buttons. The order of the storage classes in drop-down lists matches the order that you specify in the Custom Storage Class Designer.
- To prevent a user of a storage class from applying the storage class to parameter data or to signal and state data, use the **For parameters** and **For signals** check boxes. When you clear one of these properties, for the corresponding kind of data, the storage class does not appear in the drop-down list.

  For example, if your storage class yields a macro in the generated code because you set **Data initialization** to `Macro`, to prevent users from applying the storage class to signal data, clear **For signals**.

Protect Custom Storage Class Definitions

When you click **Save** in the Custom Storage Class Designer, the Designer saves memory section and storage class definitions into the `csc_registration.m` file in the package folder. To determine the location of this file, in the Custom Storage Class Designer, inspect the value of **Filename**.

You can prevent changes to the storage class definitions of an entire package by converting the `csc_registration.m` file from a MATLAB file to a P-file. Use the `pcode` function.

A best practice is to keep `csc_registration.m` and `csc_registration.p` in your package folder. That way, if you modify the storage classes by using the Designer, you can delete `csc_registration.p` and later regenerate it after you finish the modifications. Because the P-coded version of the file takes precedence, when both files exist in the package, the storage classes are protected and you cannot modify them in the Custom Storage Class Designer.

Further Customize Generated Code by Writing TLC Code

If creating your own storage class by manipulating the properties in the Custom Storage Class Designer does not meet your requirements, you can finely control the generated code by writing TLC code for your storage class. Use an advanced mode of the Custom Storage Class Designer and, for the storage class, set **Type** to `Other`. See “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23.

See Also

Related Examples

- “Create and Apply a Storage Class” on page 40-14
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Data Objects”
- “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23
- “Access Structured Data Through a Pointer That External Code Defines” on page 40-10

Finely Control Data Representation by Writing TLC Code for a Storage Class

If creating your own storage class by using the properties in an Embedded Coder Dictionary or the Custom Storage Class Designer does not meet your requirements for controlling data representation in the generated code, you can write TLC code that explicitly controls the effect that a storage class has on the code. For example, to create a storage class that yields arbitrarily nested structures, you must write TLC code. In the Custom Storage Class Designer, these advanced storage classes have **Type** set to `Other`. You cannot create such a storage class in an Embedded Coder Dictionary, so to use the storage class in the Code Mappings editor, you must configure the dictionary of the model to refer to the package that contains the storage class, as described in “Refer to Code Generation Definitions in a Package”.

For an example, see “Generate Code That Dereferences Data from a Literal Memory Address” on page 71-15. For general information about TLC code, see “Why Use the Target Language Compiler?”.

Create Custom Attributes Class for Storage Class

As described in “Allow Users of Storage Class to Specify Property Value” on page 40-18, instance-specific properties enable users of a storage class to control the effect that the storage class has on each data item. For example, the built-in storage class `ExportToFile` has several instance-specific properties such as **Header file** and **Definition file**.

When you create a storage class with **Type** set to `Other`, to add your own instance-specific properties that are not built into the Custom Storage Class Designer, create a custom attributes class for your package. A custom attributes class is a MATLAB class that you create as a subclass of `Simulink.CustomStorageClassAttributes`. Each property that you add to your custom attributes class appears to the user of the storage class as an instance-specific property.

To create your custom attributes class, see “Define Data Classes”.

Write TLC Code for Storage Class

To control the effect of your storage class, write TLC code that specifies the code to generate for each data item.

1. In your package folder (for example, `+myPackage`), create a `tlc` folder.
2. Copy a TLC template such as `TEMPLATE_v2.tlc` from the folder *matlabroot*`/toolbox/rtw/targets/ecoder/csc_templates` (open) into your `tlc` folder.
3. Write your TLC code, following the comments in the template file. The comments describe how to specify, for example, how the generated code declares, defines, and accesses (by value or by reference) each data item.

Create Storage Class by Using Custom Storage Class Designer

To create your storage class in your package, you open the Custom Storage Class Designer in an advanced mode.

1. At the command prompt, enter:

```
cscdesigner -advanced
```

2. Select your package and create a storage class.
3. For the storage class, set **Type** to `Other`. In the **Other Attributes** pane, specify the name of your TLC file and the name of your custom attributes class.
4. Set the properties on the **Other Attributes** pane.
   - **Is grouped**: Select this option if you intend to combine multiple data items into a single entity in the generated code. For example, the built-in storage classes `BitField` and `Struct` are grouped because they can aggregate multiple data items into a single structure variable.
   - **TLC file name**: Enter the name of your TLC file. The Custom Storage Class Designer assumes that the file exists in the package `tlc` folder, so specify only the name of the file, not the file path.
   - **CSC attributes class name**: (optional) If you created a custom attributes class, enter the full name of the class, including the package name. For example, specify `myPackage.myCustomAttsClass`. For more information, see "Create Custom Attributes Class for Storage Class" on page 40-23.
5. On the **General** and **Comments** panes, specify values for the remaining properties.

To apply the storage class to data objects, you create the data objects from your package and configure the objects to use your storage class. For more information on applying a custom storage class to data objects, see "Apply Storage Class" on page 40-15. To apply the storage class in a model by using the Code Mappings editor, you must configure the Embedded Coder Dictionary that the model uses to refer to your package. When you refer to the package, the storage class appears in the Embedded Coder Dictionary and becomes available in the Code Mappings editor. For more information, see "Refer to Code Generation Definitions in a Package".

See Also

Related Examples

- "Target Language Compiler Basics"
- "Generate Code That Dereferences Data from a Literal Memory Address" on page 71-15
- "Create Storage Classes by Using the Custom Storage Class Designer" on page 40-14
- "Organize Parameter Data into a Structure by Using Struct Storage Class" on page 33-67
- "Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture" on page 32-2
- "Data Object Information in model.rtw File"

Access Data Through Functions with Storage Class GetSet

To integrate the generated code with legacy code that uses specialized functions to read from and write to data, you can use the storage class `GetSet`. Signals, block parameters, and states that use `GetSet` appear in the generated code as calls to accessor functions. You provide the function definitions.

You can also create your own storage class in Embedded Coder Dictionary to access data through functions. Creating your own storage class in Embedded Coder Dictionary gives you the flexibility of customizing function names and return types. For more information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary” on page 40-41.

To generate code that conforms to the AUTOSAR standard by accessing data through `Rte` function calls, use the AUTOSAR code perspective. See “AUTOSAR Component Configuration” (AUTOSAR Blockset).

Access Legacy Data Using Get and Set Functions

This example shows how to generate code that interfaces with legacy code by using specialized `get` and `set` functions to access data.

View the example legacy header file `ComponentDataHdr.h`. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHeader.h','/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `scalars` is a substructure that uses the structure type `ScalarData`. The structure type `ScalarData` defines three scalar fields: `inSig`, `scalarParam`, and `outSig`.

```
rtwdemodbtype('ComponentDataHeader.h','/* ScalarData */','} ScalarData;',1,1)
```

```
/* ScalarData */

typedef struct {
    double inSig;
    double scalarParam;
    double outSig;
} ScalarData;
```

View the example legacy source file `GetSetSourceData.c`. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `scalars`.

```
rtwdemodbtype('GetSetSourceData.c','/* Field "scalars" */','/* End of "scalars" */',1,1)
```

```
    /* Field "scalars" */
    {
    3.9,

    12.3,

    0.0
    },
    /* End of "scalars" */
```

The file also defines functions that read from and write to the fields of the substructure `scalars`. The functions simplify data access by dereferencing the leaf fields of the global structure variable `ex_getset_data`.

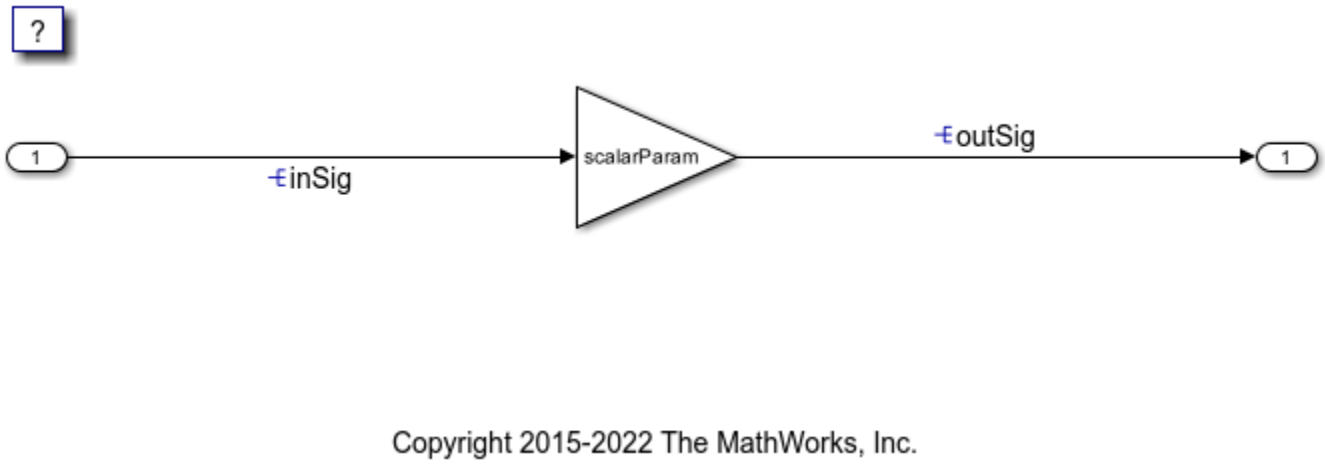
```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Scalar get() and set() functions */','/* End of scalar functions */',1,1)
```

```
/* Scalar get() and set() functions */

double get_inSig(void)
{
    return ex_getset_data.scalars.inSig;
}

void set_inSig(double value)
{
    ex_getset_data.scalars.inSig = value;
}

double get_scalarParam(void)
{
    return ex_getset_data.scalars.scalarParam;
}

void set_scalarParam(double value)
{
    ex_getset_data.scalars.scalarParam = value;
}

double get_outSig(void)
{
    return ex_getset_data.scalars.outSig;
}

void set_outSig(double value)
{
    ex_getset_data.scalars.outSig = value;
}
```

View the example legacy header file `GetSetSourceDataScalarHdr.h`. The file contains the `extern` prototypes for the `get` and `set` functions defined in `GetSetSourceData.c`.

Open the example model `rtwdemo_GetSetScalar`. The model creates the data objects `inSig`, `outSig`, and `scalarParam` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('GetSetScalar')
```



In the base workspace, double-click the object `inSig` to view its properties. The object uses the storage class `GetSet`. The `GetFunction` and `SetFunction` properties are set to the defaults, `get_$N` and `set_$N`. The generated code uses the function names that you specify in `GetFunction` and `SetFunction` to read from and write to the data. The code replaces the token `$N` with the name of the data object. For example, for the data object `inSig`, the generated code uses calls to the legacy functions `get_inSig` and `set_inSig`.

For the data object `inSig`, the `HeaderFile` property is set to `GetSetSourceDataScalarHdr.h`. This legacy header file contains the `get` and `set` function prototypes. The data objects `outSig` and `scalarParam` also use the storage class `GetSet` and the header file `GetSetSourceDataScalarHdr.h`.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, on the **Code information** tab, select **Source files**. The **Source files** box identifies the source file `GetSetSourceData.c` for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
slbuild('GetSetScalar');
```

```
### Starting build procedure for: GetSetScalar
### Successful completion of build procedure for: GetSetScalar

Build Summary

Top model targets built:

Model         Action                        Rebuild Reason
========================================================================================
GetSetScalar  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 38.183s
```

In the code generation report, view the file `rtwdemo_GetSetScalar.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm. The generated code accesses the legacy signal and parameter data by calling the custom, handwritten `get` and `set` functions.

```
rtwdemodbtype(fullfile('GetSetScalar_ert_rtw','GetSetScalar.c'),...
    '/* Model step function */','}',1,1)
```

```
/* Model step function */
void GetSetScalar_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  set_outSig(get_scalarParam() * get_inSig());
}
```

You can generate code that calls your custom `get` and `set` functions as long as the functions that you write accept and return the expected values. For scalar data, the functions must have these characteristics:

- The `get` function must return a single scalar numeric value of the appropriate data type, and must not accept any arguments (`void`).
- The `set` function must not return anything (`void`), and must accept a single scalar numeric value of the appropriate data type.

Use GetSet with Vector Data

This example shows how to apply the storage class `GetSet` to signals and parameters that are vectors.

View the example legacy header file `ComponentDataHeader.h`. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHeader.h','/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `vectors` is a substructure that uses the structure type `VectorData`. The structure type `VectorData` defines three vector fields: `inVector`, `vectorParam`, and `outVector`. The vectors each have five elements.

```
rtwdemodbtype('ComponentDataHeader.h','/* VectorData */','} VectorData;',1,1)
```

```
/* VectorData */
```

```
typedef struct {
    double inVector[5];
    double vectorParam[5];
    double outVector[5];
} VectorData;
```

View the example legacy source file `GetSetSourceData.c`. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `vectors`.

```
rtwdemodbtype('GetSetSourceData.c','/* Field "vectors" */','/* End of "vectors" */',1,1)
```

```
    /* Field "vectors" */
    {
        {5.7, 6.8, 1.2, 3.5, 10.1},

        {12.3, 18.7, 21.2, 28, 32.9},

        {0.0, 0.0, 0.0, 0.0, 0.0}
    },
    /* End of "vectors" */
```

The file also defines functions that read from and write to the fields of the substructure `vectors`. The functions simplify data access by dereferencing the leaf fields of the global structure variable `ex_getset_data`. To access the vector data, the functions accept an integer index argument. The `get` function returns the vector value at the input index. The `set` function assigns the input `value` to the input index.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Vector get() and set() functions */','/* End of vector functions */',1,1)
```

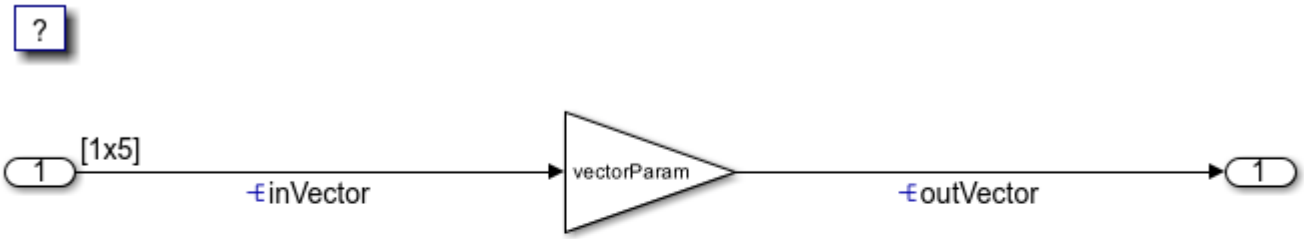
```
/* Vector get() and set() functions */

double get_inVector(int index)
{
    return ex_getset_data.vectors.inVector[index];
}

void set_inVector(int index, double value)
{
    ex_getset_data.vectors.inVector[index] = value;
}

double get_vectorParam(int index)
{
    return ex_getset_data.vectors.vectorParam[index];
}

void set_vectorParam(int index, double value)
{
    ex_getset_data.vectors.vectorParam[index] = value;
}

double get_outVector(int index)
{
    return ex_getset_data.vectors.outVector[index];
```

```
}

void set_outVector(int index, double value)
{
       ex_getset_data.vectors.outVector[index] = value;
}
```

View the example legacy header file `GetSetSourceDataVectorHdr.h`. The file contains the `extern` prototypes for the `get` and `set` functions defined in `GetSetSourceData.c`.

Open the example model `GetSetVector`. The model creates the data objects `inVector`, `outVector`, and `vectorParam` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('GetSetVector')
```



Copyright 2015-2022 The MathWorks, Inc.

In the base workspace, double-click the object `inVector` to view its properties. The object uses the storage class `GetSet`. The property `HeaderFile` is specified as `GetSetSourceDataVectorHdr.h`. This legacy header file contains the `get` and `set` function prototypes.

In the Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `GetSetSourceData.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
slbuild('GetSetVector');
```

```
### Starting build procedure for: GetSetVector
### Successful completion of build procedure for: GetSetVector

Build Summary

Top model targets built:

Model         Action                        Rebuild Reason
```

```
==========================================================================================
GetSetVector  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.034s
```

In the code generation report, view the file `GetSetVector.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('GetSetVector_ert_rtw','GetSetVector.c'),...
    '/* Model step function */','}',1,1)
```

```
/* Model step function */
void GetSetVector_step(void)
{
  int32_T i;
  for (i = 0; i < 5; i++) {
    /* Gain: '<Root>/Gain' incorporates:
     *  Inport: '<Root>/In1'
     */
    set_outVector(i, get_vectorParam(i) * get_inVector(i));
  }
```

When you use the storage class `GetSet` with vector data, the `get` and `set` functions that you provide must accept an index input. The `get` function must return a single element of the vector. The `set` function must write to a single element of the vector.

Use GetSet with Structured Data

This example shows how to apply the storage class `GetSet` to nonvirtual bus signals and structure parameters in a model.

View the example legacy header file `ComponentDataHeader.h`. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHeader.h',...
    '/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `structs` is a substructure that uses the structure type `StructData`. The structure type `StructData` defines three fields: `inStruct`, `structParam`, and `outStruct`.

```
rtwdemodbtype('ComponentDataHeader.h','/* StructData */','} StructData;',1,1)
```

```
/* StructData */
```

```
typedef struct {
    SigBus inStruct;
    ParamBus structParam;
    SigBus outStruct;
} StructData;
```

The fields `inStruct`, `structParam`, and `outStruct` are also substructures that use the structure types `SigBus` and `ParamBus`. Each of these two structure types define three scalar fields.

```
rtwdemodbtype('ComponentDataHeader.h','/* SigBus */','} ParamBus',1,1)
```

```
/* SigBus */

typedef struct {
    double cmd;
    double sensor1;
    double sensor2;
} SigBus;

/* ParamBus */

typedef struct {
    double offset;
    double gain1;
    double gain2;
} ParamBus;
```

View the example legacy source file `GetSetSourceData.c`. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `structs`.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Field "structs" */','/* End of "structs" */',1,1)
```

```
    /* Field "structs" */
    {
        {1.3, 5.7, 9.2},

        {12.3, 9.6, 1.76},

        {0.0, 0.0, 0.0}
    },
    /* End of "structs" */
```

The file also defines functions that read from and write to the fields of the substructure `structs`. The functions simplify data access by dereferencing the fields of the global structure variable `ex_getset_data`. The functions access the data in the fields `inStruct`, `structParam`, and `outStruct` by accepting and returning complete structures of the types `SigBus` and `ParamBus`.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Structure get() and set() functions */',...
    '/* End of structure functions */',1,1)
```

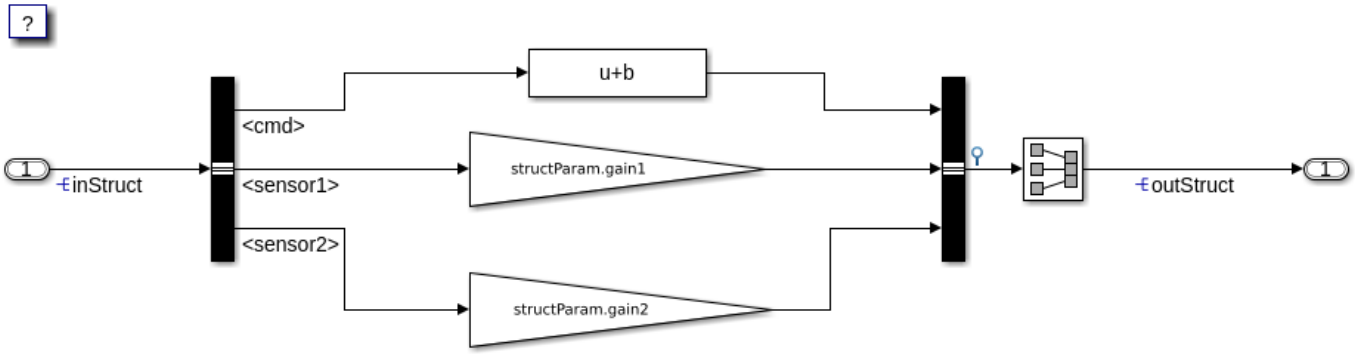
```
/* Structure get() and set() functions */
```

```
SigBus get_inStruct(void)
{
    return ex_getset_data.structs.inStruct;
}

void set_inStruct(SigBus value)
{
    ex_getset_data.structs.inStruct = value;
}

ParamBus get_structParam(void)
{
    return ex_getset_data.structs.structParam;
}

void set_structParam(ParamBus value)
{
    ex_getset_data.structs.structParam = value;
}

SigBus get_outStruct(void)
{
    return ex_getset_data.structs.outStruct;
}

void set_outStruct(SigBus value)
{
    ex_getset_data.structs.outStruct = value;
}
```

View the example legacy header file `GetSetSourceDataStructHdr.h`. The file contains the `extern` prototypes for the `get` and `set` functions defined in `GetSetSourceData.c`.

Open the example model `GetSetStruct`. The model creates the data objects `inStruct`, `structParam`, and `outStruct` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('GetSetStruct')
```



Copyright 2015-2022 The MathWorks, Inc.

In the base workspace, double-click the object `inStruct` to view its properties. The object uses the storage class `GetSet`. The property `HeaderFile` is specified as `GetSetSourceDataStructHdr.h`. This legacy header file contains the `get` and `set` function prototypes.

The model also creates the bus objects `ParamBus` and `SigBus` in the base workspace. The signals and parameter in the model use the bus types that these objects define. The property `DataScope` of each bus object is set to `Imported`. The property `HeaderFile` is set to `ComponentDataHeader.h`. The generated code imports these structure types from the legacy header file `ComponentDataHeader.h`.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `GetSetSourceData.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
slbuild('GetSetStruct');
```

```
### Starting build procedure for: GetSetStruct
### Successful completion of build procedure for: GetSetStruct

Build Summary

Top model targets built:

Model         Action                        Rebuild Reason
========================================================================================
GetSetStruct  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 30.612s
```

In the code generation report, view the file `GetSetStruct.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('GetSetStruct_ert_rtw',...
    'GetSetStruct.c'),...
    '/* Model step function */','}',1,1)
```

```
/* Model step function */
void GetSetStruct_step(void)
{
  /* Bias: '<Root>/Bias' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.BusCreator.cmd = (get_inStruct()).cmd + (get_structParam()).offset;

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.BusCreator.sensor1 = (get_structParam()).gain1 * (get_inStruct()).sensor1;

  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In1'
   */
```

```
  rtDW.BusCreator.sensor2 = (get_structParam()).gain2 * (get_inStruct()).sensor2;

  /* SignalConversion: '<Root>/Signal Conversion' */
  set_outStruct(rtDW.BusCreator);
}
```

When you use the storage class `GetSet` with structured data, the `get` and `set` functions that you provide must return and accept complete structures. The generated code dereferences individual fields of the structure that the `get` function returns.

The output signal of the Bus Creator block is a test point. This signal is the input for a Signal Conversion block. The test point and the Signal Conversion block exist so that the generated code defines a variable for the output of the Bus Creator block. To provide a complete structure argument for the function `set_outStruct`, you must configure the model to create this variable.

Use GetSet with Matrix Data

This example shows how to apply the storage class `GetSet` to signals and parameters that are matrices.

View the example legacy header file `ComponentDataHeader.h`. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHeader.h',...
    '/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `matrices` is a substructure that uses the structure type `MatricesData`. The structure type `MatricesData` defines three fields: `matrixInput`, `matrixParam`, and `matrixOutput`. The fields store matrix data as serial arrays. In this case, the input and parameter fields each have 15 elements. The output field has nine elements.

```
rtwdemodbtype('ComponentDataHeader.h'...
    ,'/* MatricesData */','} MatricesData;',1,1)
```

```
/* MatricesData */

typedef struct {
    double matrixInput[15];
    double matrixParam[15];
    double matrixOutput[9];
} MatricesData;
```

View the example legacy source file `GetSetSourceData.c`. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `matrices`.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Field "matrices" */','/* End of "matrices" */',1,1)
```

```
    /* Field "matrices" */
    {
        {12.0, 13.9, 7.4,
         0.5, 11.8, 6.4,
         4.7, 5.3, 13.0,
         0.7, 16.1, 13.5,
         1.6, 0.5, 3.1},

        {8.3, 12.0, 11.5, 2.0, 5.7,
         7.5, 12.8, 11.1, 8.4, 9.9,
         10.9, 4.6, 2.7, 16.3, 3.8},

        {0.0, 0.0, 0.0,
         0.0, 0.0, 0.0,
         0.0, 0.0, 0.0}
    }
    /* End of "matrices" */
```

The input matrix has five rows and three columns. The matrix parameter has three rows and five columns. The matrix output has three rows and three columns. The file defines macros that indicate these dimensions.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Matrix dimensions */','/* End of matrix dimensions */',1,1)
```

```
/* Matrix dimensions */

#define MATRIXINPUT_NROWS 5
#define MATRIXINPUT_NCOLS 3

#define MATRIXPARAM_NROWS 3
#define MATRIXPARAM_NCOLS 5

#define MATRIXOUTPUT_NROWS MATRIXPARAM_NROWS
#define MATRIXOUTPUT_NCOLS MATRIXINPUT_NCOLS
```

The file also defines functions that read from and write to the fields of the substructure `matrices`.

```
rtwdemodbtype('GetSetSourceData.c',...
    '/* Matrix get() and set() functions */','/* End of matrix functions */',1,1)
```

```
/* Matrix get() and set() functions */

double get_matrixInput(int colIndex)
{
    int rowIndexGetInput = MATRIXINPUT_NCOLS * (colIndex % MATRIXINPUT_NROWS) + colIndex/MATRIXI
    return ex_getset_data.matrices.matrixInput[rowIndexGetInput];
}

void set_matrixInput(int colIndex, double value)
{
    int rowIndexSetInput = MATRIXINPUT_NCOLS * (colIndex % MATRIXINPUT_NROWS) + colIndex/MATRIXI
```

```
    ex_getset_data.matrices.matrixInput[rowIndexSetInput] = value;
}

double get_matrixParam(int colIndex)
{
    int rowIndexGetParam = MATRIXPARAM_NCOLS * (colIndex % MATRIXPARAM_NROWS) + colIndex/MATRIXPA
    return ex_getset_data.matrices.matrixParam[rowIndexGetParam];
}

void set_matrixParam(int colIndex, double value)
{
    int rowIndexSetParam = MATRIXPARAM_NCOLS * (colIndex % MATRIXPARAM_NROWS) + colIndex/MATRIXPA
    ex_getset_data.matrices.matrixParam[rowIndexSetParam] = value;
}

double get_matrixOutput(int colIndex)
{
    int rowIndexGetOut = MATRIXOUTPUT_NCOLS * (colIndex % MATRIXOUTPUT_NROWS) + colIndex/MATRIXOU
    return ex_getset_data.matrices.matrixOutput[rowIndexGetOut];
}

void set_matrixOutput(int colIndex, double value)
{
    int rowIndexSetOut = MATRIXOUTPUT_NCOLS * (colIndex % MATRIXOUTPUT_NROWS) + colIndex/MATRIXOU
    ex_getset_data.matrices.matrixOutput[rowIndexSetOut] = value;
}
```

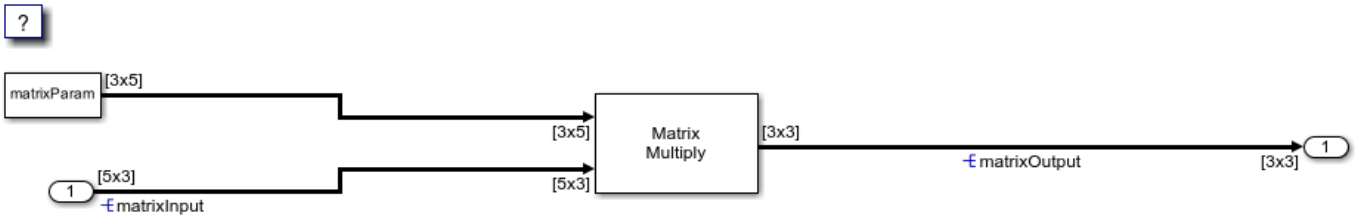
The code that you generate from a model represents matrices as serial arrays. Therefore, each of the `get` and `set` functions accept a single scalar index argument.

The generated code uses column-major format to store and to access matrix data. However, many C applications use row-major indexing. To integrate the generated code with the example legacy code, which stores the matrices `matrixInput` and `matrixParam` using row-major format, the custom `get` functions use the column-major index input to calculate an equivalent row-major index. The generated code algorithm, which interprets matrix data using column-major format by default, performs the correct matrix math because the `get` functions effectively convert the legacy matrices to column-major format. The `set` function for the output, `matrixOutput`, also calculates a row-major index so the code writes the algorithm output to `matrixOutput` using row-major format. Alternatively, to integrate the column-major generated code with your row-major legacy code, you can manually convert the legacy code to column-major format by transposing your matrix data and algorithms.

View the example legacy header file `GetSetSourceDataMatrixHdr.h`. The file contains the `extern` prototypes for the `get` and `set` functions defined in `GetSetSourceData.c`.

Open the example model `GetSetMatrix`. The model creates the data objects `matrixInput`, `matrixParam`, and `matrixOutput` in the base workspace. The objects correspond to the signals and parameter in the model.

```
load_system('GetSetMatrix')
set_param('GetSetMatrix','SimulationCommand','Update')
open_system('GetSetMatrix')
```



Copyright 2015-2022 The MathWorks, Inc.

In the base workspace, double-click the object `matrixInput` to view its properties. The object uses the storage class `GetSet`. The property `HeaderFile` is specified as `GetSetSourceDataMatrixHdr.h`. This legacy header file contains the `get` and `set` function prototypes.

In the Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `GetSetSourceData.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
slbuild('GetSetMatrix');
```

```
### Starting build procedure for: GetSetMatrix
### Successful completion of build procedure for: GetSetMatrix

Build Summary

Top model targets built:

Model         Action                        Rebuild Reason
========================================================================================
GetSetMatrix  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.759s
```

In the code generation report, view the file `GetSetMatrix.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('GetSetMatrix_ert_rtw',...
    'GetSetMatrix.c'),'/* Model step function */','}',1,1)
```

```
/* Model step function */
void GetSetMatrix_step(void)
{
  int32_T i;
  int32_T i_0;
  int32_T i_1;
  int32_T matrixOutput_tmp;
  for (i_0 = 0; i_0 < 3; i_0++) {
    for (i = 0; i < 3; i++) {
      /* Product: '<Root>/Product' incorporates:
```

```
 *  Constant: '<Root>/Constant'
 *  Inport: '<Root>/In1'
 */
matrixOutput_tmp = 3 * i_0 + i;
set_matrixOutput(matrixOutput_tmp, 0.0);
for (i_1 = 0; i_1 < 5; i_1++) {
  set_matrixOutput(matrixOutput_tmp, get_matrixParam(3 * i_1 + i) *
                   get_matrixInput(5 * i_0 + i_1) + get_matrixOutput
                   (matrixOutput_tmp));
}
```

Specify Header File or Function Naming Scheme for Data Items

By default, you specify a header file name, `get` function name, and `set` function name for each data item, such as a signal or parameter, that uses the storage class `GetSet`.

To configure a single header file, `get` function naming scheme, or `set` function naming scheme to use for every data item, you can use the Custom Storage Class Designer to create your own copy of `GetSet`. You can specify the header file or function names in a single location.

Follow these steps to create your own storage class by creating your own data class package, creating a copy of `GetSet`, and applying the new storage class to data items in your model.

1. Create your own data class package, `myPackage`, by copying the example package folder `+SimulinkDemos` and renaming the copied folder as `+myPackage`. Modify the `Parameter` and `Signal` class definitions so that they use the storage class definitions from `myPackage`. For an example, see “Create Data Class Package” on page 40-14.
2. Set your current folder to the folder that contains the package folder. Alternatively, add the folder containing the package folder to your MATLAB path.
3. Open the Custom Storage Class Designer.

   ```
   cscdesigner('myPackage')
   ```
4. Select the storage class `GetSet`. Click **Copy** to create a copy called `GetSet_1`.
5. Select the new storage class `GetSet_1`. In the **General** tab, set **Name** to `myGetSet`.
6. Set the drop-down list **Header file** to `Specify`. In the new text box, set **Header file** to `myFcnHdr.h`. Click **Apply**.
7. On the **Access Function Attributes** tab, set the drop-down lists **Get function** and **Set function** to `Specify`.
8. In the new boxes, set **Get function** to `myGetFcn_$N` and **Set function** to `mySetFcn_$N`. Click **OK**. Click **Yes** in response to the message about saving changes.

   When you generate code, the token `$N` expands into the name of the data item that uses this storage class.
9. Apply the storage class `myGetSet` from your package to a data item. For example, create a `myPackage.Parameter` object in the base workspace.

   ```
   myParam = myPackage.Parameter(15.23);
   myParam.CoderInfo.StorageClass = 'Custom';
   myParam.CoderInfo.CustomStorageClass = 'myGetSet';
   ```
10. Use the object to set a parameter value in your model. When you generate code, the code algorithm accesses the parameter through the functions that you specified. The code uses an `#include` directive to include the header file that you specified.

Access Scalar and Array Data Through Macro Instead of Function Call

If you implement the `get` mechanism for scalar or array data as a macro instead of a function, you can generate code that omits parentheses when reading that data.

- For scalar data, your macro must yield the scalar value.
- For array data, your macro must yield the starting memory address.

Create your own `AccessFunction` storage class by using the Custom Storage Class Designer, as described in “Specify Header File or Function Naming Scheme for Data Items” on page 40-39. In the Designer, on the **Access Function Attributes** tab, select **Get data through macro (omit parentheses)**.

GetSet Storage Class Restrictions

- `GetSet` does not support complex signals.
- Multiple data in the same model cannot use the same `GetFunction` or `SetFunction`.
- Some blocks do not directly support `GetSet`.
- Custom S-functions do not directly support `GetSet`.

To use `GetSet` with an unsupported block or a custom S-function:

1. Insert a Signal Conversion block at the output of the block or function.
2. In the Signal Conversion block dialog box, select **Exclude this block from 'Block reduction' optimization**.
3. Assign the storage class `GetSet` to the output of the Signal Conversion block.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Create and Apply a Storage Class” on page 40-14
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23
- “Generate Code That Dereferences Data from a Literal Memory Address” on page 71-15

Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary

To replace the direct access to data in the generated code with your own or legacy functions that read and write data in a customized way, you can enable function access on your storage classes. You create these storage classes in the Embedded Coder Dictionary. Root-level inports, root-level outports, and model parameters that use your storage class appear in the generated code as calls to specified functions. You provide the function declarations and definitions in separate header and source files pointed to by the storage class.

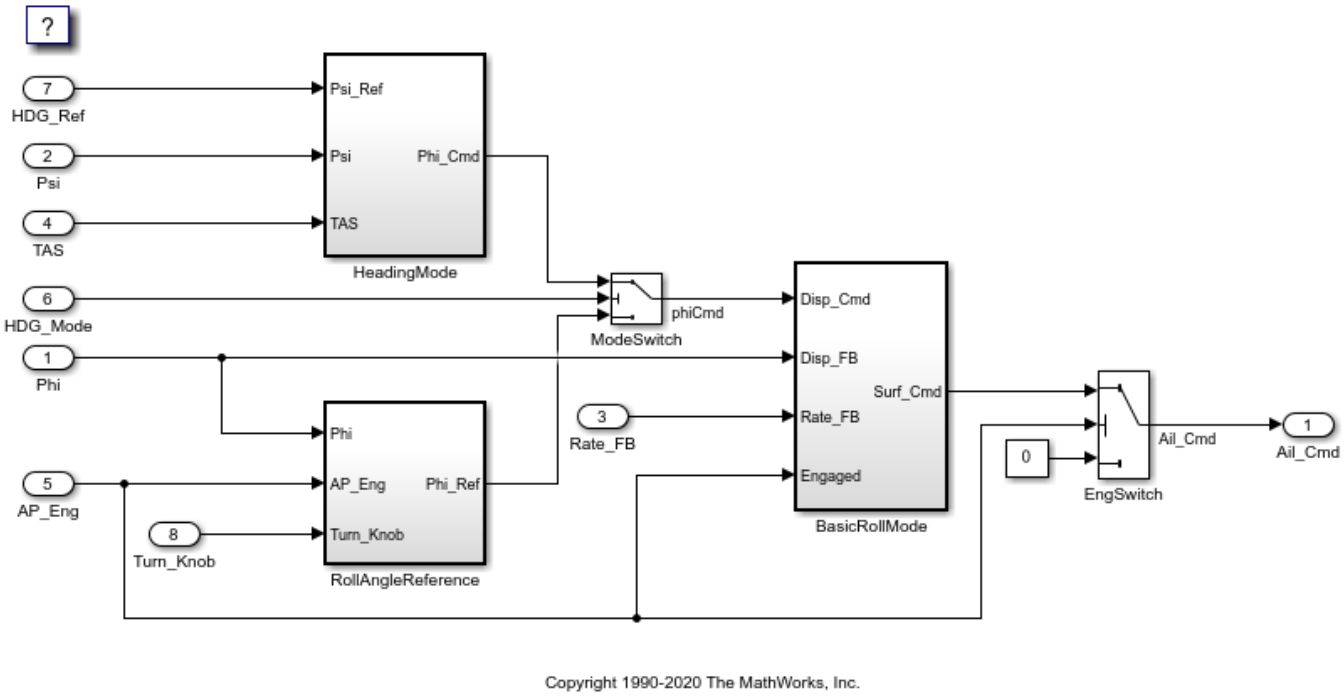
Open Embedded Coder Dictionary

You can define and apply a storage class to root-level inports, root-level outports, and model parameters so they can be accessed by customizable get and set functions. Such customization can be useful, for example, to abstract layers of software, gain access to data from an external file or to control access to critical sections of code.

This example shows how to set up a storage class in the Embedded Coder Dictionary and map the storage class to root-level inports/outports and model parameters in the Code Mappings editor.

Open example model `RollAxisAutopilot`.

```
open_system('RollAxisAutopilot')
```



The example gives you access to the supporting files that define the `get` and `set` functions required for the example: `RollAxisAutopilot_Value.h`, `RollAxisAutopilot_Value.c`, `RollAxisAutopilot_Pointer.h`, and `RollAxisAutopilot_Pointer.c`.

1. In the Apps gallery, under **Code Generation**, click **Embedded Coder**.
2. To open the Embedded Coder Dictionary, on the **C Code** tab, click **Code Interface** > **Embedded Coder Dictionary**.

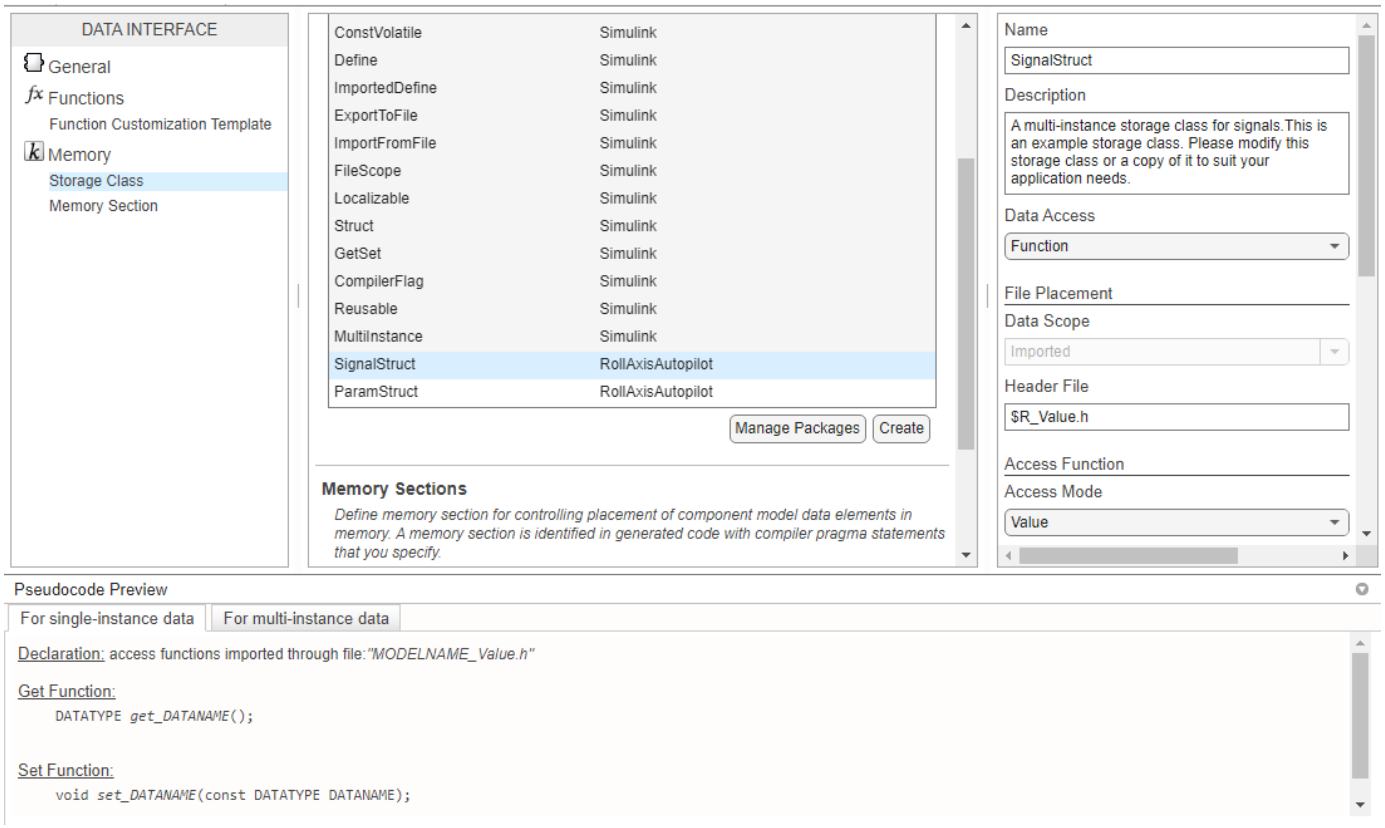
Access Legacy Data by Value

To access data by using a value, follow these steps:

1. In the Embedded Coder Dictionary, on the left pane, click **Storage Class**.
2. The `RollAxisAutopilot` model has two storage classes: `SignalStruct` and `ParamStruct`. They are already created by using the **Create** button. This example uses these storage classes. You can perform the following steps for a storage class that you create as well.
3. For the `SignalStruct` storage class, in the **Property Inspector** pane, update these property values:

   - **Data Access** to `Function`
   - **Header File** to `$R_Value.h`

   Make sure that **Access Mode** is set to `Value`. **Data Scope** is set to `Imported` by default and cannot be changed.



For the `ParamStruct` storage class, in the **Property Inspector** pane, update these property values:

- **Data Access** to `Function`

- **Header File** to `$R_Value.h`

These properties are already set:

- **Access Mode** to `Value`
- **Data Initialization** to `Static`

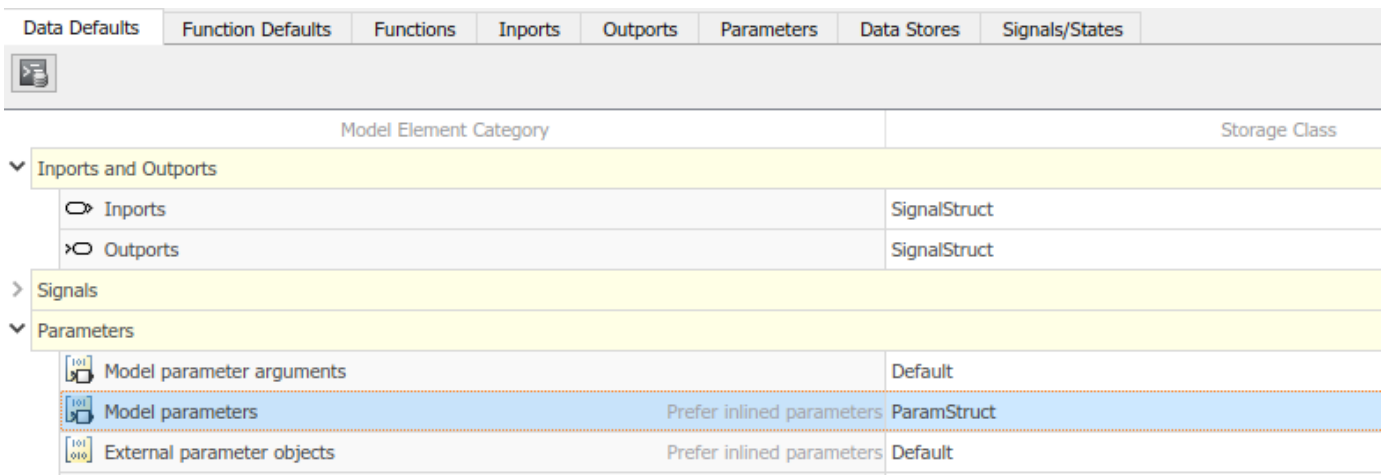
Storage classes used for parameters can have **Data Initialization** set to `Auto`,`Static`, or `None`.

4 Open the Configuration Parameters dialog box. In the toolstrip, on the **C Code** tab, click **Settings**. Select **Code Generation** > **Custom Code** > **Code information** > **Source files** and specify:

```
RollAxisAutopilot_Value.c
```

5 In the Embedded Coder app, open the Code Mappings editor. On the **Data Defaults** tab, map these model element categories:

- **Inports** to `SignalStruct`
- **Outports** to `SignalStruct`
- **Model parameters** to `ParamStruct`



6 Build the model and generate code.

In the generated code, view the file `RollAxisAutopilot.c`. The model `step` function uses the specified `get` and `set` functions to execute the algorithm. The generated code accesses the legacy data by calling the custom, handwritten `get` and `set` functions. For example, here is the code snippet for the HDG_Ref inport:

```
if (get_HDG_Mode()) {
    /* Outputs for Atomic SubSystem: '<Root>/HeadingMode' */
    rtb_Sum1 = (get_HDG_Ref() - get_Psi()) * 0.015F * get_TAS();

    /* End of Outputs for SubSystem: '<Root>/HeadingMode' */
  } else {
    /* Outputs for Atomic SubSystem: '<Root>/RollAngleReference' */
    if ((real32_T)fabs(get_Turn_Knob()) >= 3.0F) {
      rtb_Sum1 = get_Turn_Knob();
    }
```

You can generate code that calls your custom `get` and `set` functions if the functions that you write accept and return the expected values.

Access Legacy Data by Pointer

To access data by using a pointer, follow all the preceding steps but make these changes:

- In the Embedded Coder Dictionary, for storage classes `SignalStruct` and `ParamStruct`, specify:
  - **Access Mode** to `Pointer`
  - **Header File** to `$R_Pointer.h`
- Select **Configuration Parameters** > **Code Generation** > **Custom Code** > **Code information** > **Source files** and replace `RollAxisAutopilot_Value.c` with:

  ```
  RollAxisAutopilot_Pointer.c
  ```

With the pointer access, a `get` function now returns a pointer whose value must be dereferenced with the * before use. For example, the generated code for the HDG_Ref inport now looks like this code:

```
if (*get_HDG_Mode()) {
    /* Outputs for Atomic SubSystem: '<Root>/HeadingMode' */
    rtb_Sum1 = (*get_HDG_Ref() - *get_Psi()) * 0.015F * *get_TAS();

    /* End of Outputs for SubSystem: '<Root>/HeadingMode' */
  } else {
    /* Outputs for Atomic SubSystem: '<Root>/RollAngleReference' */
    if ((real32_T)fabs(tmp_1) >= 3.0F) {
      rtb_Sum1 = tmp_1;
    }
```

Creating your own storage class in Embedded Coder Dictionary gives you the flexibility of customizing function names and return types. You can also use the storage class `GetSet` to access data through `get` and `set` functions. For more information, see “Access Data Through Functions with Storage Class GetSet” on page 40-25.

See Also

Embedded Coder Dictionary

Related Examples

- “Access Data Through Functions with Storage Class GetSet” on page 40-25
- “Access Structured Data Through a Pointer That External Code Defines” on page 40-10

Integrate External Application Code with Code Generated from PID Controller

This example shows how to generate C code from a control algorithm model and integrate that code with existing, external application code. In the complete application, which consists of three modules, the algorithm module must exchange data with other modules through global variables. In the example, you configure the generated code to interact with existing external variables, resemble the external code in appearance and organization, and conform to specific coding requirements.

Inspect External Code

Run the script `prepare_ext_code`. The script copies external code files into several folders.

```
exampleFolder = pwd;
run('prepare_ext_code.m');
```

Open `ext_code_main.c`. This application code represents an embedded system with a trivial scheduling algorithm (a `while` loop) and three modules that include algorithms that run in every execution cycle: `ex_ext_inputs_proc`, `ex_ext_ctrl_alg`, and `ex_ext_outputs_proc`.

```
type('ext_code_main.c')
```

In the `shared` folder, inspect `ex_ext_projTypes.h`. The file defines two custom data types (`typedef`) that the data in the modules use.

```
type(fullfile('shared','ex_ext_projTypes.h'))
```

In the `io_drivers` folder, inspect `ex_sensor_accessors.c`. This file defines functions `get_fromSensor_flow` and `get_fromSensor_temp`, which return raw data recorded by a flow sensor and a temperature sensor. For this example, the functions return trivial sinusoidal stimuli for the control algorithm.

```
type(fullfile('io_drivers','ex_sensor_accessors.c'))
```

In the `ex_ext_inputs_proc` folder, inspect `ex_ext_inputs_proc.c`. The `ex_ext_inputs_proc` module reads the sensor data (by calling the accessor functions), filters the data, and stores it in two global variables, `PROC_INPUT_FLOW` and `PROC_INPUT_TEMP`. These global variables are defined in the file `ex_ext_proc_inputs.c` and declared in `ex_ext_proc_inputs.h`.

```
type(fullfile('ex_ext_inputs_proc','ex_ext_inputs_proc.c'))
```

The filter algorithm in this module and the algorithms in the other two modules require state data, which must persist between execution cycles of the application. Each module stores relevant state data as global variables. For example, in the `ex_ext_inputs_proc` module, `ex_ext_inputs_proc.c` defines these variables:

- `flowFilterIn_state_data`
- `tempFilterIn_state_data`
- `flowFilterIn_tmp_data`
- `tempFilterIn_tmp_data`

The empty `ex_ext_ctrl_alg` folder is a placeholder for the generated code. The `ex_ext_ctrl_alg` module must perform PID control on the filtered flow and temperature

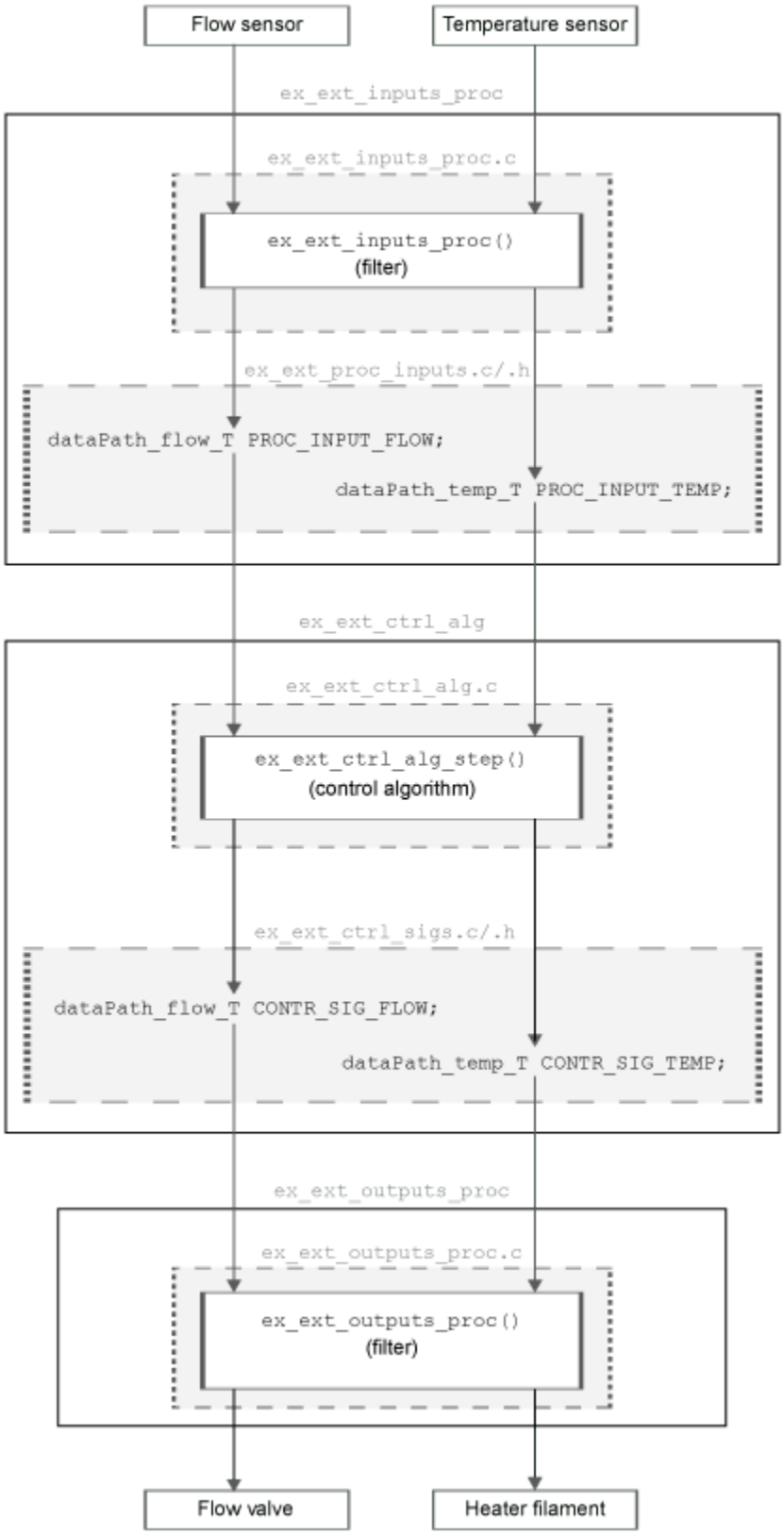
measurements. Later, you examine the requirements for this code, which are based on the code in the other modules.

In the `ex_ext_outputs_proc` folder, inspect `ex_ext_outputs_proc.c`. This module reads the PID output signals from global variables named `CONTR_SIG_FLOW` and `CONTR_SIG_TEMP`. The `ex_ext_ctrl_alg` module (the generated code) must define these variables. The `ex_ext_outputs_proc` module filters these signals and passes the filtered values to functions that represent device drivers for actuators (for example, a valve and a heater filament).

```
type(fullfile('ex_ext_outputs_proc','ex_ext_outputs_proc.c'))
```

The actuator functions are defined in the `io_drivers` folder in the file `ex_actuator_accessors.c`.

The figure shows the intended flow of data through the system. The figure also shows the code files that define and declare the data (global variables) that cross the module boundaries.


Flow sensor
Temperature sensor
ex_ext_inputs_proc
ex_ext_inputs_proc.c
ex_ext_inputs_proc()
(filter)
ex_ext_proc_inputs.c/.h
dataPath_flow_T PROC_INPUT_FLOW;
dataPath_temp_T PROC_INPUT_TEMP;
ex_ext_ctrl_alg
ex_ext_ctrl_alg.c
ex_ext_ctrl_alg_step()
(control algorithm)
ex_ext_ctrl_sigs.c/.h
dataPath_flow_T CONTR_SIG_FLOW;
dataPath_temp_T CONTR_SIG_TEMP;
ex_ext_outputs_proc
ex_ext_outputs_proc.c
ex_ext_outputs_proc()
(filter)
Flow valve
Heater filament

Inspect Simulink Model and Determine Requirements for Generated Code

Open the example model `ex_ext_ctrl_alg`.

```
open('ex_ext_ctrl_alg')
```

The blocks in the model implement the required PID controller algorithm. The model uses default code generation settings for an ERT-based system target file (`ert.tlc`).

To complete the application, you must generate code that performs the function of the `ex_ext_ctrl_alg` module. The code generated from the model must:

- Use the custom data types `dataPath_flow_T` and `dataPath_temp_T` declared in the external file `ex_ext_projTypes.h`.
- Read filtered sensor data from global variables `PROC_INPUT_FLOW` and `PROC_INPUT_TEMP`. The generated code must not define these variables because the module `ex_ext_inputs_proc` defines them, declaring them in `ex_ext_proc_inputs.h`.
- Write PID control signals to global variables named `CONTR_SIG_FLOW` and `CONTR_SIG_TEMP`. The generated code must define these variables and declare them in file `ex_ext_ctrl_sigs.h`. Then, the `ex_ext_outputs_proc` module can read the raw control signals from them.
- Conform to variable naming standards that govern the external application. For example, the generated code must store state data in global variables with names that end in `_data`. In addition, for this example, the names of local function variables must end in `_local`.
- Conform to standards that govern the organization of the code in each of the external files. For example, each file contains sections, delimited by comments, which aggregate similar code constructs such as type definitions, variable declarations, and function definitions.

So that you and others can interact with the control algorithm during execution, you configure the generated code to define and declare `const` global variables named `PARAM_setpoint_flow` and `PARAM_setpoint_temp`, which represent the PID controller setpoints. The definitions must be in a file named `ex_ext_ctrl_params.c` and the declarations must be in a file named `ex_ext_ctrl_params.h`.

Configure Model to Use Custom Data Types

Set your current folder to the `shared` folder.

Use the function `Simulink.importExternalCTypes` to generate `Simulink.AliasType` objects that represent the custom data types `dataPath_flow_T` and `dataPath_temp_T`.

```
cd('shared');
Simulink.importExternalCTypes('ex_ext_projTypes.h');
```

The objects appear in the base workspace.

1. Open the model `ex_ext_ctrl_alg` model.

2. Open the Embedded Coder app.

3. Open the Model Data Editor by using the tab near the bottom the app window.

4. In the Model Data Editor, for the Inport block that represents `PROC_INPUT_FLOW`, set **Data Type** to `dataPath_flow_T`. To add the custom data types to the **Data Type** list, click `Refresh data types`.

5. For the Inport block that represents `PROC_INPUT_TEMP`, set **Data Type** to `dataPath_temp_T`.

6. Select the **Signals** tab.

7. Configure the data type for the Constant blocks. For each Constant block, in the model, select the block output signal. Then, in the Model Data Editor, set **Data Type** to `Inherit: Inherit via back propagation`. With this setting, the Constant blocks inherit the output data type of the block immediately downstream. In this case, the downstream block is a Sum block.

Alternatively, to configure the data types programmatically or at the command prompt, use these commands.

```
set_param('ex_ext_ctrl_alg/In1','OutDataTypeStr','dataPath_flow_T')
set_param('ex_ext_ctrl_alg/In2','OutDataTypeStr','dataPath_temp_T')
set_param('ex_ext_ctrl_alg/Flow Setpt','OutDataTypeStr',...
    'Inherit: Inherit via back propagation')
set_param('ex_ext_ctrl_alg/Temp Setpt','OutDataTypeStr',...
    'Inherit: Inherit via back propagation')
```

Update the block diagram. The diagram shows that, due to data type inheritance and propagation, signals in the model use a custom data type.

Configure Code Interface for Inports and Outports

Configure the model to generate code that accesses and defines global variables for model inports and outports.

1. In the Embedded Coder app, on the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

2. In the Code Mappings editor, click the **Data Defaults** tab.

3. Expand **Inports and Outports**.

4. For **Inports**, set **Storage Class** to `ImportFromFile`. Click the pencil icon and set the **HeaderFile** property to `ex_ext_proc_inputs.h`. With the storage class `ImportFromFile`, each inport appears in the generated code as a global variable. Instead of defining the variable, the generated code includes (`#include`) the variable declaration from header file `ex_ext_proc_inputs.h`.

6. For **Outports**, set **Storage Class** to `ExportToFile`. Click the pencil icon and set the **HeaderFile** property to `ex_ext_ctrl_sigs.h` and the **DefinitionFile** property to `ex_ext_ctrl_sigs.c`. With the storage class `ExportToFile`, the generated code defines the global variable.

7. On the **Inports** tab, for each inport, set **Storage Class** to `Model default: ImportFromfile`.

8. On the **Outports** tab, for each outport, set **Storage Class** to `Model default: ExportToFile`.

9. Save the model. Alternatively, to configure interfaces for the inports and outports programmatically or at the command prompt, use these commands.

```
cm = coder.mapping.api.get('ex_ext_ctrl_alg');
setDataDefault(cm,'Inports','StorageClass','ImportFromFile','HeaderFile', ...
    'ex_ext_proc_inputs.h');
setInport(cm,'In1','StorageClass','Model default');
setInport(cm,'In2','StorageClass','Model default');
```

```
setDataDefault(cm,'Outports','StorageClass','ExportToFile','HeaderFile', ...
    'ex_ext_ctrl_sigs.h','DefinitionFile','ex_ext_ctrl_sigs.c');
setOutport(cm,'Out1','StorageClass','Model default');
setOutport(cm,'Out2','StorageClass','Model default');
```

Configure Code Interface for Constant Parameters

To apply a storage class to the parameters represented by Constant blocks `Flow Setpt` and `Temp Setpt`, use the Model Explorer to create `Simulink.Parameter` objects to represent the data. Create two model workspace parameters.

1. Open the Model Explorer.

2. In the **Model Hierarchy** pane, select **Model Workspace**.

3. Create two parameter data objects. Select **Add** > **Simulink Parameter**. Name the parameter objects `PARAM_setpoint_flow` and `PARAM_setpoint_temp`. Set the value of `PARAM_setpoint_flow` to 3. Set the value of `PARAM_setpoint_temp` to 2.

4. To configure the storage class for the model parameters, click **Configure**.

5. In the Embedded Coder app, in the Code Mappings editor, click the **Data Defaults** tab.

6. Expand **Parameters**.

7. For **Model Parameters**, set **Storage class** to `Const`. Click the pencil icon and set the **HeaderFile** property to `ex_ext_ctrl_params.h` and the **DefinitionFile** property to `ex_ext_ctrl_params.c`.

8. On the **Parameters** tab, for each parameter, set **Storage Class** to `Model default: Const`.

9. In the model, for Constant block `Flow Setpt`, set block parameter **Constant value** to `PARAM_setpoint_flow`. For Constant block `Temp Setpt`, set **Constant value** to `PARAM_setpoint_temp`.

10. For the parameters to be tunable, set model configuration parameter **Default parameter behavior** to `Tunable`.

11. Save the model.

Alternatively, to configure the parameter interfaces programmatically or at the command prompt, use these commands.

```
setDataDefault(cm,'ModelParameters','StorageClass','Const','HeaderFile', ...
    'ex_ext_ctrl_params.h','DefinitionFile','ex_ext_ctrl_params.c');
setModelParameter(cm,'PARAM_setpoint_flow','StorageClass','Model default');
setModelParameter(cm,'PARAM_setpoint_temp','StorageClass','Model default');
set_param('ex_ext_ctrl_alg','DefaultParameterBehavior','Tunable');
```

Configure Code for States

In the external code, internal data that does not participate in the module interfaces, such as state data and local variables in functions, conform to naming schemes. In the model, configure internal data that appears in the generated code to align with the external code naming schemes.

1. In the model Configuration Parameters dialog box, open the **Code Generation** > **Identifiers** pane.

2. Set **Global variables** to naming rule `$R$N_data$M`. This setting controls the names of global variables, such as those that represent state data. The naming rule `$R$N_data_$M` most closely approximates the scheme that the state variables in the external code use. In the naming rule:

- `$R` represents the name of the model, `ex_ext_ctrl_alg`.
- `$N` represents the name of the model element to which the naming rule applies, such as a signal, block state, or standard data structure.
- `$M` represents name-mangling text that the code generator inserts to avoid name clashes. For most naming rules, this token is required.

3. Set **Local temporary variables** and **Local block output variables** to `$N_local$M`.

Alternatively, to set the configuration parameters programmatically or at the command prompt, use these commands.

```
set_param('ex_ext_ctrl_alg','CustomSymbolStrGlobalVar','$R$N_data$M')
set_param('ex_ext_ctrl_alg','CustomSymbolStrTmpVar','$N_local$M')
set_param('ex_ext_ctrl_alg','CustomSymbolStrBlkIO','$N_local$M')
```

Configure the state data in the model to appear in the generated code as separate global variables instead of fields of the standard DWork structure.

1. In the Code Mappings editor, click the **Data Defaults** tab.

2. Expand **Signals** and select the `Signals, states, and internal data` row.

3. Set the storage class to `ExportedGlobal`.

4. On the **Signals/States** tab, for each state, set the storage class to `Model default: Exported Global`.

5. Save the model.

Alternatively, to configure the state data for code generation programmatically or at the command prompt, use these commands:

```
setDataDefault(cm,'InternalData','StorageClass','ExportedGlobal');
blockHandles = find(cm, 'States');
setState(cm,blockHandles,'StorageClass','Model default');
```

Configure Organization of Code in Generated Files

By default, when you generate code with system target file `ert.tlc`, the code generator uses built-in code generation template (CGT) file `ert_code_template.cgt`. The template governs the organization and appearance of code in each generated file.

For this example, use the example template file `ex_my_code_template.cgt` instead. The example template file conforms more closely to the organization of the files for this example. For example, the template organizes similar code constructs into named sections (delimited by comments) and, at the top of the template, specifies minimal information about each generated file.

```
cd(exampleFolder)
type('ex_my_code_template.cgt')
```

1. Copy `ex_my_code_template.cgt` to the `ex_ext_ctrl_alg` folder.

```
copyfile("ex_my_code_template.cgt","ex_ext_ctrl_alg");
```

2. In the model Configuration Parameters dialog box, open the **Code Generation** > **Templates** pane.

3. Under **Code templates** and **Data templates**, set parameters **Sourc file template** and **Header file template** to|ex_my_code_template.cgt|.

Alternatively, to set the parameters programmatically or at the command prompt, use these commands.

```
set_param('ex_ext_ctrl_alg','ERTSrcFileBannerTemplate','ex_my_code_template.cgt')
set_param('ex_ext_ctrl_alg','ERTHdrFileBannerTemplate','ex_my_code_template.cgt')
set_param('ex_ext_ctrl_alg','ERTDataSrcFileTemplate','ex_my_code_template.cgt')
set_param('ex_ext_ctrl_alg','ERTDataHdrFileTemplate','ex_my_code_template.cgt')
```

Generate and Inspect Code

Because the external code defines a `main` function, select model configuration parameter **Generate code only** and clear parameter **Generate an example main program**.

```
set_param('ex_ext_ctrl_alg','GenCodeOnly','on')
set_param('ex_ext_ctrl_alg','GenerateSampleERTMain','off')
```

Generate code from the model.

```
slbuild('ex_ext_ctrl_alg')
```

In the **Code** view or code generation report, inspect the generated files to confirm that the code meets the requirements.

The file `ex_ext_ctrl_sigs.c` defines the control output signals `CONTR_SIG_FLOW` and `CONTR_SIG_TEMP`.

```
dataPath_flow_T CONTR_SIG_FLOW;
dataPath_temp_T CONTR_SIG_TEMP;
```

The setpoint parameters appear in `ex_ext_ctrl_params.c`.

```
const dataPath_flow_T PARAM_setpoint_flow = 3.0;
const dataPath_temp_T PARAM_setpoint_temp = 2.0;
```

The file `ex_ext_ctrl_alg.c` defines global variables to store state data. The variables follow the specified naming rule.

```
dataPath_temp_T ex_ext__Integrator_DSTATE_datag;
dataPath_flow_T ex_ext_c_Integrator_DSTATE_data;
dataPath_temp_T ex_ext_ctrl_Filter_DSTATE_datad;
dataPath_flow_T ex_ext_ctrl__Filter_DSTATE_data;
```

In the same file, the model execution function, `ex_ext_ctrl_alg_step`, creates local function variables to store temporary calculations. The variables follow the specified naming scheme.

```
dataPath_flow_T FilterCoefficient_local;
dataPath_flow_T IntegralGain_local;
dataPath_temp_T Diff1_local;
dataPath_temp_T FilterCoefficient_locali;
```

Some of the global and local variable names contain extra mangling characters that prevent name clashes. The extra characters correspond to the `$M` token in the specified naming rules.

The generated code files are in the generated folders `ex_ext_ctrl_alg_ert_rtw` and `slprj`. You must configure file management systems and build tools to use these folders and files.

See Also

Related Examples

- “Choose an External Code Integration Workflow” on page 58-2
- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Control Data and Function Interface in Generated Code” on page 36-44
- “Configure Generated Code According to Interface Control Document Specifications” on page 40-54
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Specify Templates For Code Generation” on page 55-48

Configure Generated Code According to Interface Control Document Specifications

This example shows how to configure code generation settings for a model according to specifications in an interface control document (ICD). The example also shows how to store shared Simulink® variables and data objects in a data dictionary.

An ICD describes the data interface between software components. To exchange and share data, the components declare and define global variables that store signal and parameter values according to the ICD. The ICD names the variables and lists characteristics such as data type, physical units, and parameter values. When you create component models in Simulink, you can configure the models such that the generated code conforms to the ICD. For example, you can use the ICD as a reference source while configuring models interactively or you can automate configuraiton by using a script to import data from an ICD.

Explore Interface Control Document

In Microsoft® Excel® or another compatible program, open the `ex_ICD_PCG.xlsx` workbook and review the contents of the worksheets.

```
open ex_ICD_PCG.xlsx
```

- **`Signals` worksheet.** Each row represents a signal that crosses the interface boundary. Inspect the cell values in the worksheet. The `Owner` column identifies the name of the component that allocates memory for a signal. The `DataType` column names the signal data type in memory. For example, the worksheet uses the expression `Bus: EngSensors` to name the structure type `EngSensors`.
- **`Parameters` worksheet.** The `Value` column lists the value of each parameter. If the value of a parameter is nonscalar, the value is stored in its own separate worksheet, which has the same name as the parameter.
- **`Numeric Types` worksheet.** Each row represents a named numeric data type. In this ICD, the data uses fixed-point data types (Fixed-Point Designer). The `IsAlias` column indicates whether the C code uses the name of the data type (for example, `s16En3`) or uses the name of the primitive integer data type that corresponds to the word length (such as `short`). The `DataScope` column indicates whether the generated code exports or imports the data type definition.
- **`Structure Types` worksheet.** Each row represents a structure type or a field of a structure type. For structure types, the value in the `DataType` column is `struct`. Subsequent rows that do not use `struct` represent fields of the preceding structure type. This ICD defines structure type, `EngSensors`, with four fields: `throttle`, `speed`, `ego`, and `map`.
- **`Enumerated Types` worksheet.** Similar to the `Structure Types` worksheet, each row represents an enumerated type or an enumeration member. This ICD defines enumerated type `sld_FuelModes`.

Explore External Code

Some data items in the ICD belong to `other_component`, which is a component that exists outside of MATLAB®. Open and explore the content of these example files, which define and declare the external data:

```
edit ex_inter_types.h
edit ex_inter_sigs.c
edit ex_inter_sigs.h
```

Explore Example Model

In this example, you generate code from the controller model `rtwdemo_fuelsys_dd_controller`. Open your copy of the model.

```
open_system('rtwdemo_fuelsys_dd_controller')
```

Some signals in the controller model have names, for example, the input signal `sensors`. Some block parameters in the model refer to `Simulink.Parameter` objects in a data dictionary. For example, in the `airflow_calc` subsystem, the `Pumping Constant` block uses the parameter objects `PumpCon`, `SpeedVect`, and `PressVect`. These parameter objects set the values of the corresponding block parameters. You can apply code generation settings to the signals and parameter objects.

The controller model is linked to data dictionary rtw`demo_fuelsys_dd_controller.sldd`. Explore the data dictionary.

1. In the lower-left corner of the controller model, click the model data badge.
2. Click the **External Data** link.
3. In the Model Explorer **Model Hierarchy** pane, expand the nodes **rtwdemo_fuelsys_dd_controller**, **External data**, and **rtwdemo_fuelsys_dd_controller**.
4. Select **Design Data**.

The dictionary stores:

- Parameter objects
- `Simulink.NumericType` objects, such as `u8En7`
- `Simulink.Bus` object, `EngSensors`

Configure Model Design Data According to ICD

1. Open the ICD spreadsheet, if not open already.
2. In the Simulink Editor window for the controller model, navigate to the root level of the model.
3. On the **Modeling** tab, click **Model Data Editor**.
4. In the Model Data Editor, click the **Change scope** button. The Model Data Editor shows information about data items in the model subsystems.
5. Click the **Show/refresh additional information** button. The Model Data Editor shows information about data objects (the `Simulink.Parameter` objects in the data dictionary) that the model uses.

Configure Design Settings for Signals

1. In the ICD, click the **Signals** tab.
2. In the Model Data Editor, click the **Inports/Outports** tab.
3. Make sure that the configuration settings for Inport block `sensors` match the ICD specification. **Data Type** must be set to Bus: `EngSensors`.
4. Make sure that the configuration settings for Outport block `fuel_rate` match the ICD specification. **Data Type** must be set to `s16En7`. **Min** must be set to `0.8000`. **Max** must be set to `1.7000`. **Unit** must be set to `g/s`.

5. Make sure that the configuration settings for signal `fuel_mode` match the ICD specification. Because `fuel_mode` is an output signal of a Stateflow® chart, you must use Model Explorer to configure the design settings. In the model, navigate into the Stateflow chart. Open Model Explorer. In the Model Explorer **Contents** pane (the middle pane), select `fuel_mode`. **Data Type** must be set to `Enum:sld_FuelModes`.

Configure Design Settings for Parameters

1. In the ICD, click the **Parameters** tab.
2. Navigate to the root level of the control model.
3. In the Model Data Editor, click the **Parameters** tab.
4. Use the **Filter contents** box to search for parameter `PressEst`. The Model Data Editor shows two rows: one row that corresponds to parameter object `PressEst` and one row that corresponds to the block parameter that uses `PressEst`.
5. Make sure that the configuration settings for parameter `PressEst` match the ICD specification. **Data Type** must be `u8En7`. **Value** must be a 19x45 matrix of numeric values.
6. Make sure that the **Data Type** and **Value** settings for parameters `PumpCon`, `SpeedEst`, `ThrotEst`, and `RampRateKiz` match the ICD specification.

Configure Design Settings for Numeric Types

1. In the ICD, click the **Numeric Types** tab.
2. If the Model Explorer is not open, open it.
3. In the **Model Hierarchy** pane, navigate to the design data in the data dictionary for the controller model (**rtwdemo_fuelsys_dd_controller> External Data>rtwdemo_fuelsys_dd_controller>Design Data**).
4. In the **Contents** (middle) pane, select the `Simulink.NumericType` object `u8En7`. This object represents one of the `typedef` statements in `ex_inter_types.h`.
5. In the right pane, on the **Design** tab, confirm that the setting for **Data type mode** aligns with the ICD specification.
6. Make sure that the design configuration for numeric types `s16En3`, `s16En7`, and `s16En15` matches the ICD specification.

Configure Design Settings for Structures

1. In the ICD, click the **Structure Types** tab.
2. In the Model Explorer **Contents** pane, select `EngSensors`. This `Simulink.Bus` object represents the structure type defined in `ex_inter_types.h`.
3. Confirm that the data type configuration aligns with the ICD specification. In the right pane, on the **Design** tab, or in the Type Editor, adjust the bus signal configurations (for example, settings for **Min** and **Max**) to align with the ICD specification.

Configure Design Settings for Enumerated Types

1. In the ICD, click the **Enumerated Types** tab.
2. Open the file `sld_FuelModes.m`, which defines the class for the enumerated type `sld_FuelModes`.
3. Confirm that the class definitnion aligns with the ICD specification. In the file sld_FuelModes.m, adjust member names and underlying integer values to align with the ICD specification.

Configure Model for Code Generation According to ICD

1. Open the ICD spreadsheet, if not open already.
2. Open the Embedded Coder app.

Configure Code Generation Settings for Inport and Outport Blocks

1. In the ICD, click the **Signals** tab. The Owner column in the ICD implies that a different component (`other_component`), not `rtwdemo_fuelsys_dd_controller`, provides the code definition of the `sensors` variable.
2. Include the C source code file that defines the variable `sensors`. In the Simulink Editor, on the C Code tab, select **Settings**> **C/C++ Code generation settings**. Make sure that the code generation custom code parameter **Source files** is set to `ex_inter_sigs.c`.
3. In the Simulink Editor, on the **C Code** tab, select **Code Interface**> **Individual Element Code Mappings**.
4. In the Code Mappings editor, click the **Inports** tab.
5. Select the row for `sensors`. Because the source code definition is provided in the external file `ex_inter_sigs.c`, which you configured in step 2, **Storage class** must be set to `ImportFromFile`.
6. Click the icon. Check that the **HeaderFile** property setting matches the ICD specification, `ex_inter_sigs.h`. Make sure that a value is specified for the **Identifier** property. This property specifes the variable name for the data element in the generated code.
7. In the Code Mappings editor, click the **Outports** tab.
8. Select the row for `fuel_rate`. Because the ICD specifies `rtwdemo_fuelsys_dd_controller` as the owner of `fuel_rate`, **Storage Class** must be set to `ExportToFile`.
9. Click the icon. Make sure that the settings for properties **HeaderFile, DefinitionFile**, and **Owner** match the ICD specifications `global_data.h`, `signals.c`, and `rtwdemo_fuelsys_dd_controller`, respectively. Make sure that a value is specified for the **Identifier** property.

Configure Code Generation Settings for Signals

1. In the Code Mappings editor, click the **Signals/States** tab. The third signal listed on the **Signals** tab of the ICD is `fuel_mode`.
2. Add the `fuel_mode` signal to the model code mappings. In the model diagram, select signal `fuel_mode`. Place your cursor on the ellipsis that appears above or below the signal line to open the action bar. Click the **Add selected signals to code mappings** button. In the Code Mappings editor, the **Signals** node expands and lists the signal that you added. The Storage Class setting must indicate that the signal resolves to a signal object and that the storage class for the object is set to `ExportToFile`. Because the signal is associated with a signal object, you must use the Model Explorer to confirm that the code generation configuration aligns with the ICD specification.
3. Open the Model Explorer. In the Model Hierarchy pane, navigate to the design data in the data dictionary for the controller model (**rtwdemo_fuelsys_dd_controller**> **External Data**> **rtwdemo_fuelsys_dd_controller**> **Design Data**). In the **Contents** pane, select `fuel_mode`.
4. In the right pane, click the **Code Generation** tab. **Storage class** must be set to `ExportToFile`. Confirm that properties **HeaderFile**, **DefintionFile**, and **Owner** match the ICD specifications `global_data.h`, `signals.c`, and `rtwdemo_fuelsys_dd_controller`, respectively.

Configure Code Generation for Parameters

1. In the ICD, click the **Parameters** tab.
2. In the Code Mappings editor, click the **Paramters** tab.
3. Use the **Filter contents** box to search for parameter `PressEst`. The Code Mappings editor shows a row that corresponds to parameter object `PressEst`. Because the ICD specifies `rtwdemo_fuelsys_dd_controller` as the owner of `PressEst`, **Storage Class** should be set to `ExportToFile`.
4. Click the icon. Check that the settings for properties **HeaderFile, DefinitionFile**, and **Owner** match the ICD specifications `global_data.h`, `params.c`, and `rtwdemo_fuelsys_dd_controller`, respectively. Also, make sure that a value is specified for the **Identifier** property.
5. Check the **Storage Class**, **HeaderFile**, **DefinitionFile**, **Identifier**, and **Owner** settings for parameters `PumpCon`, `SpeedEst`, `ThrotEst`, and `RampRateKiz`.

Configure Code Generation for Numeric Types

1. In the ICD, click the **Numeric Types** tab.
2. If the Model Explorer is not open, open it.
3. In the **Model Hierarchy** pane, navigate to the design data in the data dictionary for the controller model (**rtwdemo_fuelsys_dd_controller> External Data> rtwdemo_fuelsys_dd_controller> Design Data**).
4. In the **Contents** pane, select the `Simulink.NumericType` object `u8En7`. This object represents one of the `typedef` statements in `ex_inter_types.h`.
5. In the right pane, click the **Code Generation** tab. Confirm that the settings for **Data scope** and **Header file** align with the ICD specifications, `Imported` and `ex_inter_types.h`, respectively.
6. Check and adjust the code generation configuration for numeric types `s16En3`, `s16En7`, and `s16En15`.

Configure Code Generation for Structures

1. In the ICD, click the **Structure Types** tab.
2. In the Model Explorer **Contents** pane, select `EngSensors`. This `Simulink.Bus` object represents the structure type defined in `ex_inter_types.h`.
3. In the right pane, click the **Code Generation** tab. Confirm that the settings for **Data scope** and **Header file** align with the ICD specifications, `Imported` and `ex_inter_types.h`, respectively.

Generate and Inspect Code

1. Generate code for the controller model.

```
slbuild('rtwdemo_fuelsys_dd_controller')
```

```
### Starting build procedure for: rtwdemo_fuelsys_dd_controller
### Successful completion of build procedure for: rtwdemo_fuelsys_dd_controller

Build Summary

Top model targets built:

Model                          Action                       Rebuild Reason
```

```
==========================================================================================
rtwdemo_fuelsys_dd_controller  Code generated and compiled  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 3m 1.371s
```

2. Inspect the generated code. The generated header file rtwdemo_fuelsys_dd_controller_types.h:

- Includes (#include) the external header file ex_inter_types.h, which defines numeric data types u8En7, s16En3, and s16En7 and the structure type EngSensors
- Defines the enumeration sld_FuelModes

```
file = fullfile('rtwdemo_fuelsys_dd_controller_ert_rtw',...
    'rtwdemo_fuelsys_dd_controller_types.h');
rtwdemodbtype(file,'#include "ex_inter_types.h"','} sld_FuelModes;',1,1)
```

```
#include "ex_inter_types.h"

/* Model Code Variants */
#ifndef DEFINED_TYPEDEF_FOR_sld_FuelModes_
#define DEFINED_TYPEDEF_FOR_sld_FuelModes_

typedef enum {
  LOW = 1,                              /* Default value */
  RICH,
  DISABLED
} sld_FuelModes;
```

The file rtwdemo_fuelsys_dd_controller_private.h includes the header file ex_inter_sigs.h. This external header file contains the extern declaration of the signal sensors, which a different software component owns.

The data header file global_data.h declares the exported parameters and signals that the ICD specifies. To share this data, other components can include this header file.

```
file = fullfile('rtwdemo_fuelsys_dd_controller_ert_rtw','global_data.h');
rtwdemodbtype(file,'/* Exported data declaration */',...
    'fuel_rate',1,1)
```

```
/* Exported data declaration */

/* Declaration for custom storage class: ExportToFile */
extern real32_T PressEst[855];   /* Referenced by: '<S6>/Pressure Estimation' */
extern real32_T PumpCon[551];       /* Referenced by: '<S1>/Pumping Constant' */
extern real32_T RampRateKiz[25];       /* Referenced by: '<S1>/Ramp Rate Ki' */
extern real32_T SpeedEst[1305];     /* Referenced by: '<S7>/Speed Estimation' */
extern real32_T ThrotEst[551];   /* Referenced by: '<S8>/Throttle Estimation' */
extern sld_FuelModes fuel_mode;        /* '<Root>/control_logic' */
extern real32_T fuel_rate;             /* '<Root>/fuel_rate' */
```

The algorithm code is in the model step function in the file rtwdemo_fuelsys_dd_controller.c. The algorithm uses the global data that the ICD identifies. For example, the algorithm uses the value of the signal fuel_mode in a switch block to control the flow of execution.

```
file = fullfile('rtwdemo_fuelsys_dd_controller_ert_rtw',...
    'rtwdemo_fuelsys_dd_controller.c');
```

```
rtwdemodbtype(file,'/* End of MultiPortSwitch: ''<S9>/Multiport Switch'' */',...
    '/* Outport: ''<Root>/fuel_rate'' incorporates:',0,0)
```

```
  /* SwitchCase: '<S10>/Switch Case' */
  switch (fuel_mode) {
   case LOW:
    /* Outputs for IfAction SubSystem: '<S10>/low_mode' incorporates:
     *  ActionPort: '<S12>/Action Port'
     */
    /* DiscreteFilter: '<S12>/Discrete Filter' incorporates:
     *  DiscreteIntegrator: '<S1>/Discrete Integrator'
     */
    denAccum = rtDWork.DiscreteIntegrator_DSTATE - -0.7408F *
      rtDWork.DiscreteFilter_states_g;
```

```
bdclose('rtwdemo_fuelsys_dd_controller')
Simulink.data.dictionary.closeAll('rtwdemo_fuelsys_dd_controller.sldd','-discard')
```

Import ICD Specifications into Simulink

You can use a MATLAB script to import data settings from the ICD into variables in the Simulink base workspace.

1. Open and inspect the example script `ex_importICD_PCG.m`.

```
edit ex_importICD_PCG
```

The script:

- Imports the data from each worksheet of the ICD into variables in the base workspace.
- Uses the imported data to configure design and code generation properties of `Simulink.Signal` and `Simulink.Parameter` objects in the base workspace.

If the base workspace already contains a data object that corresponds to a data element in the ICD, the script configures properties of the existing object. If the object does not exist, the script creates the object.

2. Load the model and run the script.

```
load_system('rtwdemo_fuelsys_dd_controller')
run('ex_importICD_PCG')
```

3. Regenerate and inspect the code.

```
slbuild('rtwdemo_fuelsys_dd_controller')
```

```
### Starting build procedure for: rtwdemo_fuelsys_dd_controller
### Successful completion of build procedure for: rtwdemo_fuelsys_dd_controller

Build Summary

Top model targets built:

Model                          Action                        Rebuild Reason
========================================================================================
rtwdemo_fuelsys_dd_controller  Code generated and compiled  Generated code was out of date.
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 36.732s
```

Considerations

- When you make changes to an ICD, you can reuse your import script to reconfigure the model.
- If you use a script to import data from an ICD to the base workspace, consider migrating the data objects and variables to a data dictionary. A data dictionary permanently stores and tracks changes to data objects and variables. For example, you can import the definition of enumerated type `sld_FuelModes` into the controller model dictionary. See “Import and Export Dictionary Data” and “Enumerations in Data Dictionary”.
- You can store configuration settings for signal and state data elements, such as data types, minimum and maximum values, and physical units inside or outside of model files. `Simulink.Signal` objects store the configuration settings outside of a model file. Alternatively, you can store the configuration settings in the model file by using block and port parameters that you can access through the Model Data Editor, Property Inspector, and other dialog boxes. See “Store Design Attributes of Signals and States”.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Replace and Rename Simulink Coder Data Types to Conform to Coding Standards” on page 38-24
- “Control Data and Function Interface in Generated Code” on page 36-44
- “Data Import and Export”
- “What Is a Data Dictionary?”
- “Integrate External Application Code with Code Generated from PID Controller” on page 40-45

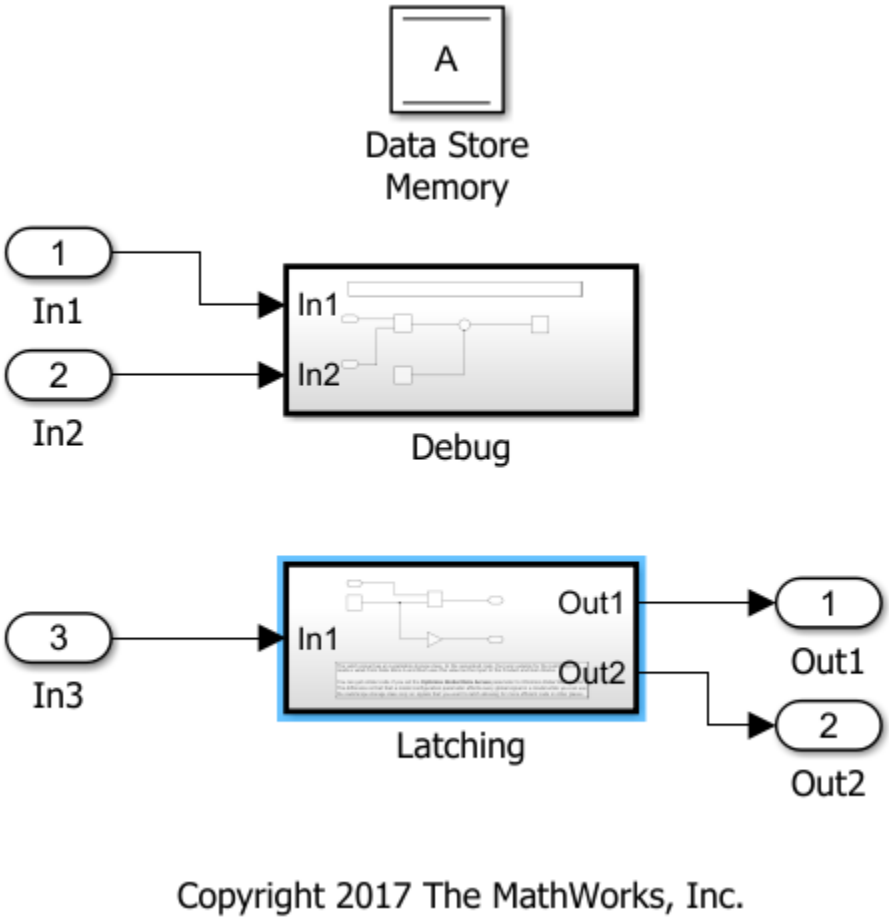
Generate Local Variables with Localizable Storage Class

For signals, if possible, generate variables that are local to functions rather than in global storage. Generating local variables prevents the code generator from implementing optimizations that remove these variables from the generated code. Local variables improve observability, readability, and are helpful in debugging the generated code.

Minimizing the use of global variables by using local variables interacts with stack usage control. For example, stack size can determine the number of local and global variables that the code generator can allocate in the generated code. For more information, see “Customize Stack Space Allocation”.

Example Model

The model `rtwdemo_localizable_csc` contains two signals that have a `Localizable` storage class. In the `Latching` subsystem, the signal with the label `Latch` has a `Localizable` storage class. In the `Debug` subsystem, the signal with the label `Debug` has a `Localizable` storage class.



Generate Code with Localizable Storage Class

1 Open the model.

```
model='rtwdemo_localizable_csc';
open_system(model);
```

2. To observe the specification, open the Code Mappings editor. On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.
3. On the **Signals/States** tab, expand **Signals**. The storage class for signals **Latch** and **Debug** is set to `Localizable`.
4. Generate and view the code. The `Debug_b` function contains this code:

```
static void Debug_b(const real_T rtu_In1[6], const real_T rtu_In2[6], real_T
                    rtd_A[6])
{
  real_T Debug;
  int32_T i;
  for (i = 0; i < 6; i++) {
    Debug = rtu_In1[i] * rtu_In2[i];
    rtd_A[i] += Debug;
  }
}
```

The `Latching` function contains this code:

```
static void Latching(const real_T rtu_In1[36], real_T rty_Out1[6], real_T
                     rty_Out2[6], real_T rtd_A[6])
{
  real_T Latch[6];
  int32_T i;
  int32_T i_0;
  for (i = 0; i < 6; i++) {
    Latch[i] = rtd_A[i];
    rty_Out2[i] = -Latch[i];
  }

  for (i = 0; i < 6; i++) {
    rty_Out1[i] = 0.0;
    for (i_0 = 0; i_0 < 6; i_0++) {
      rty_Out1[i] += rtu_In1[6 * i_0 + i] * Latch[i_0];
    }
  }
}
```

Both functions contain variables for holding intermediate values. The `Debug_b` function contains the variable `Debug`. The `Latching` function contains the variable `Latch`.

Generate Code Without Localizable Storage Class

1. Change the signal storage class from `Localizable` to `ExportToFile`. In the Code Mappings editor, for each signal, set the storage class to `ExportToFile`.
2. Generate and view the code. The `rtwdemo_localizable_csc.c` file contains these two global variable declarations:

```
real_T Debug[6];
real_T Latch[6];
```

The `Debug_b` function contains this code:

```
static void Debug_b(const real_T rtu_In1[6], const real_T rtu_In2[6], real_T
                    rtd_A[6])
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    Debug[i] = rtu_In1[i] * rtu_In2[i];
    rtd_A[i] += Debug[i];
  }
}
```

The `Latching` function contains this code:

```
static void Latching(const real_T rtu_In1[36], real_T rty_Out1[6], real_T
                     rty_Out2[6], real_T rtd_A[6])
{
  int32_T i;
  int32_T i_0;
  for (i = 0; i < 6; i++) {
    Latch[i] = rtd_A[i];
    rty_Out2[i] = -Latch[i];
  }

  for (i = 0; i < 6; i++) {
    rty_Out1[i] = 0.0;
    for (i_0 = 0; i_0 < 6; i_0++) {
      rty_Out1[i] += rtu_In1[6 * i_0 + i] * Latch[i_0];
    }
  }
}
```

The code readability and observability is the same as with the `Localizable` storage class specification except the labeled signals are global variables.

3. Remove the `Debug` and `Latch` signals from the code mappings. In the Code Mappings editor, select each signal and click the **Remove Signal** button.
4. Save the model.
5. Generate and review the code. The `Debug_b` function contains this code:

```
static void Debug(const real_T rtu_In1[6], const real_T rtu_In2[6], real_T
                  rtd_A[6])
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    rtd_A[i] += rtu_In1[i] * rtu_In2[i];
  }
}
```

The `Latching` function contains this code:

```
static void Latching(const real_T rtu_In1[36], real_T rty_Out1[6], real_T
                     rty_Out2[6], real_T rtd_A[6])
{
  int32_T i;
  int32_T i_0;
  for (i = 0; i < 6; i++) {
    rty_Out2[i] = -rtd_A[i];
    rty_Out1[i] = 0.0;
    for (i_0 = 0; i_0 < 6; i_0++) {
      rty_Out1[i] += rtu_In1[6 * i_0 + i] * rtd_A[i_0];
    }
  }
}
```

Without the `Localizable` or `ExportToFile` storage classes, the code generator removes the `Debug` and `Latch` variables. Without these variables, the readability and observability of the generated code decreases.

Additional Information

- For the code generator to localize signals with the `Localizable` storage class specification, select model configuration parameter **Enable Local Block Outputs**. This parameter is on by default.
- You can specify the same `Localizable` storage class on two signals that are in different reusable subsystems. Therefore, you can create a library block from a reusable subsystem with a `Localizable` storage class specification.
- The code generator does not create a local variable for a signal that must stay global. A few cases in which the code generator cannot localize a signal are:

- Input and output signals of a nonreusable subsystem even when the **Function interface** parameter is set to `Allow arguments (Optimized)`.
- Signal is holding a state because its receiver and drive blocks execute at different rates.
- `Localizable` specification is on a signal that crosses the boundary of a conditionally executed subsystem.

See Also

More About

- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75

Data Object Wizard in Embedded Coder

Create Data Objects for Code Generation with Data Object Wizard

To specify code generation options for signal lines, block parameters, and states in a model, you can use data objects that you store in a workspace or data dictionary. For basic information about data objects, see “Data Objects”.

You can use the Data Object Wizard to create data objects for:

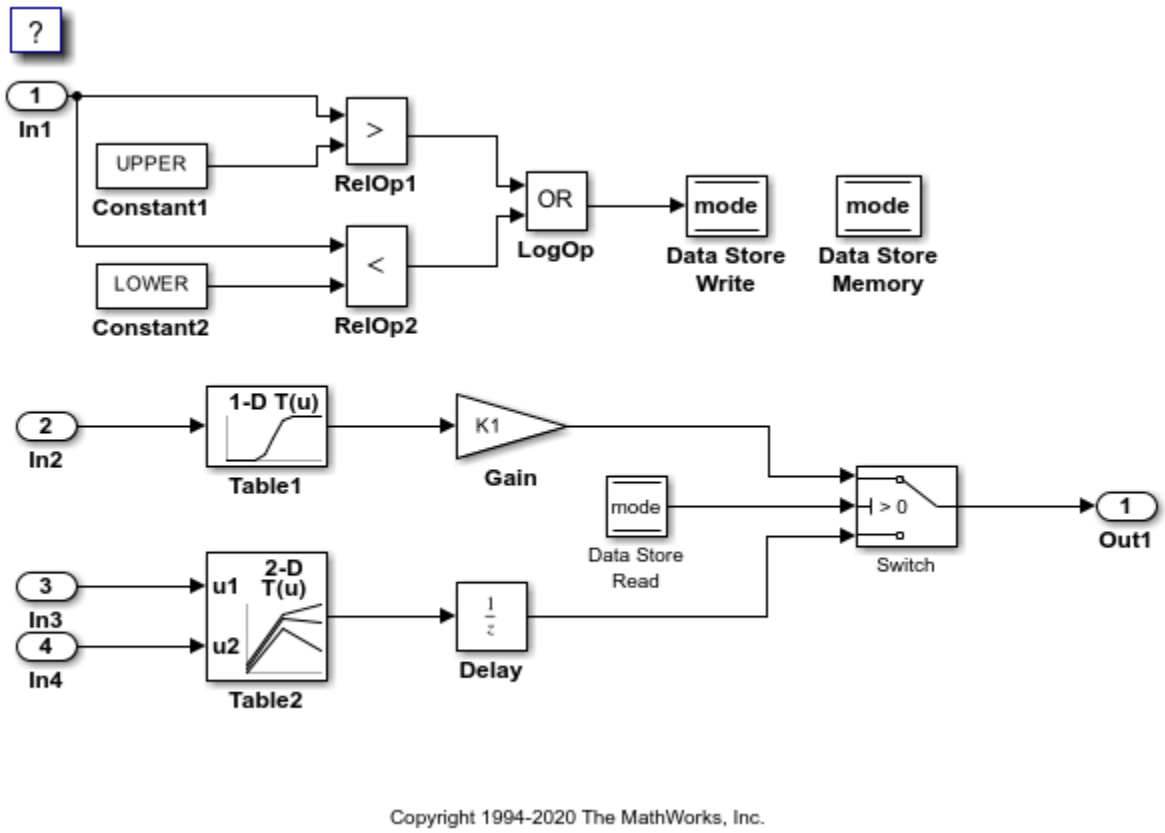
- New or existing models that do not use data objects.
- Existing models to which you have added signal lines or blocks.

This example shows how to use the Data Object Wizard to create and configure data objects for code generation from the built-in package `Simulink`.

Create Data Objects

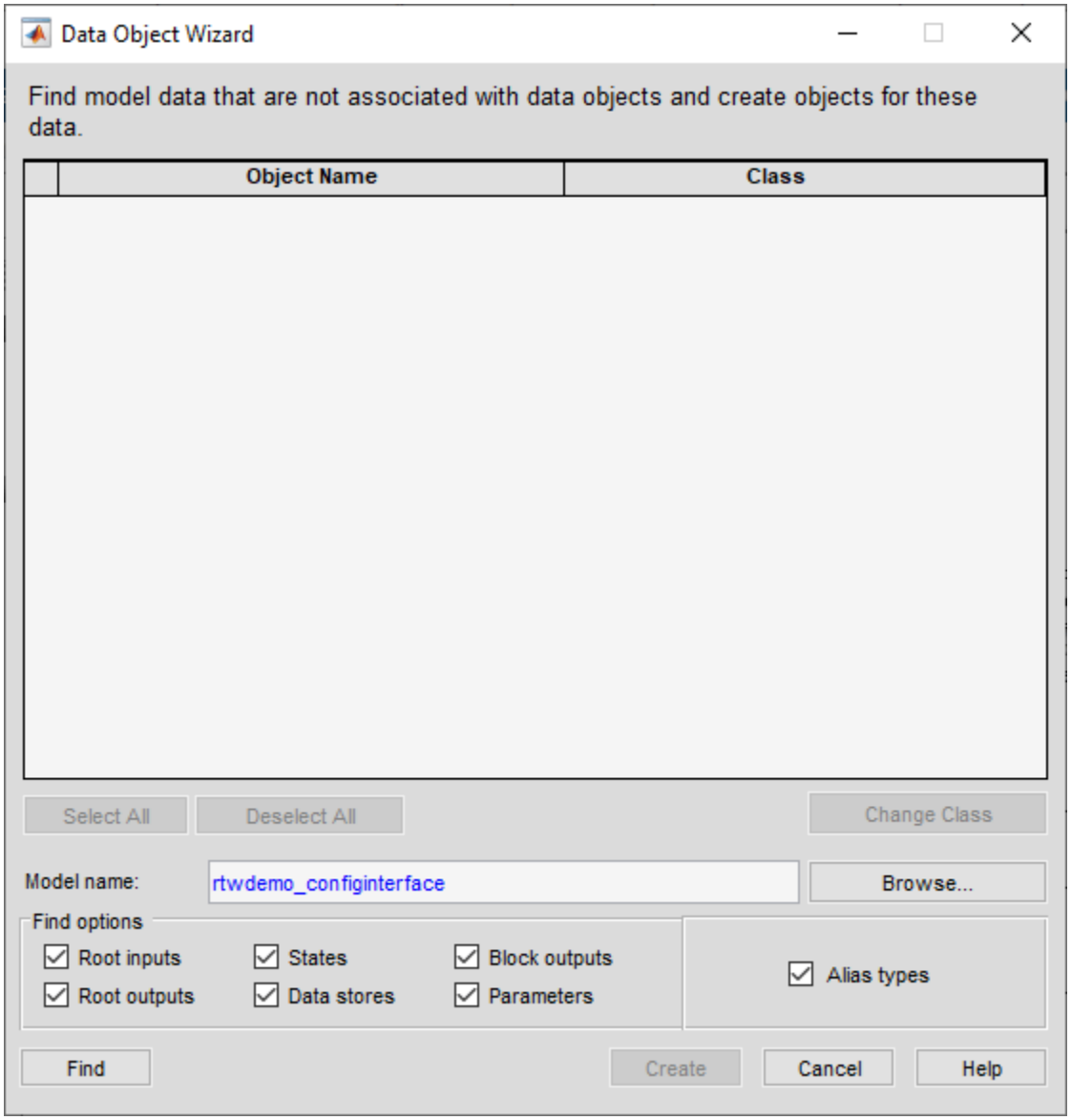
Open the example model `rtwdemo_configinterface`.

```
open_system('rtwdemo_configinterface')
```

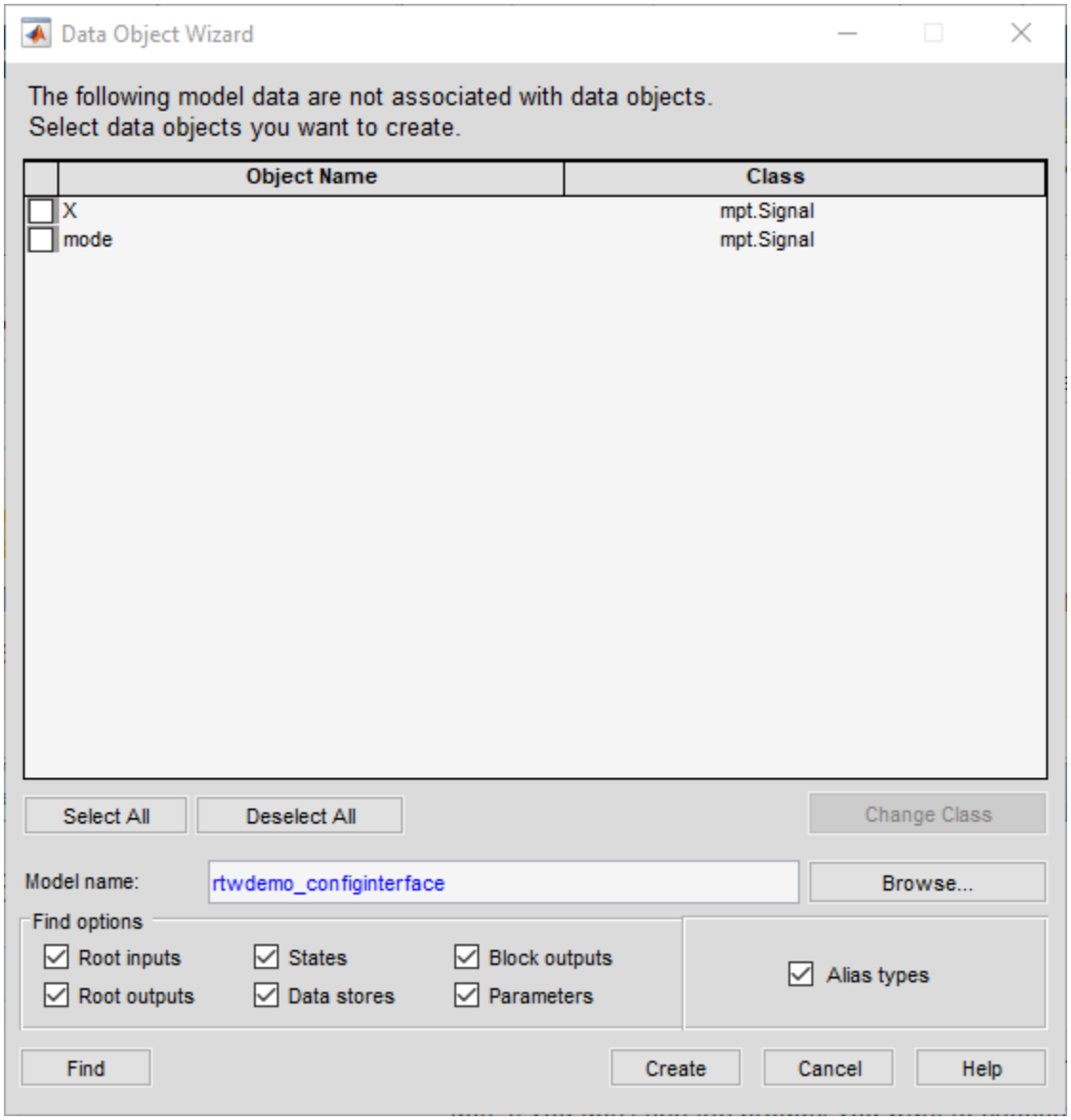


The model creates numeric variables in the base workspace. Blocks in the model use these variables to set parameter values (such as the **Gain** parameter of a Gain block). Some of the signals and block states in the model have explicit names, such as `input1`.

On the **Modeling** tab, under **Design**, click **Data Object Wizard**.



In the Data Object Wizard, click **Find**. The wizard proposes the creation of `Simulink.Parameter` objects to replace the variables and the creation of `Simulink.Signal` objects to represent the signals and states.



The wizard finds only signals, parameters, data stores, and states whose storage class is set to `Auto`. For example, if you use the Signal Properties dialog box to specify a storage class other than `Auto` for a signal line, the wizard does not propose a data object.

Click **Select All**.

Click **Create**. The data objects appear in the base workspace.

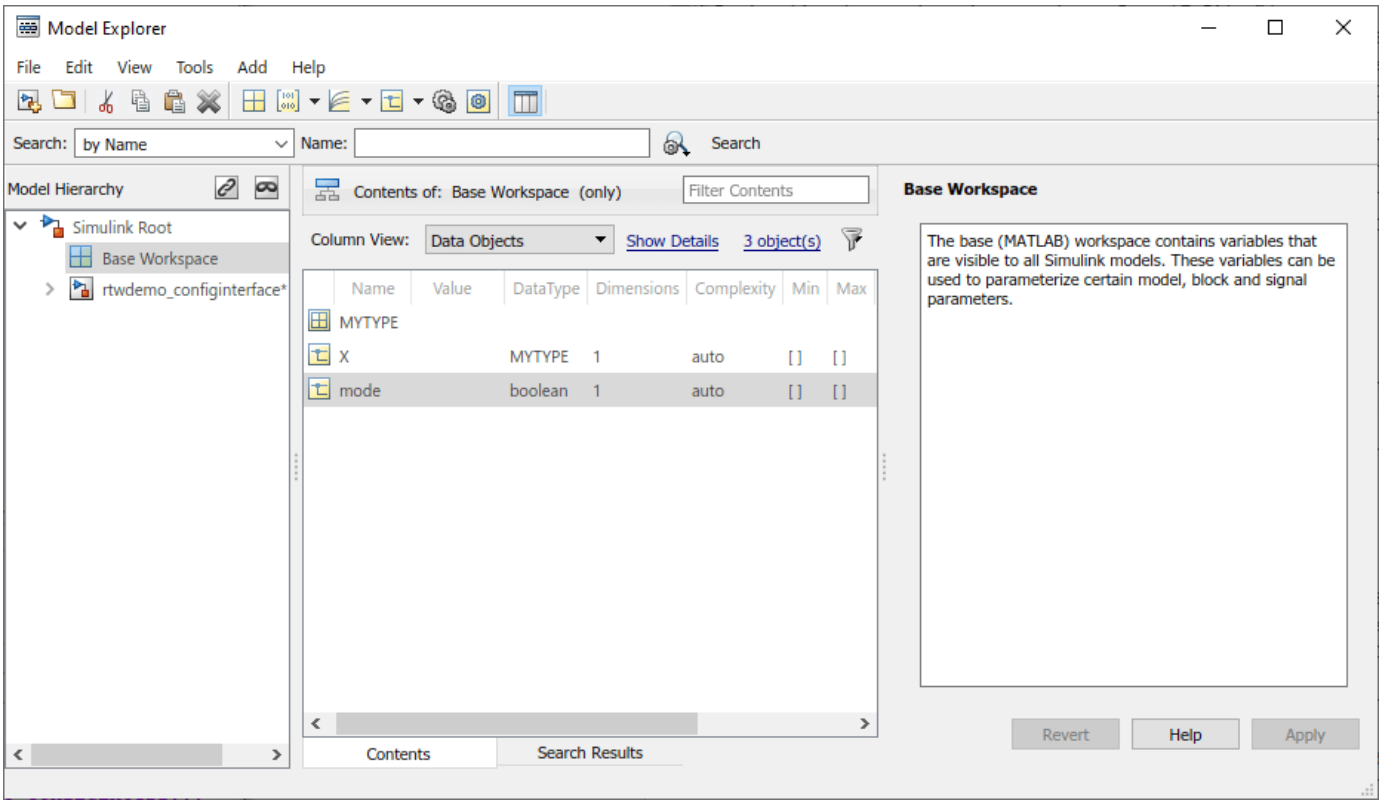
For detailed information about the options that you can choose in the Data Object Wizard, see "Create Data Objects for a Model Using Data Object Wizard".

Set Storage Class for Data Objects

Storage classes determine how the generated code uses variables to represent signals, parameters, and states. For data objects from the built-in package `Simulink`, the default storage class is `Auto`. To specify storage classes for the new data objects, use the Model Explorer.

Open the Model Explorer.

In the **Model Hierarchy** pane, select **Base Workspace**.



In the **Contents** pane, from the drop-down list **Column View**, select `Storage Class`.

Select all of the new data objects. For example, select the object `input1`, hold **Shift**, and select the object `X`.

Set the property `StorageClass` for all of the data objects to `ExportToFile`. To change the storage class for all of the selected objects, in the **StorageClass** column, click any of the objects. In the drop-down list, select `ExportToFile`. The change that you make propagates to all of the selected objects.

Specify the `HeaderFile` property for all of the objects as `myExportedHdrFile.h`.

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors storage classes such as `ExportToFile`.

Generate and Inspect Code

Generate code from the model.

```
### Starting build procedure for: rtwdemo_configinterface
### Successful completion of code generation for: rtwdemo_configinterface

Build Summary

Top model targets built:
```

```
Model                    Action           Rebuild Reason
========================================================================================
rtwdemo_configinterface  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.59s
```

In the code generation report, view the generated file `myExportedHdrFile.h`. The file contains `extern` declarations for the global variables that correspond to the data objects.

```
/* Exported data declaration */
/* Declaration for custom storage class: ExportToFile */
extern real_T X;
```

View the file `rtwdemo_configinterface.c`. The file contains the definitions for the global variables. The code assigns numeric values for the variables that correspond to parameter objects.

```
/* Exported data definition */

/* Definition for custom storage class: ExportToFile */
real_T X;
```

See Also

`Simulink.Parameter` | `Simulink.Signal`

Related Examples

- “Data Objects”
- “C Code Generation Configuration for Model Interface Elements”
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67

Entry-Point Functions and Scheduling in Simulink Coder

- “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2
- “Generate C++ Class Interface to Model or Subsystem Code” on page 42-15
- “Execution of Code Generated from a Model” on page 42-18
- “Rapid Prototyping Model Functions” on page 42-27

Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks

With Embedded Coder, you can customize the generated C and C++ function interfaces for Simulink Function and Function Caller blocks. Function code interface configuration supports easier integration of generated code with functions or function calls in external code and customizations for coding standards or design requirements.

You can customize generated C and C++ function interfaces for:

- Global Simulink functions
- Exported scoped Simulink functions

You cannot customize the generated function interface for a scoped Simulink Function block that is not located at the root level of a model. For more information, see “Scoped, Global, and Port-Scoped Simulink Function Blocks Overview”.

By opening a dialog box from a selected Simulink Function or Function Caller block, you can customize the function prototype generated for that block. Your changes for the selected block also update other corresponding Simulink Function and Function Caller blocks in the model. The function visibility, global or scoped, set in the function Trigger Port block determines which function attributes you can modify:

- For a global function, you can change the function name, and the names, type qualifiers, and order of function arguments.
- For an exported scoped function, you can change the argument type qualifiers and the order of arguments. You cannot change the function and argument names.

Your changes do not graphically alter the model and do not affect the Simulink function prototype defined in the block.

Embedded Coder supports Simulink function code interface configuration for models configured with ERT-based system target files, except for AUTOSAR system target files.

Configure Generated C Function Interface for Global Simulink Function Block

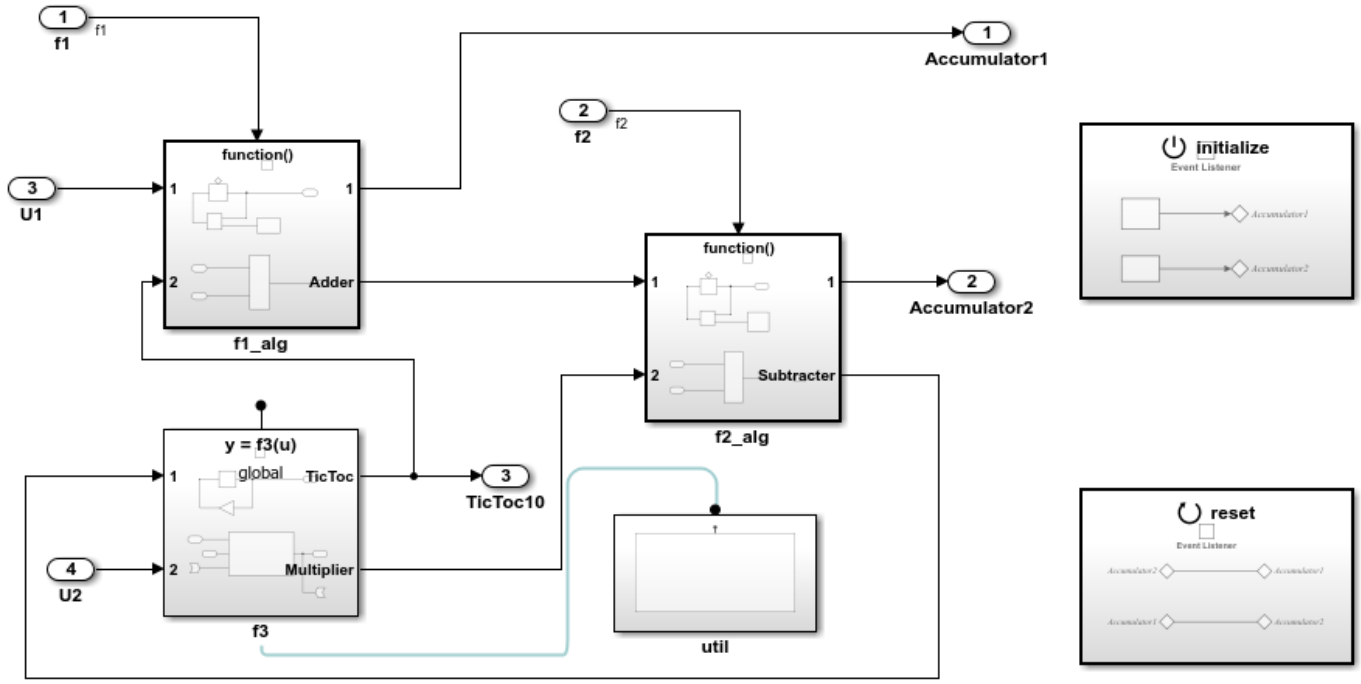
This example shows how to customize the C function interface that the code generator produces for a global Simulink Function block.

1. Open the example model.
2. Open the Configure C/C++ Function Interface Dialog Box.
3. Configure the Simulink function prototype.
4. Generate and review the code.

Open Example Model

Open example model `SimulinkFunctions`. Save a copy of the model to a writable location.

```
open_system('SimulinkFunctions');
```



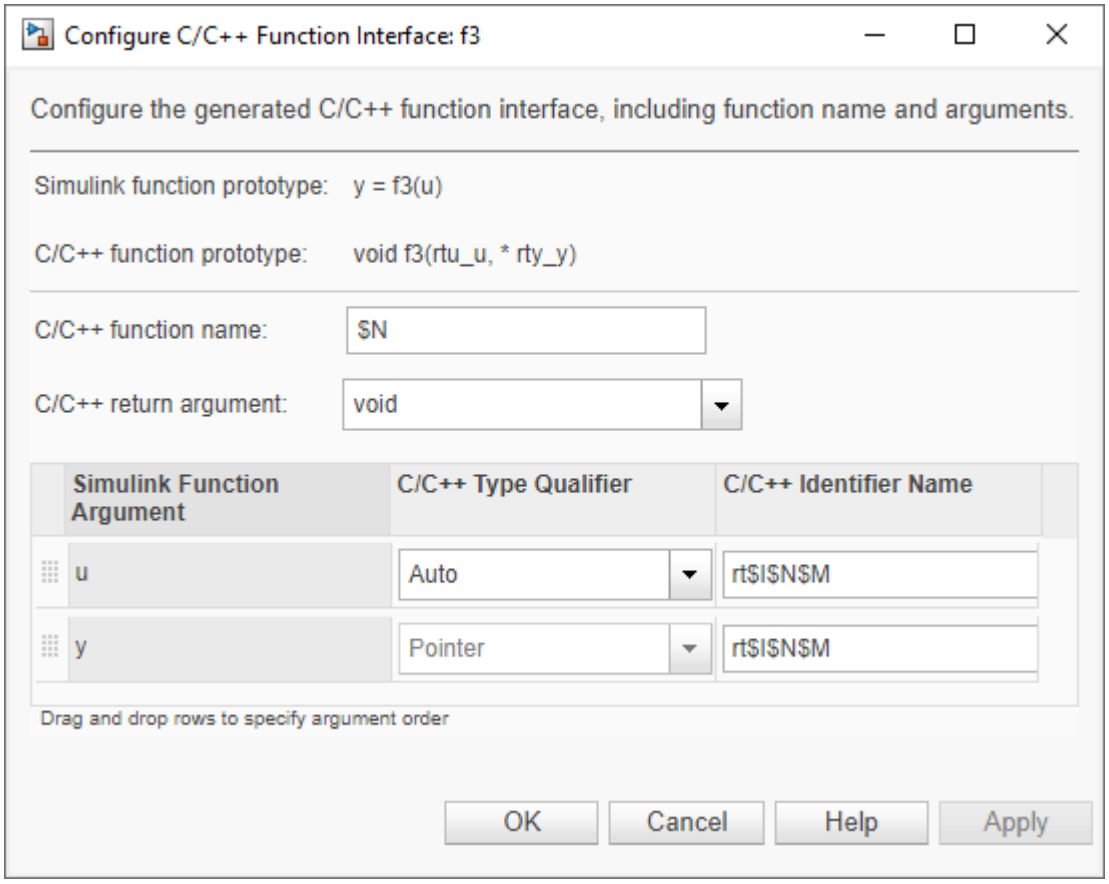
Open Configure C/C++ Function Interface Dialog Box

In the apps gallery, open the **Embedded Coder** app.

On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

In the Code Mappings editor, click the **Functions** tab.

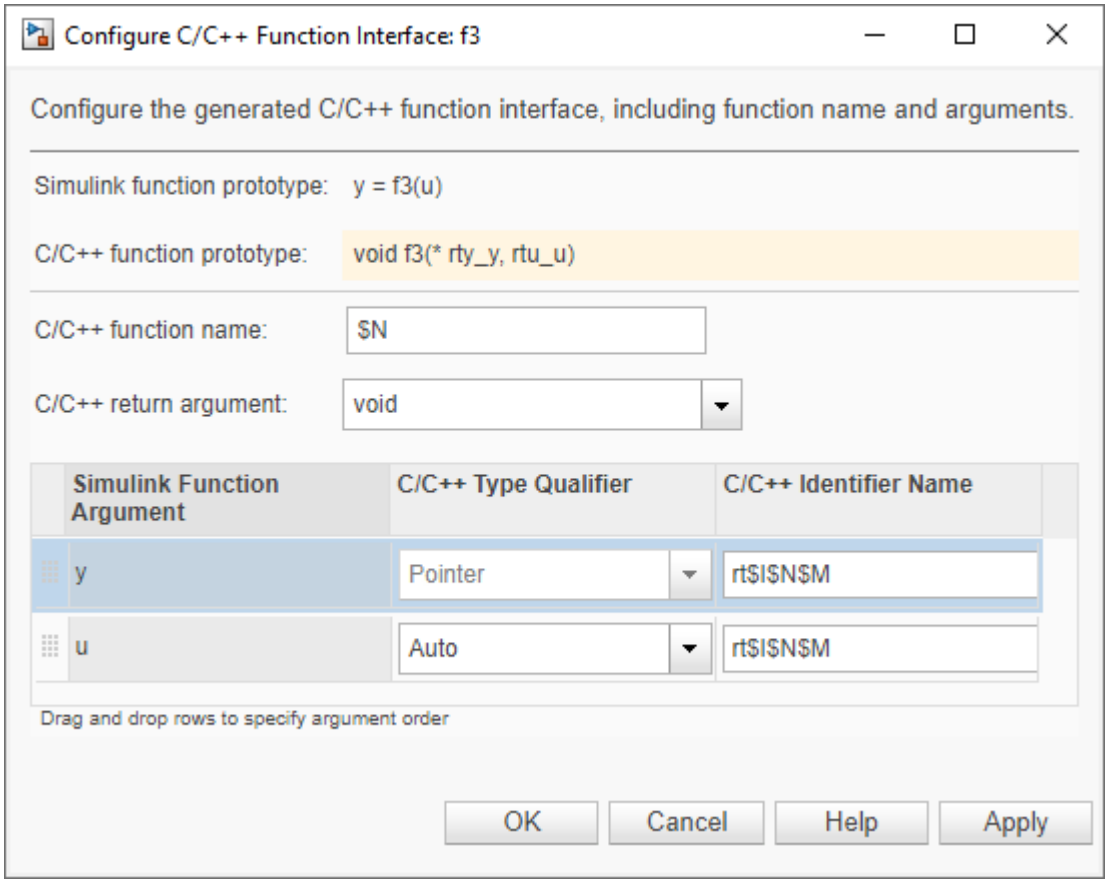
In the row for `Simulink Function:f3`, click the function preview hyperlink. The Configure C/C++ Function Interface dialog box shows **Simulink function prototype** as `y = f3(u)` and **C/C++ function prototype** as `y = SimulinkFunctions_f3(rtu_u)`. The preview for **C/C++ function prototype** is updated as you make changes.



Configure Simulink Function Prototype

Configure the function name and arguments.

- Modify the function and argument identifier names. Model configuration parameter **Subsystem method arguments** defines the default naming rule for Simulink Function arguments. Changes that you make in the C/C++ Function Interface dialog box override the default naming rule. In the **C/C++ function name field**, and in the **C/C++ Identifier Name** column for each Simulink function argument, enter a custom name or naming rule. Specify valid C-identifier characters, identifier format macros, or a combination of characters and macros. For this example, name the function `function3` and, for the arguments, enter naming rule `$N$M`. To see tips about available macros, place your cursor over **C/C++ function name** and **C/C++ Identifier Name**. For more information about identifier naming rules, see “Identifier Format Control” on page 55-17.
- For argument u, set **C/C++ Type Qualifier** to `Pointer to const`.
- Reorder the arguments. Drag the y argument row above the u argument row.



Verify your changes by checking the function prototype preview.

Click **Apply**. Your modifications, whether made to a Simulink Function block or a Function Caller block, affect code generation for the Simulink Function block and corresponding Function Caller blocks in the model.

Optionally, you can change **C/C++ return argument** from `void` to `y`. In that case, the C function prototype is `y = function3(const * u)`.

Save your changes by clicking **OK**. Then, save the model.

Generate and Review Code

Make sure that the Simulink Function block is not selected. Then, generate code for the model.

In the **Code** view, open the generated file SimulinkFunctions.c and search for `function3`. The generated function code reflects the changes to the generated C function prototype.

```
void function3(real_T *y, const real_T *u)
{
  rtY.TicToc10 = rtDWork.Delay_DSTATE;

  rtDWork.Delay_DSTATE = (int8_T)(int32_T)-(int32_T)rtY.TicToc10;

  adder(rtB.Subtract, rtU.U2, *u, &rtB.FunctionCaller);
```

```
  *y = rtB.FunctionCaller;
}
```

More About

- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Simulink Functions Overview”
- “Scoped Simulink Function Blocks in Models”
- “Simulink Function Blocks in Referenced Models”

Configure Generated C Function Interface for Exported Scoped Simulink Function Block

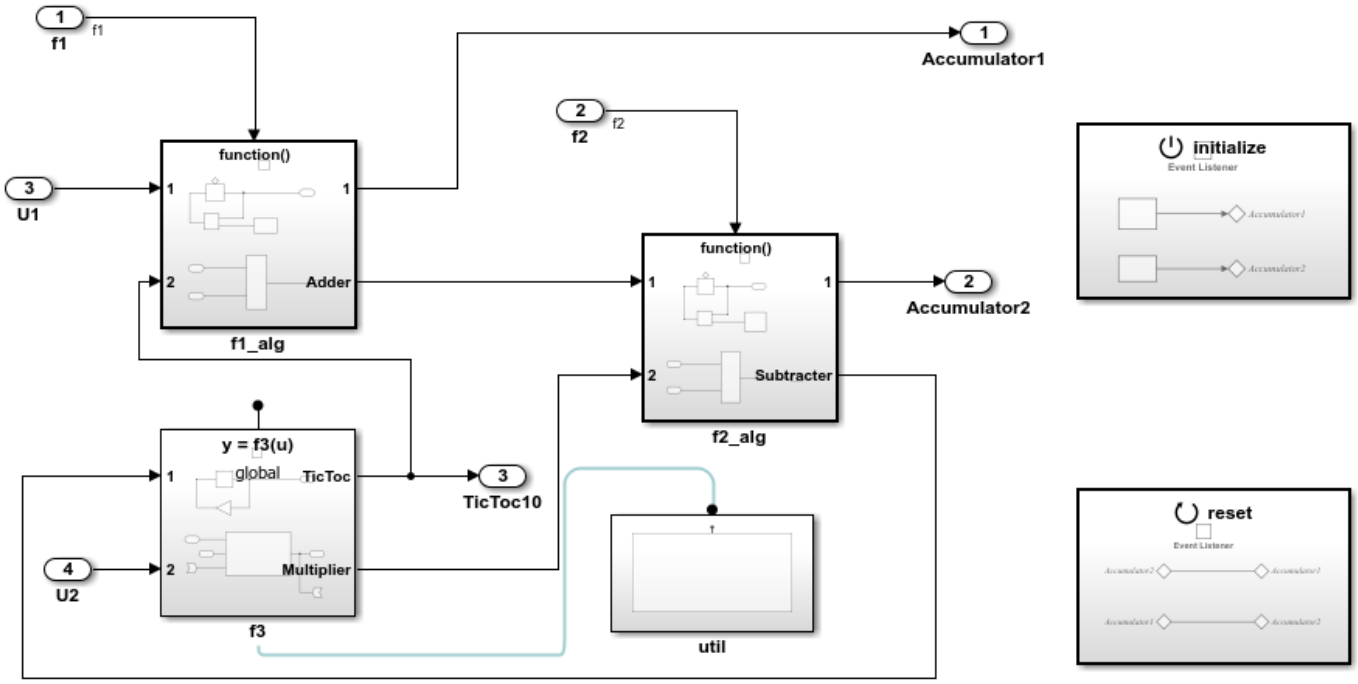
This example shows how to customize the C function interface that the code generator produces for an exported, scoped Simulink Function block.

For an exported, scoped function, you can modify the generated return argument, the argument type qualifiers, and the order of arguments. You cannot change the generated function name and argument names.

1. Open the example model.
2. Change the visibility of the Simulink function.
3. Open the Configure C/C++ Function Interface dialog box.
4. Configure the Simulink Function prototype.
5. Generate and review the code.

Open Example Model

Open example model `SimulinkFunctions`. Save a copy of the model to a writable location.



Change Visibility of Simulink Function

Change the visibility of the global function `f3` from global to scoped. Open the Simulink Function block `f3`. Inside the function block, double-click the Trigger Port block `f3`. In the block parameters dialog box, set **Function visibility** to `scoped`. Click **Apply** and **OK**.

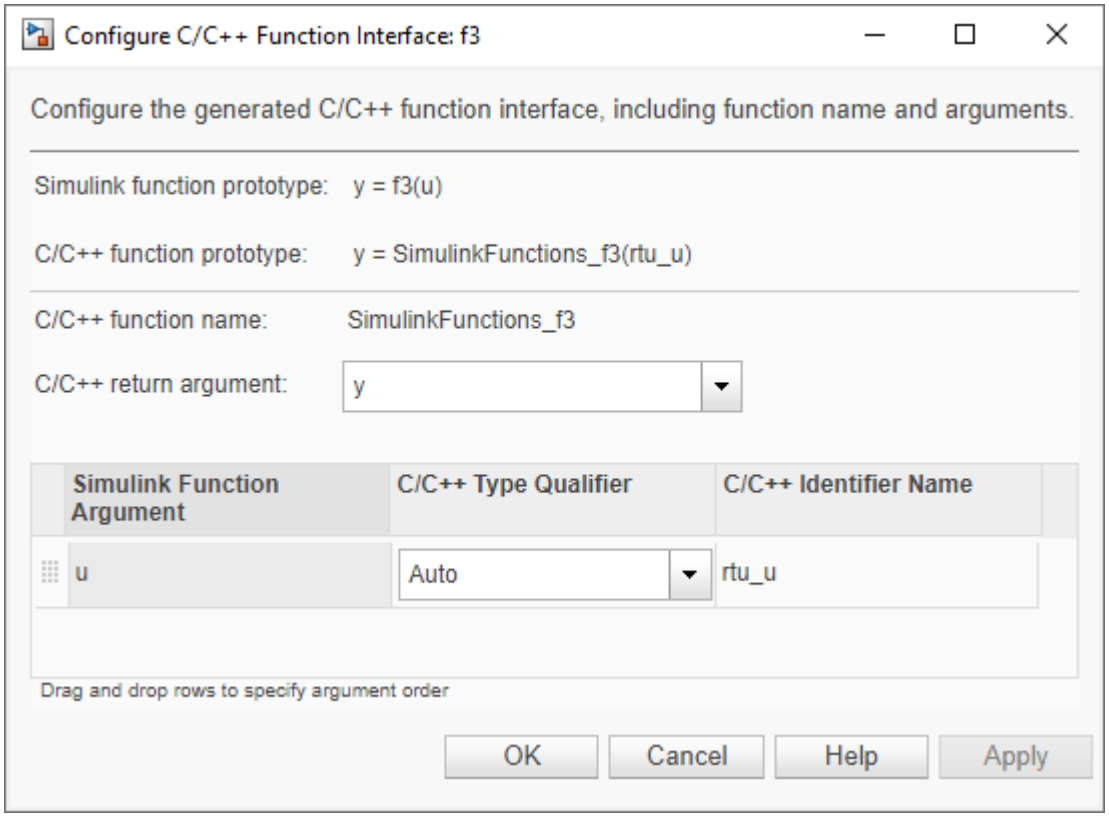
Open Configure C/C++ Function Interface Dialog Box

In the apps gallery, open the **Embedded Coder** app.

On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

In the Code Mappings editor, click the **Functions** tab.

In the row for `Simulink Function:f3`, click the function preview hyperlink. The Configure C/C++ Function Interface dialog box shows **Simulink function prototype** as `y = f3(u)` and **C/C++ function prototype** as `void f3(rtu_u, * rty_y)`. The preview for **C/C++ function prototype** is updated as you make changes.



For models that you configure for C multi-instantiable code, the interface preview for scoped Simulink functions includes a `self` argument. The `self` argument is a pointer to a version of the real-time model (`RT_MODEL`) data structure that stores multi-instance data associated with reusable functions.

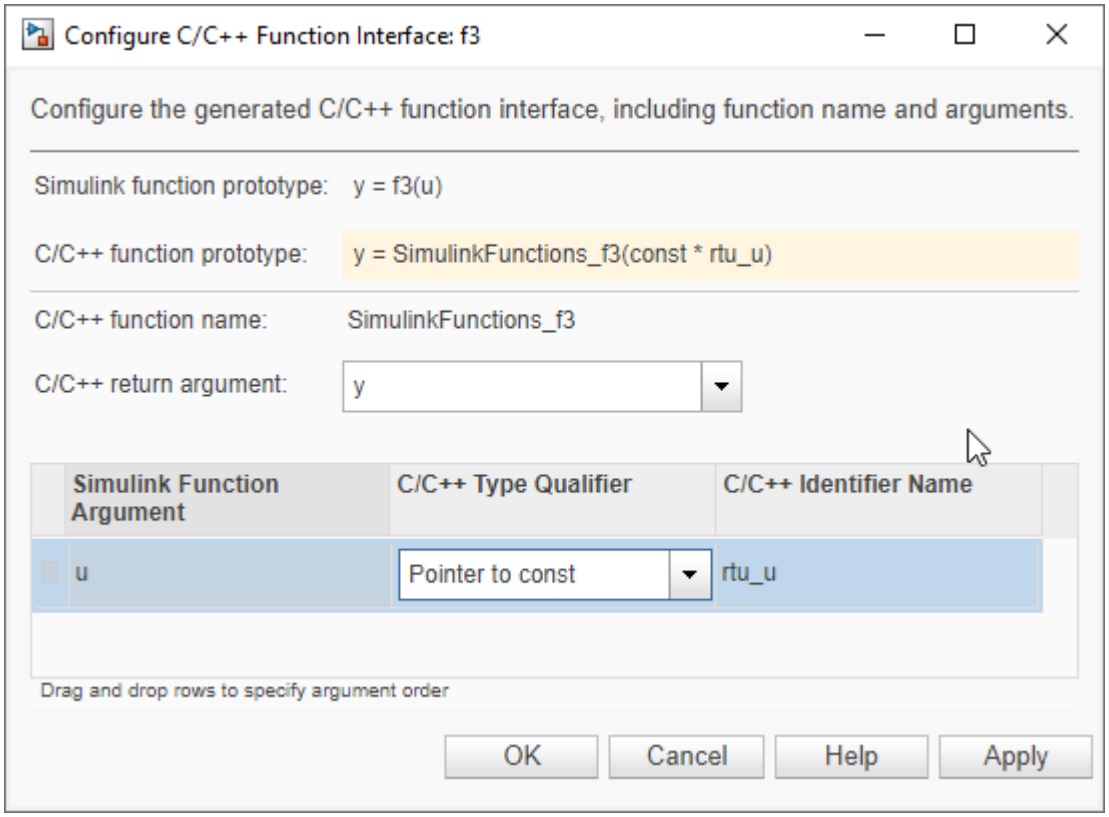
If model configuration parameter **Code interface packaging** is set to `Reusable function` and **Total number of instances allowed per top model** is set to `Multiple`, the argument appears as `self`, indicating that the code generator produces a self structure.

If model configuration parameter **Code interface packaging** is set to `Reusable function` or **Total number of instances allowed per top model** is set to `Multiple`, the argument appears as `[self]`, indicating that the argument is optional. The code generator produces a `self` structure only if you use the model as a top model and **Code interface packaging** is set to `Reusable function` or if you use the model as a referenced model and **Total number of instances allowed per top model** is set to `Multiple`.

For more information about configuring a model for multi-instantiable code generation, see “Generate Reentrant Code from Simulink Function Blocks” on page 8-18.

Configure Simulink Function Prototype

Configure the function arguments. For this example, configure the `u` argument. Set **C/C++ type Qualifier** to `Pointer to const`.



Verify your change by checking the function prototype preview.

Click **Apply**. Your modifications, whether made to a Simulink Function block or a Function Caller block, affect code generation for the Simulink Function block and corresponding Function Caller blocks in the model.

Optionally, you can change **C/C++ return argument** from `y` to `void`. In that case, the C function prototype is `void = SimulinkFunctions_f3(* rty_y, const * rtu_u)`.

Save your changes by clicking **OK**. Then, save the model.

Generate and Review Code

Make sure that the Simulink Function block is not selected. Then, generate code for the model.

In the **Code** view, open the generated file `SimulinkFunctions.c` and search for `SimulinkFunctions_f3`. The generated function code reflects the changes to the generated C function prototype.

```
real_T SimulinkFunctions_f3(const real_T *rtu_u)
{
  real_T rty_y_0;

  rtY.TicToc10 = rtDWork.Delay_DSTATE;

  rtDWork.Delay_DSTATE = (int8_T)(int32_T)-(int32_T)rtY.TicToc10;

  adder(rtB.Subtract, rtU.U2, *rtu_u, &rtB.FunctionCaller);
```

```
  rty_y_0 = rtB.FunctionCaller;

  return rty_y_0;
}
```

More About

- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Simulink Functions Overview”
- “Scoped Simulink Function Blocks in Models”
- “Simulink Function Blocks in Referenced Models”

Configure Generated C++ Function Interface for Exported Scoped Simulink Function Block

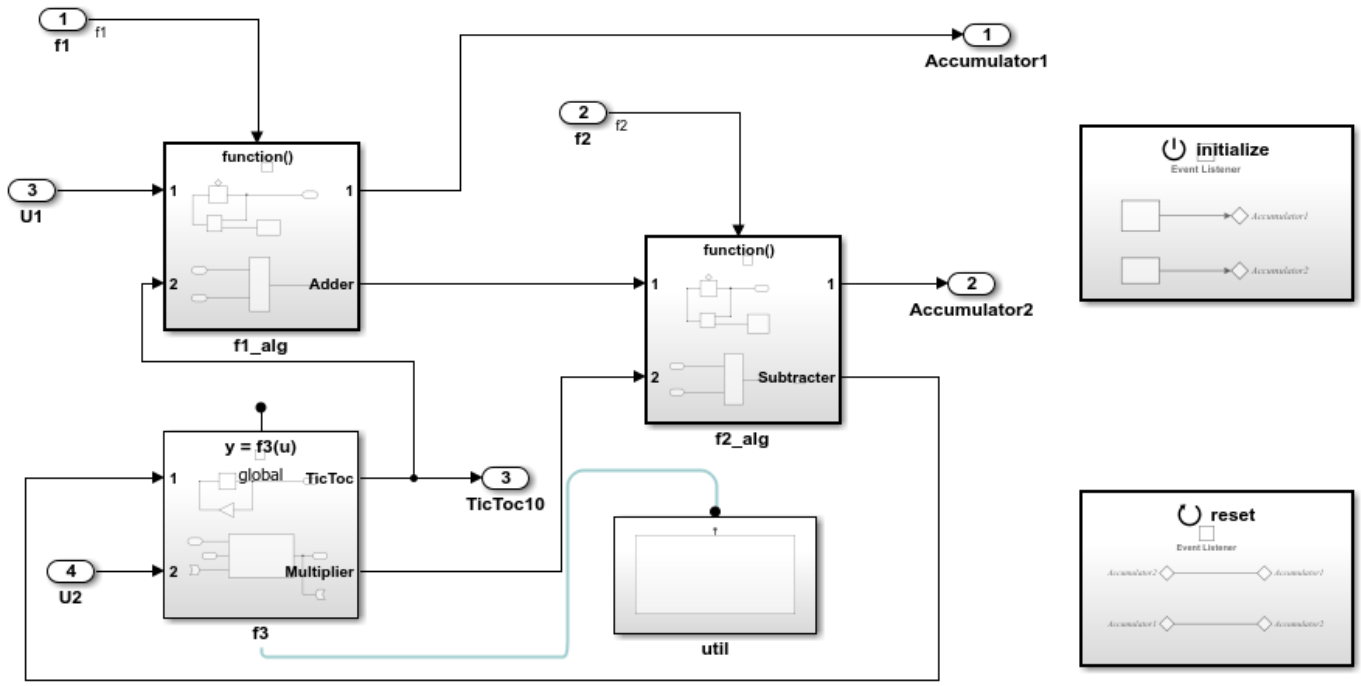
This example shows how to customize the C++ function interface that the code generator produces for an exported, scoped Simulink Function block.

For an exported, scoped function, you can modify the generated return argument, the argument type qualifiers, and the order of arguments. You cannot change the generated function name and argument names.

1. Open the example model.
2. Change the visibility of the Simulink function.
3. Change the programming language setting for code generation to C++.
4. Open the Configure C/C++ Function Interface dialog box.
5. Configure the Simulink Function prototype.
6. Generate and review the code.

Open Example Model

Open example model `SimulinkFunctions`. Save a copy of the model to a writable location.



Change Visibility of Simulink Function

Change the visibility of the global function `f3` from global to scoped. Open the Simulink Function block `f3`. Inside the function block, double-click the Trigger Port block `f3`. In the block parameters dialog box, set **Function visibility** to `scoped`. Click **Apply** and **OK**.

Change Language Setting for Code Generation

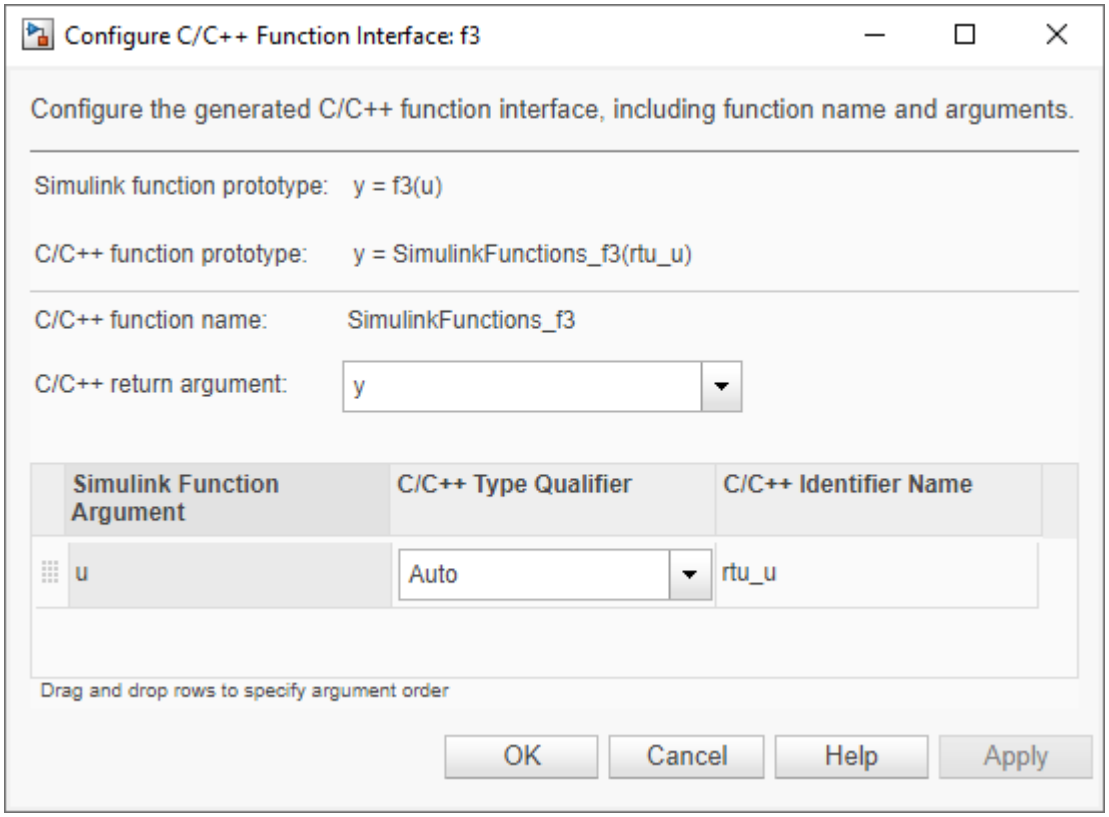
Set the model configuration parameter **Language** to `C++`.

Open Configure C/C++ Function Interface Dialog Box

In the Simulink Editor, right-click the Simulink Function block `f3`. In the context menu, select **C/C++ Code** > **Configure C/C++ Function Interface**. The dialog box shows **Simulink function prototype** as `y = f3(u)` and **C/C++ function prototype** as `y = rtwdemo_functions_f3(rtu_u)`. The preview for **C/C++ function prototype** is updated as you make changes.

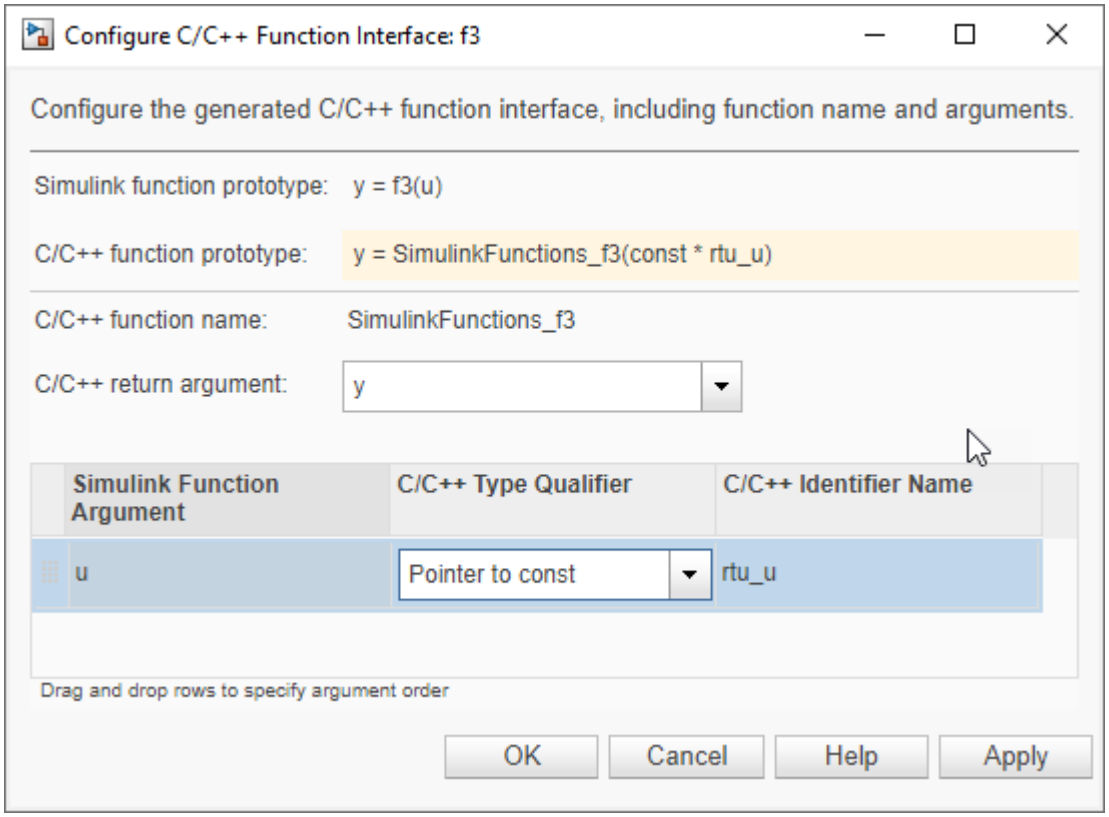
In the Code Mappings editor, click the **Functions** tab.

In the row for `Simulink Function:f3`, click the function preview hyperlink. The Configure C/C++ Function Interface dialog box shows **Simulink function prototype** as `y = f3(u)` and **C/C++ function prototype** as `void f3(rtu_u, * rty_y)`. The preview for **C/C++ function prototype** is updated as you make changes.



Configure Simulink Function Prototype

Configure the function arguments. For this example, configure the u argument. Set **C/C++ type Qualifier** to `Pointer to const`.



Verify your change by checking the function prototype preview.

Click **Apply**. Your modifications, whether made to a Simulink Function block or a Function Caller block, affect code generation for the Simulink Function block and corresponding Function Caller blocks in the model.

Optionally, you can change **C/C++ return argument** from `y` to `void`. In that case, the C++ function prototype is `void = SimulinkFunctions_f3(* rty_y, const * rtu_u)`.

Save your changes by clicking **OK**. Then, save the model.

Generate and Review Code

Make sure that the Simulink Function block is not selected. Then, generate code for the model.

In the **Code** view, open the generated file `SimulinkFunctions.cpp` and search for function `f3`. The generated function code reflects the changes to the generated C++ function prototype.

```
real_T SimulinkFunctionsModelClass::f3(const real_T *rtu_u)
{
  real_T rty_y_0;

  rtY.TicToc10 = rtDWork.Delay_DSTATE;


  rtDWork.Delay_DSTATE = static_cast<int8_T>(static_cast<int32_T>
    (-static_cast<int32_T>(rtY.TicToc10)));
```

```
  adder(rtB.Subtract, rtU.U2, *rtu_u, &rtB.FunctionCaller);

  rty_y_0 = rtB.FunctionCaller;

  return rty_y_0;
}
```

More About

- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18
- “Simulink Functions Overview”
- “Scoped Simulink Function Blocks in Models”
- “Simulink Function Blocks in Referenced Models”

Simulink Function Code Interface Limitations

- For C++ code generation, global Simulink functions are incompatible with C++ class interfaces for model entry-point functions.
- Simulink function code interface configuration does not support Simulink functions and function callers in Stateflow.
- The Argument Outport block cannot be a return argument for a Simulink function when its data type is set to one of the following types:
  - Complex types
  - Bus types
  - Image types
  - Non-scalar types
  - Multiword types

See Also

Related Examples

- “Simulink Function Blocks and Code Generation” on page 8-6
- “Generate Reentrant Code from Simulink Function Blocks” on page 8-18

Generate C++ Class Interface to Model or Subsystem Code

To generate a C++ class interface to model code, set model configuration parameter Code interface packaging to `C++ class`. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

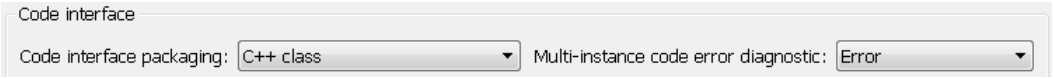
- Greater control over access to model data
- Ability to create multiple instances of model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Generate C++ Class Interface to Nonvirtual Subsystem Code” on page 42-16.)

Generate C++ Class Interface to Model Code

To generate encapsulated C++ class code from a model:

1. Set model configuration parameter **Language** to `C++`. This selection also enables C++ class code interface packaging for the model.
2. Verify that model configuration parameter **Code interface packaging** is set to `C++ class`.
3. Examine the setting of **Multi-instance code error diagnostic**. Leave the parameter at its default value `Error` unless you need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.



4. Generate code for the model.
5. Examine the C++ model class code in the generated files *`model`*`.h` and *`model`*`.cpp`. For example, the following code excerpt from the H file generated for the example model `SecondOrderSystem` shows the C++ class declaration for the model.

```
/* Class declaration for model SecondOrderSystem */
class SecondOrderSystemModelClass {
  /* public data and function members */
 public:
  /* External outputs */
  ExtY_SecondOrderSystem_T SecondOrderSystem_Y;

  /* Model entry point functions */

  /* model initialize function */
  void initialize();

  /* model step function */
  void step();

  /* model terminate function */
  void terminate();

  /* Constructor */
  SecondOrderSystemModelClass();

  /* Destructor */
  ~SecondOrderSystemModelClass();
```

```
  /* Real-Time Model get method */
  RT_MODEL_SecondOrderSystem_T * getRTM();
...
};
```

For more information about generating and calling model entry-point functions, see “Configure Generated C Function Interface for Model Entry-Point Functions”.

Note If you have an Embedded Coder license and you have selected an ERT-based system target file for your model, use additional **Code Generation** > **Interface** pane parameters to customize the generated C++ class interface.

Generate C++ Class Interface to Nonvirtual Subsystem Code

You can generate C++ class interfaces for right-click builds of nonvirtual subsystems, if the following requirements are met:

- The model is configured for the `C++` language and `C++ class` code interface packaging.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ class interfaces for a subsystem that meets the requirements:

1. Open the containing model and select the subsystem block.
2. Right-click the subsystem and select **C/C++ Code** > **Build This Subsystem**.
3. When the subsystem build completes, examine the C++ class interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point methods, see “Configure Generated C Function Interface for Model Entry-Point Functions”.

If you have an Embedded Coder license and you have selected an ERT-based system target file for your model, you can use the MATLAB command `RTW.configSubsystemBuild` to customize the generated C++ class interface to subsystem code.

C++ Class Interface Limitations

- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the C API interface is supported for `C++ class` code generation. If you select **External mode** or **ASAP2 interface**, code generation fails with a validation error.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. On the Simulink Signal Conversion block parameter dialog box, select **Exclude this block from 'Block reduction' optimization**.
- You cannot use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
  - Has a continuous sample time
  - Saves states

See Also

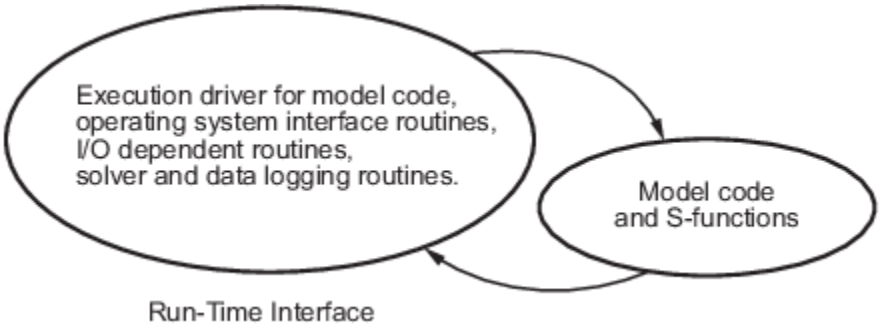
More About

- “Generate Reentrant Code from Top Models”
- “Configure Generated C Function Interface for Model Entry-Point Functions”
- “Execution of Code Generated from a Model”
- “Rapid Prototyping Model Functions”

Execution of Code Generated from a Model

The code generator produces algorithmic code as defined by your model. You can include external (for example, custom or legacy) code in a model by using techniques explained in “Choose an External Code Integration Workflow”.

The code generator also provides an interface that executes the generated model code. The interface and model code are compiled together to create an executable program. The next figure shows a high-level object-oriented view of the executable.



The Object-Oriented View of a Real-Time Program

In general, the conceptual design of the model execution driver does not change between the rapid prototyping and embedded style of generated code. The following sections describe model execution for single-tasking and multitasking environments both for simulation (non-real-time) and for real time. For most model code, the multitasking environment provides the most efficient model execution (that is, fastest sample rate).

The following concepts are useful in describing how model code executes.

- **Initialization**: `model_initialize` initializes the interface code and the model code.
- **ModelOutputs**: Calls blocks in your model that have a sample hit at the current time and has them produce their output. `model_output` can be done in major or minor time steps. In major time steps, the output is a given simulation time step. In minor time steps, the interface integrates the derivatives to update the continuous states.
- **ModelUpdate**: `model_update` calls blocks that have a sample hit at the current point in time and has them update their discrete states or similar type objects.
- **ModelDerivatives**: Calls blocks in your model that have continuous states and has them update their derivatives. `model_derivatives` is only called in minor time steps.
- **ModelTerminate**: `model_terminate` terminates the program if it is designed to run for a finite time. It destroys the real-time model data structure, deallocates memory, and can write data to a file.

Program Execution

A real-time program cannot require 100% of the CPU time. This requirement provides an opportunity to run background tasks during the free time.

Background tasks include operations such as writing data to a buffer or file, allowing access to program data by third-party data monitoring tools, or updating program parameters.

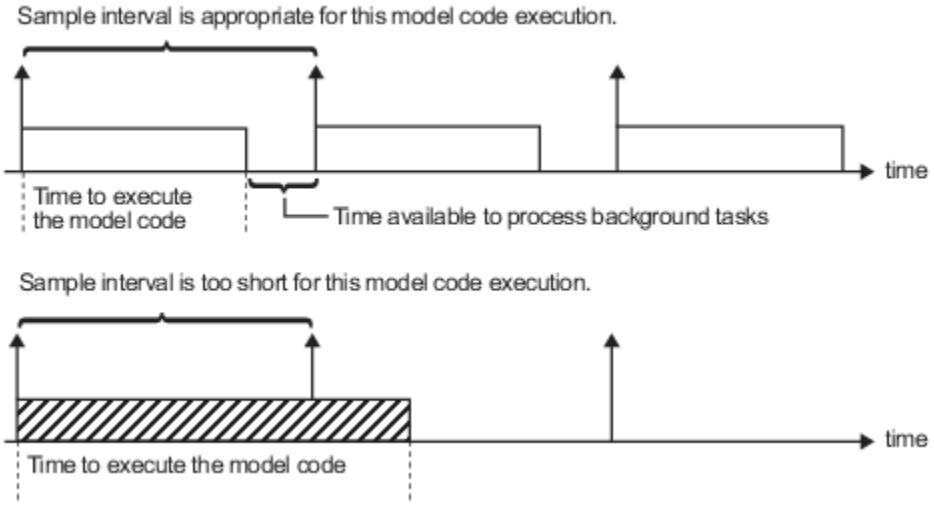
It is important, however, that the program be able to preempt the background task so the model code can execute in real time.

The way the program manages tasks depends on capabilities of the environment in which it operates.

Program Timing

Real-time programs require careful timing of the task invocations (either by using an interrupt or a real-time operating system tasking primitive) so that the model code executes to completion before another task invocation occurs. The timing includes time to read and write data to and from external hardware.

The next figure illustrates interrupt timing.



Task Timing

The sample interval must be long enough to allow model code execution between task invocations.

In the figure above, the time between two adjacent vertical arrows is the sample interval. The empty boxes in the upper diagram show an example of a program that can complete one step within the interval and still allow time for the background task. The gray box in the lower diagram indicates what happens if the sample interval is too short. Another task invocation occurs before the task is complete. Such timing results in an execution error.

If the real-time program is designed to run forever (that is, the final time is 0 or infinite so that the `while` loop never exits), then the shutdown code does not execute.

For more information on how the timing engine works, see “Absolute and Elapsed Time Computation”.

External Mode Communication

External mode allows communication between the Simulink block diagram and the standalone program that is built from the generated code. In this mode, the real-time program functions as an interprocess communication server, responding to requests from the Simulink engine.

Data Logging in Single-Tasking and Multitasking Model Execution

"Configure Model for Debugging" on page 48-34 explains how you can save system states, outputs, and time to a MAT-file at the completion of the model execution. The `LogTXY` function, which performs data logging, operates differently in single-tasking and multitasking environments.

If you examine how `LogTXY` is called in the single-tasking and multitasking environments, notice that for single-tasking `LogTXY` is called after `ModelOutputs`. During this `ModelOutputs` call, blocks that have a hit at time *t* execute, whereas in multitasking, `LogTXY` is called after `ModelOutputs(tid=0)`, which executes only the blocks that have a hit at time *t* and that have a task identifier of 0. This results in differences in the logged values between single-tasking and multitasking logging. Specifically, consider a model with two sample times, the faster sample time having a period of 1.0 second and the slower sample time having a period of 10.0 seconds. At time t = k*10, k=0,1,2... both the fast (`tid=0`) and slow (`tid=1`) blocks execute. When executing in multitasking mode, when `LogTXY` is called, the slow blocks execute, but the previous value is logged, whereas in single-tasking the current value is logged.

Another difference occurs when logging data in an enabled subsystem. Consider an enabled subsystem that has a slow signal driving the enable port and fast blocks within the enabled subsystem. In this case, the evaluation of the enable signal occurs in a slow task, and the fast blocks see a delay of one sample period; thus the logged values will show these differences.

To summarize differences in logged data between single-tasking and multitasking, differences will be seen when

- A root outport block has a sample time that is slower than the fastest sample time
- A block with states has a sample time that is slower than the fastest sample time
- A block in an enabled subsystem where the signal driving the enable port is slower than the rate of the blocks in the enabled subsystem

For the first two cases, even though the logged values are different between single-tasking and multitasking, the model results are not different. The only real difference is where (at what point in time) the logging is done. The third (enabled subsystem) case results in a delay that can be seen in a real-time environment.

Non-Real-Time Single-Tasking Systems

This pseudocode shows the execution of a model for a non-real-time single-tasking system.

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs     -- Major time step.
    LogTXY           -- Log time, states and root outports.
    ModelUpdate      -- Major time step.
    Integrate        -- Integration in minor time step for
                     -- models with continuous states.
      ModelDerivatives
      Do 0 or more
        ModelOutputs
        ModelDerivatives
      EndDo -- Number of iterations depends upon the solver
      Integrate derivatives to update continuous states.
```

```
    EndIntegrate
  EndWhile
  Termination
}
```

The initialization phase begins first. This consists of initializing model states and setting up the execution engine. The model then executes, one step at a time. First `ModelOutputs` executes at time t, then the workspace I/O data is logged, and then `ModelUpdate` updates the discrete states. Next, if your model has continuous states, `ModelDerivatives` integrates the continuous states' derivatives to generate the states for time $t_{new} = t + h$, where h is the step size. Time then moves forward to t_{new} and the process repeats.

During the `ModelOutputs` and `ModelUpdate` phases of model execution, only blocks that reach the current point in time execute.

Non-Real-Time Multitasking Systems

This pseudocode shows the execution of a model for a non-real-time multitasking system.

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs(tid=0)   -- Major time step.
    LogTXY                -- Log time, states, and root
                          -- outports.
    ModelUpdate(tid=0)    -- Major time step.
    Integrate       -- Integration in minor time step for
                    -- models with continuous states.
      ModelDerivatives
      Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
    For i=1:NumTids
      ModelOutputs(tid=i) -- Major time step.
      ModelUpdate(tid=i)  -- Major time step.
    EndFor
  EndWhile
  Termination
  }
```

Multitasking operation is more complex than single-tasking execution because the output and update functions are subdivided by the *task identifier* (`tid`) that is passed into these functions. This allows for multiple invocations of these functions with different task identifiers using overlapped interrupts, or for multiple tasks when using a real-time operating system. In simulation, multiple tasks are emulated by executing the code in the order that would occur if preemption did not exist in a real-time system.

Multitasking execution assumes that task rates are multiples of the base rate. The Simulink product enforces this when you create a fixed-step multitasking model. The multitasking execution loop is very similar to that of single-tasking, except for the use of the task identifier (`tid`) argument to `ModelOutputs` and `ModelUpdate`.

You cannot use `tid` values from code generated by a target file and not by Simulink Coder. Simulink Coder tracks the use of `tid` when generating code for a specific subsystem or function type. When you generate code in a target file, this argument cannot be tracked because the scope does not have subsystem or function type. Therefore, `tid` becomes an undefined variable and your target file fails to compile.

Real-Time Single-Tasking Systems

This pseudocode shows the execution of a model in a real-time single-tasking system where the model is run at interrupt level.

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs    -- Major time step.
  LogTXY          -- Log time, states and root outports.
  ModelUpdate     -- Major time step.
  Integrate       -- Integration in minor time step for models
                  -- with continuous states.
     ModelDerivatives
     Do 0 or more
       ModelOutputs
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives to update continuous states.
  EndIntegrate
}

main()
{
  Initialization (including installation of rtOneStep as an
  interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Real-time single-tasking execution is very similar to non-real-time single-tasking execution, except that instead of free-running the code, the `rt_OneStep` function is driven by a periodic timer interrupt.

At the interval specified by the program's base sample rate, the interrupt service routine (ISR) preempts the background task to execute the model code. The base sample rate is the fastest in the model. If the model has continuous blocks, then the integration step size determines the base sample rate.

For example, if the model code is a controller operating at 100 Hz, then every 0.01 seconds the background task is interrupted. During this interrupt, the controller reads its inputs from the analog-to-digital converter (ADC), calculates its outputs, writes these outputs to the digital-to-analog converter (DAC), and updates its states. Program control then returns to the background task. These steps must occur before the next interrupt.

Real-Time Multitasking Systems

This pseudocode shows how a model executes in a real-time multitasking system where the model is run at interrupt level.

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs(tid=0)     -- Major time step.
  LogTXY                  -- Log time, states and root outports.
  ModelUpdate(tid=0)      -- Major time step.
  Integrate               -- Integration in minor time step for
                          -- models with continuous states.
     ModelDerivatives
     Do 0 or more
       ModelOutputs(tid=0)
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives and update continuous states.
  EndIntegrate
  For i=1:NumTasks
    If (hit in task i)
      ModelOutputs(tid=i)
      ModelUpdate(tid=i)
    EndIf
  EndFor
}

main()
{
  Initialization (including installation of rtOneStep as an
    interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Running models at interrupt level in a real-time multitasking environment is very similar to the previous single-tasking environment, except that overlapped interrupts are employed for concurrent execution of the tasks.

The execution of a model in a single-tasking or multitasking environment when using real-time operating system tasking primitives is very similar to the interrupt-level examples discussed above. The pseudocode below is for a single-tasking model using real-time tasking primitives.

```
tSingleRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    ModelOutputs            -- Major time step.
    LogTXY                  -- Log time, states and root
                            -- outports
```

```
    ModelUpdate             -- Major time step
    Integrate               -- Integration in minor time step
                            -- for models with continuous
                            -- states.
      ModelDeriviatives
      Do 0 or more
        ModelOutputs
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndMainLoop
}

main()
{
  Initialization
  Start/spawn task "tSingleRate".
  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

In this single-tasking environment, the model executes as real-time operating system tasking primitives. In this environment, create a single task (`tSingleRate`) to run the model code. This task is invoked when a clock tick occurs. The clock tick gives a `clockSem` (clock semaphore) to the model task (`tSingleRate`). The model task waits for the semaphore before executing. The clock ticks occur at the fundamental step size (base rate) for your model.

Multitasking Systems Using Real-Time Tasking Primitives

This pseudocode is for a multitasking model using real-time tasking primitives.

```
tSubRate(subTaskSem,i)
{
  Loop:
    Wait on semaphore subTaskSem.
    ModelOutputs(tid=i)
    ModelUpdate(tid=i)
  EndLoop
}
tBaseRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    For i=1:NumTasks
      If (hit in task i)
        If task i is currently executing, then error out due to
          overflow.
        Do a "semGive" on subTaskSem for task i.
      EndIf
    EndFor
    ModelOutputs(tid=0)    -- major time step.
    LogTXY                 -- Log time, states and root outports.
    ModelUpdate(tid=0)     -- major time step.
    Loop:                  -- Integration in minor time step for
```

```
                          -- models with continuous states.
      ModelDeriviatives
      Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (number of iterations depends upon the solver).
      Integrate derivatives to update continuous states.
    EndLoop
  EndMainLoop
}
main()
{
  Initialization
  Start/spawn task "tSubRate".
  Start/spawn task "tBaseRate".

  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

In this multitasking environment, the model is executed using real-time operating system tasking primitives. Such environments require several model tasks (`tBaseRate` and several `tSubRate` tasks) to run the model code. The base rate task (`tBaseRate`) has a higher priority than the subrate tasks. The subrate task for `tid=1` has a higher priority than the subrate task for `tid=2`, and so on. The base rate task is invoked when a clock tick occurs. The clock tick gives a `clockSem` to `tBaseRate`. The first thing `tBaseRate` does is give semaphores to the subtasks that have a hit at the current point in time. Because the base rate task has a higher priority, it continues to execute. Next it executes the fastest task (`tid=0`), consisting of blocks in your model that have the fastest sample time. After this execution, it resumes waiting for the clock semaphore. The clock ticks are configured to occur at the fundamental step size for your model.

Rapid Prototyping and Embedded Model Execution Differences

The rapid prototyping program framework provides a common application programming interface (API) that does not change between model definitions.

The Embedded Coder product provides a different framework called the embedded program framework. The embedded program framework provides an optimized API that is tailored to your model. When you use the embedded style of generated code, you are modeling how you would like your code to execute in your embedded system. Therefore, the definitions defined in your model should be specific to your embedded targets. Items such as the model name, parameter, and signal storage class are included as part of the API for the embedded style of code.

One major difference between the rapid prototyping and embedded style of generated code is that the latter contains fewer entry-point functions. The embedded style of code can be configured to have only one function, `model_step`.

Thus, model execution code eliminates `Loop...EndLoop` statements and groups `ModelOutputs`, `LogTXY`, and `ModelUpdate` into a single statement, `model_step`.

For more information about how generated embedded code executes, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

See Also

More About

- “Time-Based Scheduling and Code Generation” on page 29-2
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Time-Based Scheduling Example Models” on page 29-35

Rapid Prototyping Model Functions

Rapid prototyping code defines the following functions that interface with the main program (`main.c` or `main.cpp`):

- `Model()`: The model registration function. This function initializes the work areas (for example, allocating and setting pointers to various data structures) used by the model. The model registration function calls the `MdlInitializeSizes` and `MdlInitializeSampleTimes` functions. These two functions are very similar to the S-function `mdlInitializeSizes` and `mdlInitializeSampleTimes` methods.
- `MdlStart(void)`: After the model registration functions `MdlInitializeSizes` and `MdlInitializeSampleTimes` execute, the main program starts execution by calling `MdlStart`. This routine is called once at startup.

  The function `MdlStart` has four basic sections:

  - Code to initialize the states for each block in the root model that has states. A subroutine call is made to the “initialize states” routines of conditionally executed subsystems.
  - Code generated by the one-time initialization (start) function for each block in the model.
  - Code to enable the blocks in the root model that have enable methods, and the blocks inside triggered or function-call subsystems residing in the root model. Simulink blocks can have enable and disable methods. An enable method is called just before a block starts executing, and the disable method is called just after the block stops executing.
  - Code for each block in the model whose output value is constant. The block code appears in the `MdlStart` function only if the block parameters are not tunable in the generated code and if the code generator cannot eliminate the block code through constant folding.
- `MdlOutputs(int_T tid)`: `MdlOutputs` updates the output of blocks. The `tid` (task identifier) parameter identifies the task that in turn maps when to execute blocks based upon their sample time. This routine is invoked by the main program during major and minor time steps. The major time steps are when the main program is taking an actual time step (that is, it is time to execute a specific task). If your model contains continuous states, the minor time steps will be taken. The minor time steps are when the solver is generating integration stages, which are points between major outputs. These integration stages are used to compute the derivatives used in advancing the continuous states.
- `MdlUpdate(int_T tid)`: `MdlUpdate` updates the states and work vector state information (that is, states that are neither continuous nor discrete) saved in work vectors. The `tid` (task identifier) parameter identifies the task that in turn indicates which sample times are active, allowing you to conditionally update only states of active blocks. This routine is invoked by the interface after the major `MdlOutputs` has been executed. The solver is also called, and *`model`*`_Derivatives` is called in minor steps by the solver during its integration stages. All blocks that have continuous states have an identical number of derivatives. These blocks are required to compute the derivatives so that the solvers can integrate the states.
- `MdlTerminate(void)`: `MdlTerminate` contains any block shutdown code. `MdlTerminate` is called by the interface, as part of the termination of the real-time program.

The contents of the above functions are directly related to the blocks in your model. A Simulink block can be generalized to the following set of equations.

$$y = f_0(t, x_c, x_d, u)$$

Output y is a function of continuous state x_c, discrete state x_d, and input u. Each block writes its specific equation in a section of `MdlOutputs`.

$$x_{d+1} = f_u(t, x_d, u)$$

The discrete states x_d are a function of the current state and input. Each block that has a discrete state updates its state in `MdlUpdate`.

$$\dot{x} = f_d(t, x_c, u)$$

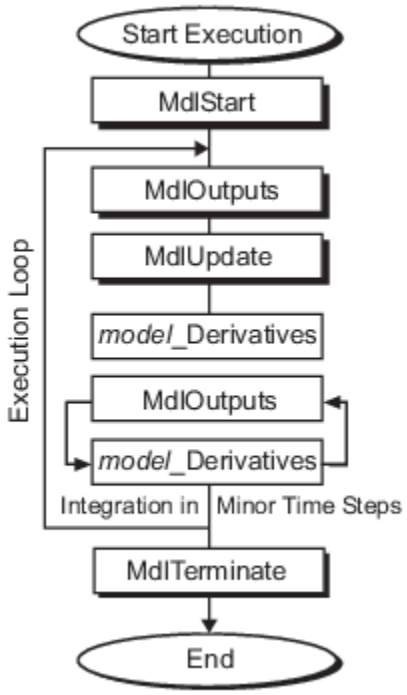
The derivatives x are a function of the current input. Each block that has continuous states provides its derivatives to the solver (for example, `ode5`) in *`model`*`_Derivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

The output, y, is generally written to the block I/O structure. Root-level Outport blocks write to the external outputs structure. The continuous and discrete states are stored in the states structure. The input, u, can originate from another block's output, which is located in the block I/O structure, an external input (located in the external inputs structure), or a state. These structures are defined in the *`model`*`.h` file that the Simulink Coder software generates.

The next example shows the general contents of the rapid prototyping style of C code written to the *`model`*`.c` file.

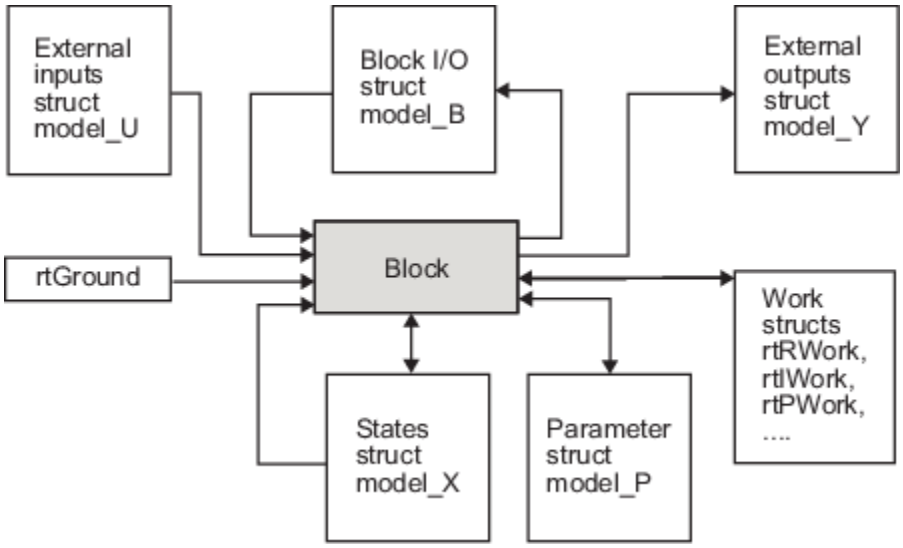
```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
*/
<includes>
void MdlStart(void)
{
  /*
   * State initialization code.
   * Model start-up code - one time initialization code.
   * Execute any block enable methods.
   * Initialize output of any blocks with constant sample times.
   */
}

void MdlOutputs(int_T tid)
{
  /* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
  /* Compute: xd+1 = fu(t,xd,u) for each block as needed. */

  /* Compute: dxc = fd(t,xc,u) for each block in model_derivatives
                               as needed. */
}
void MdlTerminate(void)
{
  /* Perform shutdown code for any blocks that
     have a termination action */
}
```

This figure shows a flow chart describing the execution of the rapid prototyping generated code.



Rapid Prototyping Execution Flow Chart

Each block places code in specific `Mdl` routines according to the algorithm that it is implementing. Blocks have input, output, parameters, and states, as well as other general items. For example, in general, block inputs and outputs are written to a block I/O structure (*`model`*`_B`). Block inputs can also come from the external input structure (*`model`*`_U`) or the state structure when connected to a state port of an integrator (*`model`*`_X`), or ground (`rtGround`) if unconnected or grounded. Block outputs can also go to the external output structure (*`model`*`_Y`). This figure shows the general mapping between these items.



Data View of the Generated Code

The following list defines the structures shown in the preceding figure:

- Block I/O structure (*model*_B): This structure consists of persistent block output signals. The number of block output signals is the sum of the widths of the data output ports of the nonvirtual blocks in your model. If you activate block I/O optimizations, the Simulink and Simulink Coder products reduce the size of the *model*_B structure by

  - Reusing the entries in the *model*_B structure
  - Making other entries local variables

  See “How Generated Code Stores Internal Signal, State, and Parameter Data” for more information on these optimizations.

  Structure field names are determined either by the block's output signal name (when present) or by the block name and port number when the output signal is left unlabeled.
- Block states structures: The continuous states structure (*model*_X) contains the continuous state information for blocks in your model that have continuous states. Discrete states are stored in a data structure called the *DWork vector* (*model*_DWork).
- Block parameters structure (*model*_P): The parameters structure contains block parameters that can be changed during execution (for example, the parameter of a Gain block).
- External inputs structure (*model*_U): The external inputs structure consists of the root-level Inport block signals. Field names are determined by either the block's output signal name, when present, or by the Inport block's name when the output signal is left unlabeled.
- External outputs structure (*model*_Y): The external outputs structure consists of the root-level Outport blocks. Field names are determined by the root-level Outport block names in your model.
- Real work, integer work, and pointer work structures (*model*_RWork, *model*_IWork, *model*_PWork): Blocks might have a need for real, integer, or pointer work areas. For example, the Memory block uses a real work element for each signal. These areas are used to save internal states or similar information.

See Also

More About

- “Time-Based Scheduling and Code Generation” on page 29-2
- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Function and Class Interfaces in Embedded Coder

- “Generate Modular Function Code for Nonvirtual Subsystems” on page 43-2
- “Generate Reentrant, Multi-Instance Code” on page 43-19
- “Generate Code for String Blocks by Using the Standard C++ String Library” on page 43-22
- “Select Code Generation Output for Target Platform Deployment” on page 43-25
- “Configure C Code Deployment Types for Model Hierarchy” on page 43-29

Generate Modular Function Code for Nonvirtual Subsystems

About Nonvirtual Subsystem Code Generation

By default, when producing code for a nonvirtual subsystem, the code generator places internal data associated with a nonvirtual subsystem in the same data structure as internal data for the parent model. This can make it difficult to trace and test code, particularly for nonreusable subsystems. Also, in large models containing nonvirtual subsystems, data structures can become large and potentially difficult to compile.

To generate modular function code for nonvirtual subsystems, including atomic subsystems and conditionally executed subsystems, use the subsystem block parameter **Function with separate data**. This block parameter instructs the code generator to produce a block I/O and DWork data structure for the nonvirtual subsystem function that is independent from the parent model data structures. As a result, generated code for the subsystem:

- Is easier to trace.
- Is easier to test.
- Reduces the size of a model's global data structures.

To use the **Function with separate data** parameter,

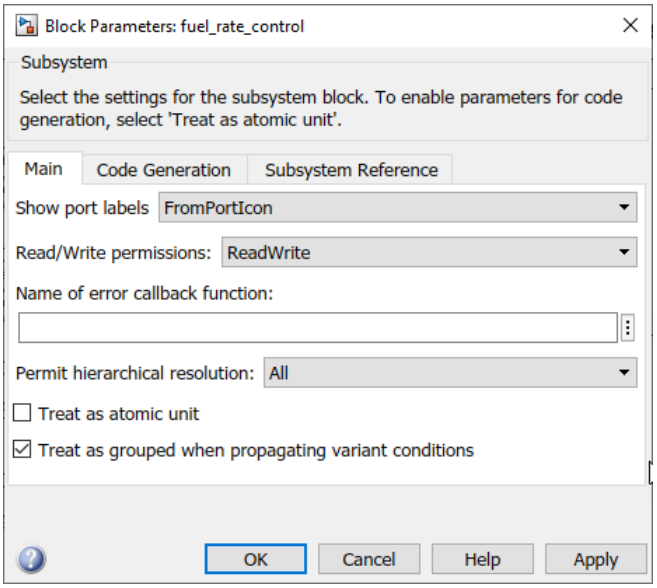
- Configure the model with an ERT-based system target file.
- Configure the subsystem to be atomic or conditionally executed.
- Set the subsystem block parameter **Function packaging** to `Nonreusable function`.

To configure your subsystem for generating modular function code, invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See “Configure Subsystem for Generating Modular Function Code” on page 43-2 and “Modular Function Code for Nonvirtual Subsystems” on page 43-5 for details. For limitations that apply, see “Nonvirtual Subsystem Modular Function Code Limitations” on page 43-17.

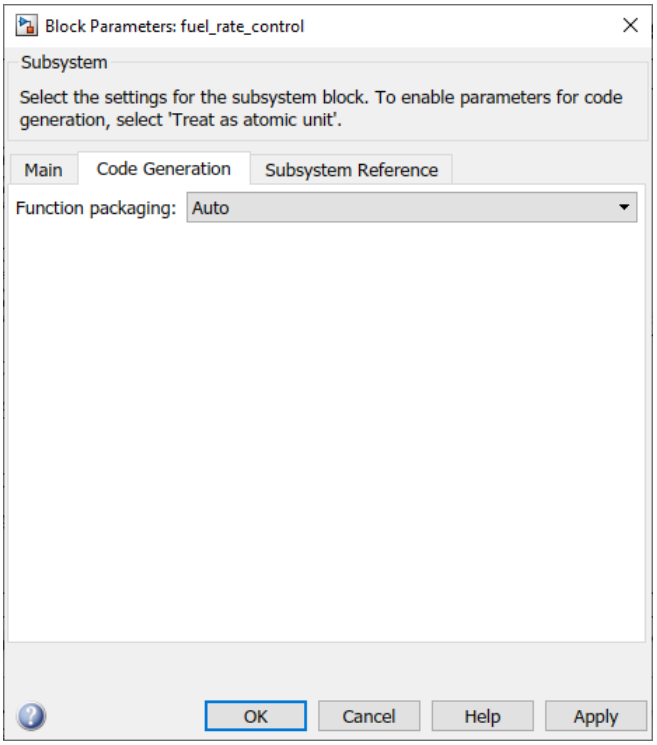
For more information about generating code for atomic subsystems, see “Generate Subsystem Code as Separate Function and Files” and “Generate Code and Executables for Individual Subsystems”.

Configure Subsystem for Generating Modular Function Code

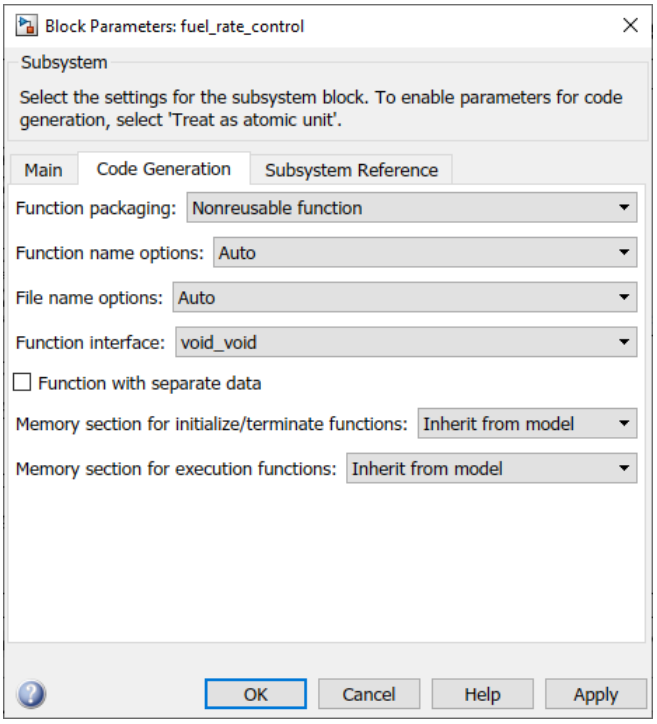
1. Verify that the model containing the subsystem uses an ERT-based system target file.
2. Select the subsystem for which you want to generate modular function code and open the Subsystem Parameters dialog box. The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)



3 If the block parameter **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally executed. Select the parameter **Treat as atomic unit**, which enables the **Function packaging** parameter on the **Code Generation** tab. Select the **Code Generation** tab.

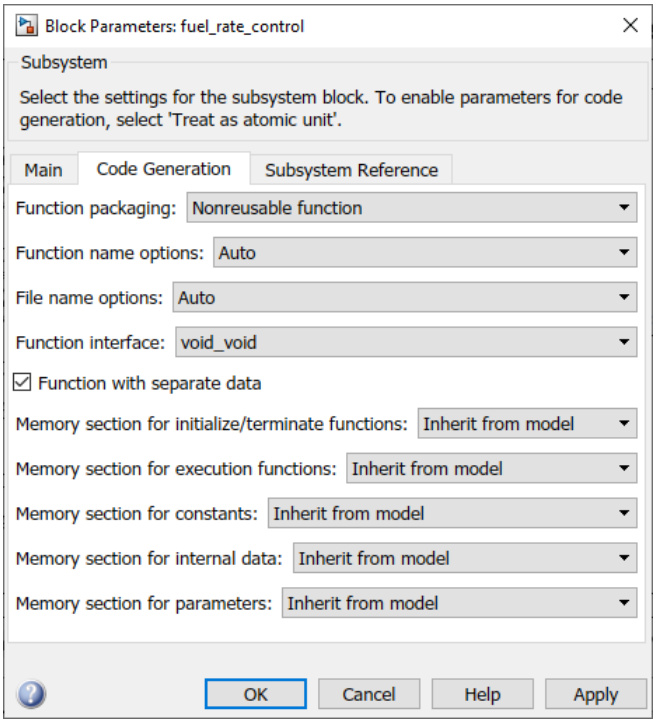


4 For the **Function packaging** parameter, select `Nonreusable function`. After you make this selection, the **Function with separate data** parameter is displayed.



Before you generate code for the nonvirtual subsystem with the **Function with separate data**parameter selected, consider generating function code with the parameter cleared and save the generated function `.c` and `.h` files in a separate directory for later comparison.

5 Select the **Function with separate data** parameter. Additional parameters appear.



To control the naming of the generated subsystem function and the subsystem files, modify subsystem parameters **Function name options** and **File name options**.

6 Save your subsystem parameter changes and exit the dialog box by clicking **OK**.

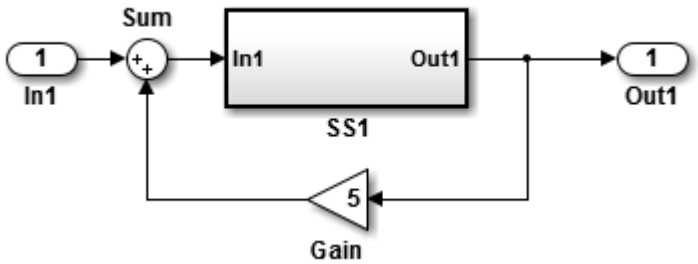
7 Generate code for the subsystem and examine the generated files, including the function `.c` and `.h` files named according to your subsystem parameter specifications.

For more information on generating code for nonvirtual subsystems, see “Generate Subsystem Code as Separate Function and Files” on page 4-7. For examples of generated subsystem function code, see “Modular Function Code for Nonvirtual Subsystems” on page 43-5.

Modular Function Code for Nonvirtual Subsystems

This example shows how to generate nonvirtual subsystem function code with the **Function with separate data** parameter cleared and selected and compares the results.

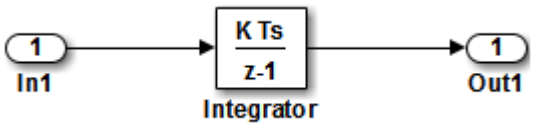
1 Open example model `rtwdemo_atomic`. Then, open the **Embedded Coder** app. Change the system target file to `ert.tlc`.



This model shows how to preserve the boundary of a virtual subsystem. When you select subsystem block parameter **Treat as atomic unit**, the code that the code generator produces for the subsystem executes as an atomic unit. When configured as atomic, you can specify how the code generator represents the subsystem by setting the **Function Packaging** parameter on the **Code Generation** tab. You can specify that the subsystem is translated to one of these types of implementation:

- `Inline`: Inlined subsystem code at call sites.
- `Function`: `void/void` function with I/O and internal data in the model global data structure.
- `Reusable Function`: Reentrant function with data passed in as function arguments.
- `Auto`: Code generator optimizes the implementation based on context.

2 Double-click the SS1 subsystem and examine the contents.



Then, close the subsystem window.

3 Right-click the SS1 subsystem, select Block Parameters (Subsystem) from the context menu, and examine the settings. Simulink and the code generator can avoid "artificial" algebraic loops when

you make the subsystem atomic with subsystem parameter **Minimize algebraic loop occurrences**.

4. Create a variant of `rtwdemo_atomic` that shows function code *without* data separation.

   a. In the Subsystem Parameters dialog box,

      - In the **Main** tab, select **Treat as atomic unit**.
      - In the **Code Generation** tab:

        - Set **Function packaging** to `Nonreusable function`.
        - Set **Function name options** to `User specified`.
        - Set **Function name** to `myfun`.
        - Set **File name options** to `Use function name`. This setting is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into files `myfun.c` and `myfun.h`.

      Do *not* select the **Function with separate data** parameter.

   b. Click **Apply** to apply the changes and click **OK** to exit the dialog box.

   c. Save the model variant with a unique file name (for example, `rtwdemo_atomic1`) to a writable location.

5. Create a variant of `rtwdemo_atomic` that shows function code *with* data separation.

   a. Open model `rtwdemo_atomic`.

   b. Open the **Embedded Coder** app. Change the system target file to `ert.tlc`.

   c. In the model canvas, right-click the SS1 subsystem and select **Block Parameters (Subsystem)**. In the Subsystem Parameters dialog box,

      - In the **Main** tab, select **Treat as atomic unit**.
      - In the **Code Generation** tab:

        - Set **Function packaging** to `Nonreusable function`.
        - Set **Function name options** to `User specified`.
        - Set **Function name** to `myfun`.
        - Set **File name options** to `Use function name`.
        - Select **Function with separate data**.

   d. Click **Apply** to apply the change and click **OK** to exit the dialog box.

   e. Save the model variant with a unique file name (for example, `rtwdemo_atomic2`) to a writable location.

6. Generate code for each model (for example, `rtwdemo_atomic1` and `rtwdemo_atomic2`).

7. Compare the *`model`*`.c/.h` and `myfun.c/.h` files generated for the two models. For code comparison discussion, see “H File Differences for Nonvirtual Subsystem Function Data Separation” on page 43-7 and “C File Differences for Nonvirtual Subsystem Function Data Separation” on page 43-7.

   In this example, there are not significant differences in the generated variants of `ert_main.c`, *`model`*`_private.h`, *`model`*`_types.h`, or `rtwtypes.h`.

H File Differences for Nonvirtual Subsystem Function Data Separation

- Selecting **Function with separate data** causes the code generator to place type definitions for subsystem data in the `myfun.h` file for `rtwdemo_atomic2`:

  ```
  /* Block states (default storage) for system '<Root>/SS1' */
  typedef struct {
    real_T Integrator_DSTATE;            /* '<S1>/Integrator' */
  } DW_myfun_T;
  ```

  For `rtwdemo_atomic1`, type definitions for subsystem data belong to the model and appear in `rtwdemo_atomic1.h`:

  ```
  /* Block signals (default storage) */
  typedef struct {
    real_T Sum;                          /* '<Root>/Sum' */
  } B_rtwdemo_atomic_1_T;

  /* Block states (default storage) for system '<Root>' */
  typedef struct {
    real_T Integrator_DSTATE;            /* '<S1>/Integrator' */
  } DW_rtwdemo_atomic_1_T;
  ```

- Selecting **Function with separate data** generates the following external declarations in the `myfun.h` file for `rtwdemo_atomic2`:

  ```
  /* Extern declarations of internal data for system '<Root>/SS1' */
  extern DW_myfun_T myfun_DW;
  extern void myfun_Update(void);
  extern void myfun(void);
  ```

  By contrast, the generated code for `rtwdemo_atomic1` contains model-level external declarations for the subsystem `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.h`:

  ```
  /* Block signals (default storage) */
  extern B_rtwdemo_atomic_1_T rtwdemo_atomic_1_B;

  /* Block states (default storage) */
  extern DW_rtwdemo_atomic_1_T rtwdemo_atomic_1_DW;
  ```

C File Differences for Nonvirtual Subsystem Function Data Separation

- Selecting **Function with separate data** causes a separate subsystem initialize function, `myfun_initialize`, to be generated in the `myfun.c` file for `rtwdemo_atomic2`:

  ```
  void myfun_initialize(void) {
    {
      ((real_T*)&rtwdemo_atomic2_myfunB.Integrator)[0] = 0.0;
    }
    rtwdemo_atomic2_myfunDW.Integrator_DSTATE = 0.0;
  }
  ```

  The subsystem initialize function in `myfun.c` is invoked by the model initialize function in `rtwdemo_atomic2.c`:

  ```
  /* Model initialize function */

  void rtwdemo_atomic2_initialize(void)
  {
  ...
  ```

```
  /* Initialize subsystem data */
  myfun_initialize();
}
```

By contrast, for `rtwdemo_atomic1`, subsystem data is initialized by the model initialize function in `rtwdemo_atomic1.c`:

```
/* Model initialize function */

void rtwdemo_atomic1_initialize(void)
{
...
  /* block I/O */
  {
 ...
    ((real_T*)&rtwdemo_atomic1_B.Integrator)[0] = 0.0;
  }

  /* states (dwork) */

  rtwdemo_atomic1_DWork.Integrator_DSTATE = 0.0;
...
}
```

- Selecting **Function with separate data** generates the following declarations in the `myfun.c` file for `rtwdemo_atomic2`:

  ```
  /* Declare variables for internal data of system '<Root>/SS1' */
  DW_myfun_T myfun_DW;
  ```

  By contrast, the generated code for `rtwdemo_atomic1` contains model-level declarations for the subsystems `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.c`:

  ```
  /* Block signals (default storage) */
  B_rtwdemo_atomic_1_T rtwdemo_atomic_1_B;

  /* Block states (default storage) */
  DW_rtwdemo_atomic_1_T rtwdemo_atomic_1_DW;
  ```

- Selecting **Function with separate data** generates identifier naming that reflects the subsystem orientation of data items. The references to subsystem data in subsystem functions, such as `myfun` and `myfun_update`, are in the model *`model`*`_step` function. For example, compare this code from `myfun` for `rtwdemo_atomic2`

  ```
  /*  DiscreteIntegrator: '<S1>/Integrator' */
    rtwdemo_atomic_2_Y.Out1 = myfun_DW.Integrator_DSTATE;
  ```

  to the corresponding code from `myfun` for `rtwdemo_atomic1`.

  ```
  /* DiscreteIntegrator: '<S1>/Integrator' */
  rtwdemo_atomic_1_Y.Out1 = rtwdemo_atomic_1_DW.Integrator_DSTATE;
  ```

Partition Functions in Generated Code

This example shows how to associate subsystems in a model with function names and files.

Learn how to:

- Specify function and file names in the generated code.
- Identify the parts of the generated code that are required for integration.
- Generate code for atomic subsystems.
- Identify data that are required to execute a generated function.

For information about the example model and other examples in this series, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Atomic and Virtual Subsystems

The example models in “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29 use *virtual subsystems*. Virtual subsystems visually organize blocks but do not affect the model functionality. *Atomic subsystems* evaluate the blocks included in a model as a unit. With atomic subsystems, you can specify additional function partitioning information. In a model, atomic subsystems appear with a bold border.

View Changes in Model Architecture

Open the example model rtwdemo_PCG_Eval_P3.

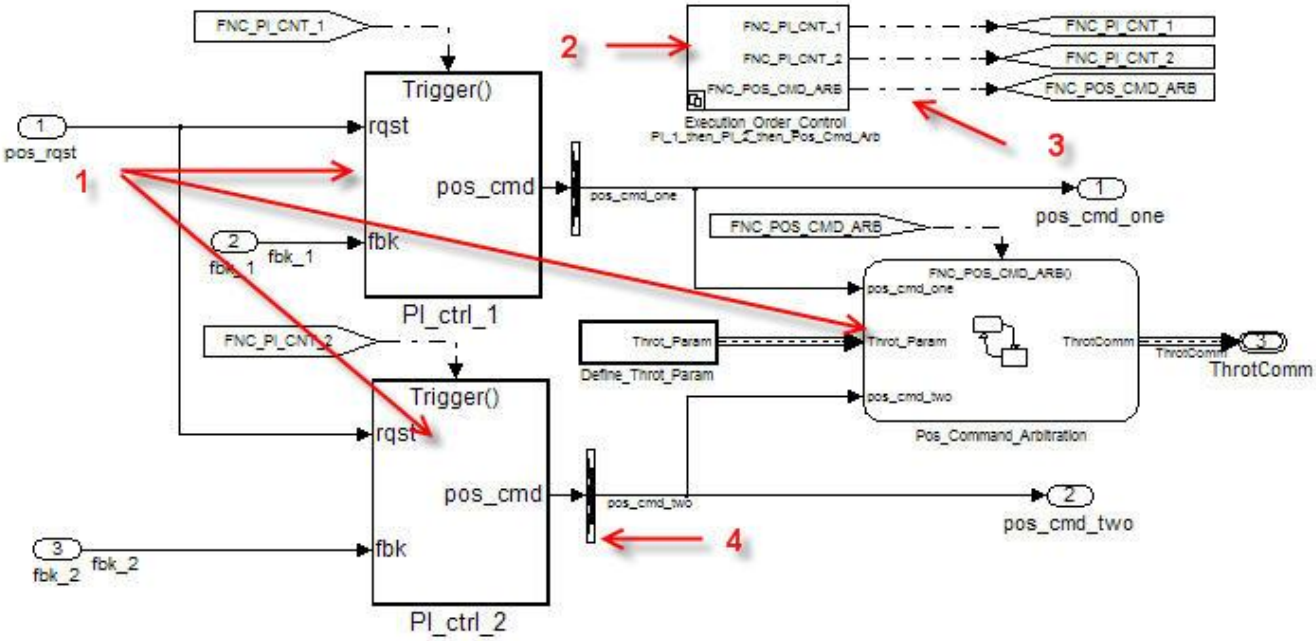
Save a copy of the model to a writable folder.

This example shows how to replace the virtual subsystems with *function-call subsystems*. Function-call subsystems:

- Are atomic subsystems
- Enable you to control subsystem execution order
- Execute when a *function call signal* triggers

By controlling the execution order of the subsystems, you can match the model with an existing system that has a specific execution order.

The figure identifies the function call subsystems (1) `PI_ctrl_1`, `PI_ctrl_2`, and `Pos_Command_Arbitration`.



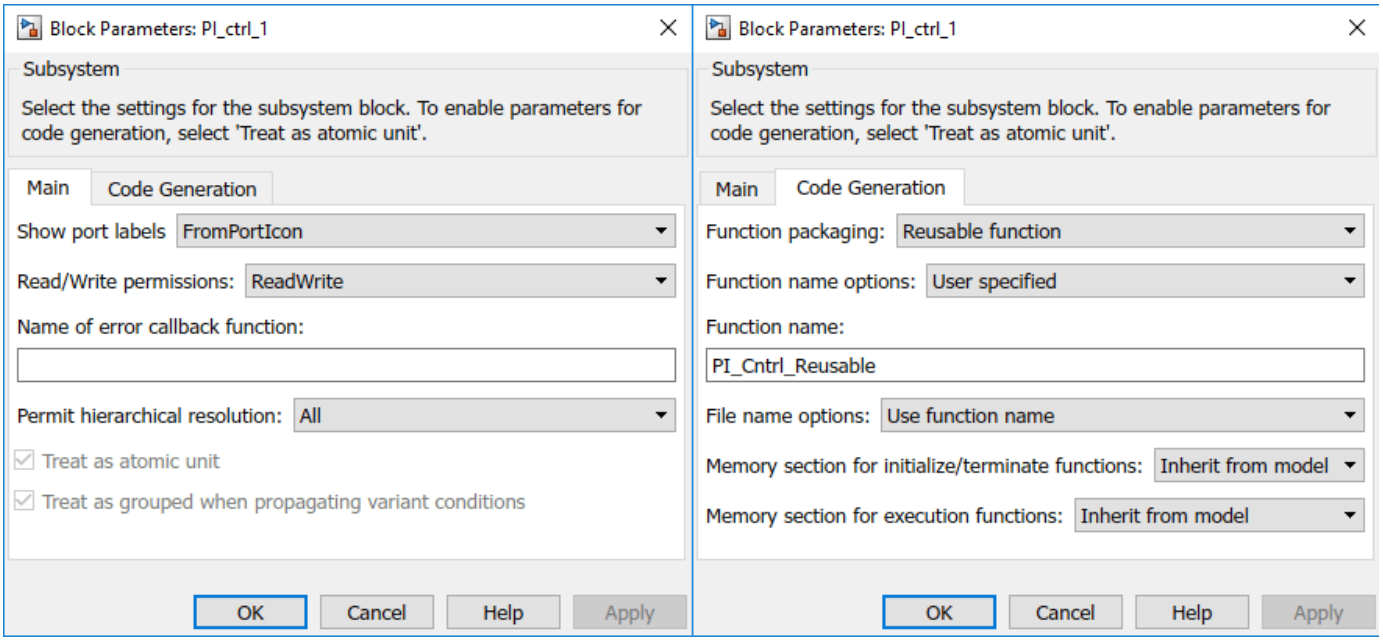
This version of the model contains the new subsystem `Execution_Order_Control` (2), which contains a Stateflow® chart that models the calling functionality of a scheduler. The subsystem controls the execution order of the function call subsystems through function call signals (3). Later in this example, you examine how changing the execution order can change the simulation results.

This version of the model contains new Signal Conversion blocks (4) at the outputs of the PI controllers. With these additional blocks in place, the code generator can generate a single reentrant function for the PI controllers.

Control Function Location and File Placement in the Generated Code

In “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29, the code generator creates a single *`model`*`_step` function that contains the control algorithm code. However, many applications require a greater level of control over the file placement of functions. By modifying the parameters of atomic subsystems, you can specify multiple functions within a single model.

The figure shows the subsystem parameters for `PI_ctrl_1`.



Treat as atomic unit

- Enables other submenus. For atomic subsystems, this parameter is automatically selected and disabled.

Sample time

- Specifies a sample time for execution. Not available for function-call subsystems.

Function packaging options

- `Auto` -- Determines how the subsystem appears in the generated code. This value is the default.
- `Inline` -- Places the subsystem code inline with the rest of the model code.
- `Function` -- Generates the code for the subsystem as a function.
- `Reusable function` -- Generates a reusable (reentrant) function from the subsystem. The function passes all input and output data through formal parameters. The function does not directly access global variables.

Function name options

- Selecting `Function` or `Reusable function` for **Function packaging** enables function name options.
- `Auto` -- Determines the function.
- `Use subsystem name` -- Bases the function on the subsystem name.
- `User specified` -- Applies the specified file name.

File name options

- Selecting `Function` or `Reusable function` for **Function packaging** enables file name options.
- `Auto` -- Places the function definition in the module generated for the parent system, or, if the model root is the parent, in `model.c`.

- `Use subsystem name` -- Generates a separate file. The name of the file is the name of the subsystem or library block.
- `Use function name` -- Generates a separate file. The name of the file is the name that you specify with **Function name options**.
- `User specified` -- Applies the specified, unique file name.

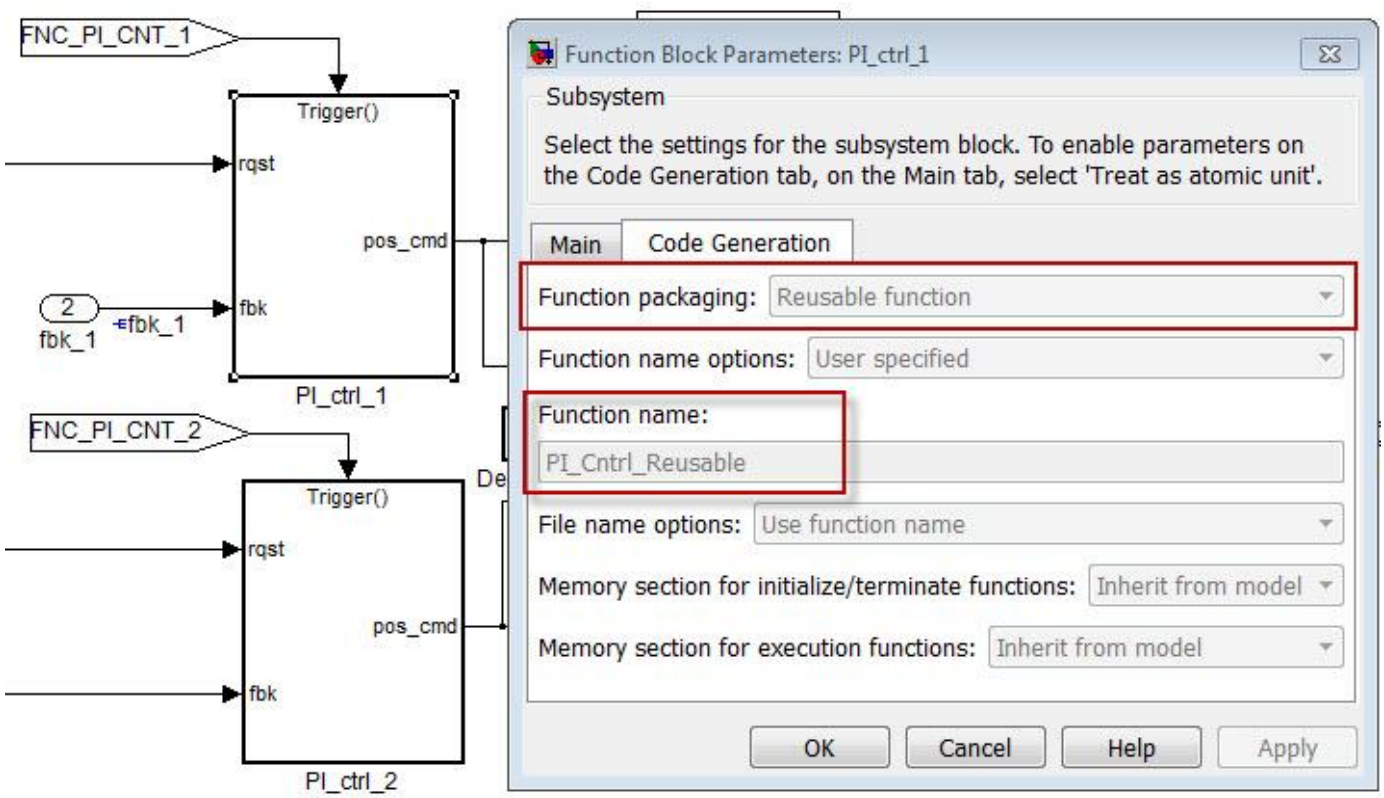
Function with separate data

- Enabled when you set **Function packaging** to `Function`. When selected, the code generator separates the internal data of the subsystem (for example, signals) from the data of the parent model. The subsystem owns this separate data.

Generate Reentrant Code

Embedded Coder® supports *reentrant code*. Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Reentrant code is used in operating systems and other system software that uses multithreading to handle concurrent events. Reentrant code does not maintain state data, so there are no persistent variables in the function. Calling programs maintain state variables and must pass the state data into the function. Multiple users or processes can share one copy of a reentrant function.

To generate reentrant code, you must first specify the subsystem as reusable by configuring the subsystem parameter **Function packaging**.



In some cases, the configuration of the model prevents reusable code. The table lists common issues.

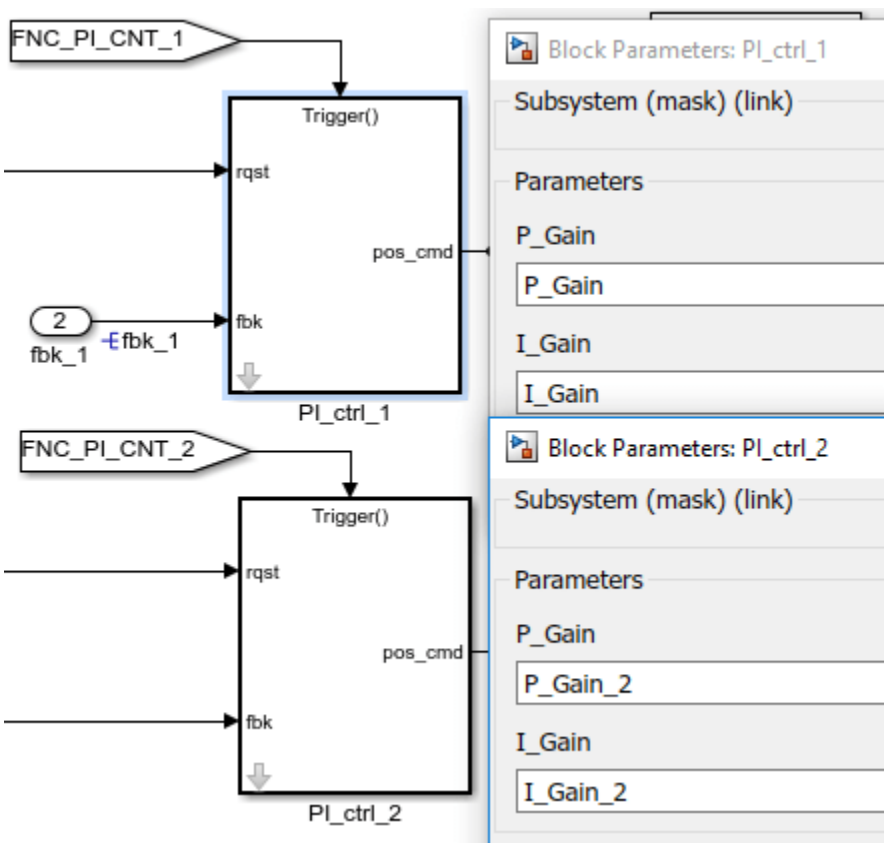
Cause	Solution

`Subsystem output feeds global signal data`	`Add a Signal Conversion block between the subsystem and the global signal.`
`Generated function receives data (formal parameters) through pointers`	`Select Configuration Parameters > Model Referencing > Pass fixed-size scalar root inputs by value for code generation.`
`Subsystem uses global signal data in internal algorithm`	`Use a port to pass the global data in and out of the subsystem.`

Use a Mask to Pass Parameter Values into Library Subsystem

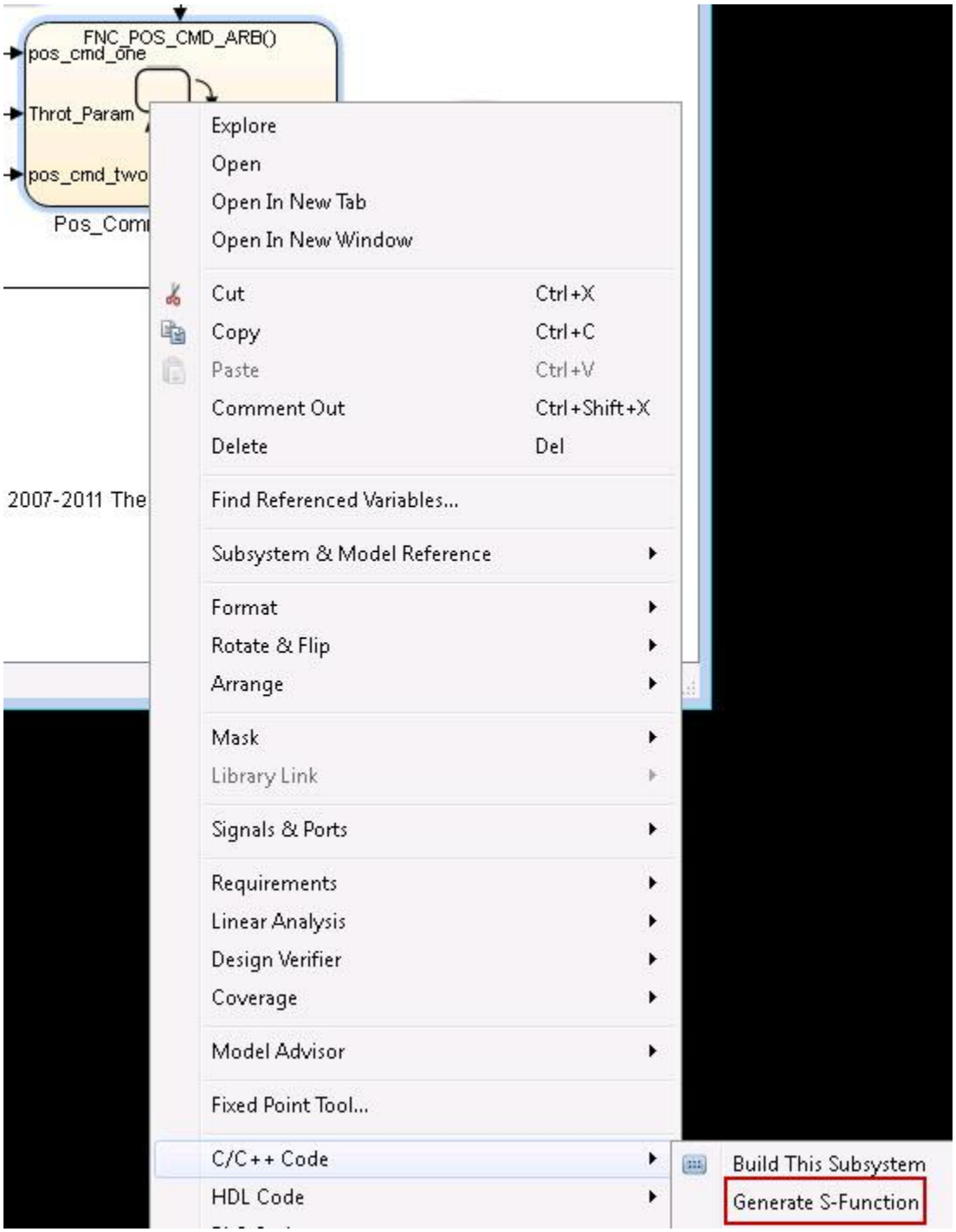
To define algorithmic parameter data (such as a gain or coefficient) outside the scope of a reusable library block or subsystem, you can apply a *mask* to the block or subsystem and create a mask parameter. You can then specify a different parameter value for each instance of the block or subsystem. Each mask parameter appears in the generated code as a formal parameter of the reentrant function.

In this version of the model, the subsystems `PI_ctrl_1` and `PI_ctrl_2` are masked. In each mask, the values of the `P` and `I` gains are set by data objects such as `I_Gain_2` and `P_Gain_2`.



Generate Code for Atomic Subsystem

In “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29, you generate code at the root level of the model. Alternatively, you can build a specific subsystem.



To initiate a subsystem build, use the context menu. You can choose from these options:

1. **Build This Subsystem**: Treats the subsystem as a separate mode and creates the full set of source C files and header files. This option does not support function-call subsystems.
2. **Generate S-Function**: Generates C code for the subsystem and creates an S-Function wrapper. You can then simulate the code in the original model. This option does not support function-call subsystems.

3. **Export Functions**: Generates C code without the scheduling code that comes with the **Build This Subsystem** option. Use this option to build subsystems that use triggers, such as function-call subsystems.

Alternatively, open the **Embedded Coder** app, select the subsystem and on the **C Code** tab, click **Build**.

Examine Generated Code

This example compares the files that are generated for the full system build with the files that are generated for exported functions. You also examine how the masked data appears in the code.

Run the build script for the three options. Then, examine the generated files by clicking the hyperlinks.

1. Generate code from the entire model.
2. Export the function `PI_ctrl_1`.
3. Export the function `Pos_Command_Arbitration`.

rtwdemo_PCG_Eval_P3.c

- Full Build: Yes, Step function
- PI_ctrl_1: No
- Pos_Command_Arbitration: No

PI_ctrl_1.c

- Full Build: No
- PI_ctrl_1: Yes, Trigger function
- Pos_Command_Arbitration: No

Pos_Command_Arbitration.c

- Full Build: No
- PI_ctrl_1: No
- Pos_Command_Arbitration: Yes, Init and Function

PI_Ctrl_Reusable.c

- Full Build: Yes
- PI_ctrl_1: Yes
- Pos_Command_Arbitration: No

ert_main.c

- Full Build: Yes
- PI_ctrl_1: Yes
- Pos_Command_Arbitration: Yes

eval_data.c

- Full Build: Yes(1)
- PI_ctrl_1: Yes(1)
- Pos_Command_Arbitration: No, Eval data not used in diagram

(1) `eval_data.c` has different content in the full and export function builds. The full build includes all of the parameters that the model uses. The export function contains only the variables that the subsystem uses.

Masked Data in the Generated Code

In the file `rtwdemo_PCG_Eval_P3.c`, the call sites of the reentrant function use the data objects `P_Gain`, `I_Gain`, `P_Gain_2`, and `I_Gain_2` as arguments.

```
PI_Cntrl_Reusable(*pos_rqst, fbk_1, &rtwdemo_PCG_Eval_P3_B->PI_ctrl_1,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl_1, I_Gain, P_Gain);
PI_Cntrl_Reusable(*pos_rqst, fbk_2, &rtwdemo_PCG_Eval_P3_B->PI_ctrl_2,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl_2, I_Gain_2, P_Gain_2);
```

Effect of Execution Order on Simulation Results

By default, Simulink® executes the subsystems in this order:

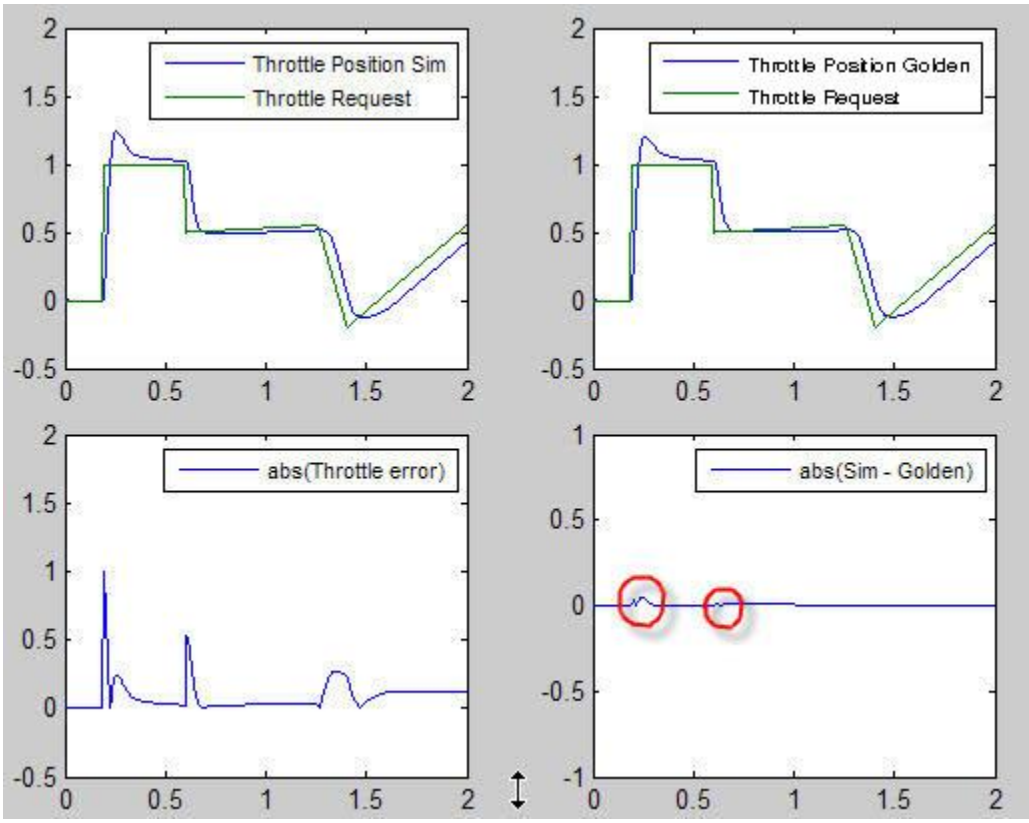
1. `PI_ctrl_1`
2. `PI_ctrl_2`
3. `Pos_Command_Arbitration`

For this example, you can specify one of two alternative orders of execution. You can then use the test harness to observe the effect of the execution order on the simulation results. The subsystem `Execution_Order_Control` has two configurations that control the execution order. To choose a configuration, use the subsystem context menu.

Change the execution order and observe the results.

1. Set the execution order to `PI_ctrl_1`, `PI_ctrl_2`, `Pos_Command_Arbitration`.
2. Open the test harness.
3. Run the test harness.
4. Change the execution order to `Pos_Command_Arbitration`, `PI_ctrl_1`, `PI_ctrl_2`.
5. Run the test harness.

The simulation results (throttle position over time) vary slightly depending on the order of execution. You can see the difference most clearly when the throttle request changes.



For the next example in this series, see “Call External C Code from Model and Generated Code” on page 53-48.

Nonvirtual Subsystem Modular Function Code Limitations

Nonvirtual subsystem block parameter **Function with separate data** has the following limitations:

- The parameter is available for models configured with an ERT-based system target file.
- The nonvirtual subsystem to which the parameter is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain noninlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as `model.h` and `model_private.h`.
- The parameter is incompatible with the **Classic call interface** parameter. Selecting both parameters generates an error.
- The parameter is incompatible with the `Reusable function` setting for model configuration parameter **Code interface packaging**. Selecting both parameters generates an error.
- When you select the parameter for a subsystem, the model that contains the subsystem cannot contain a Data Store Memory block with **Share across model instances** selected. See Data Store Memory.

See Also

More About

- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3

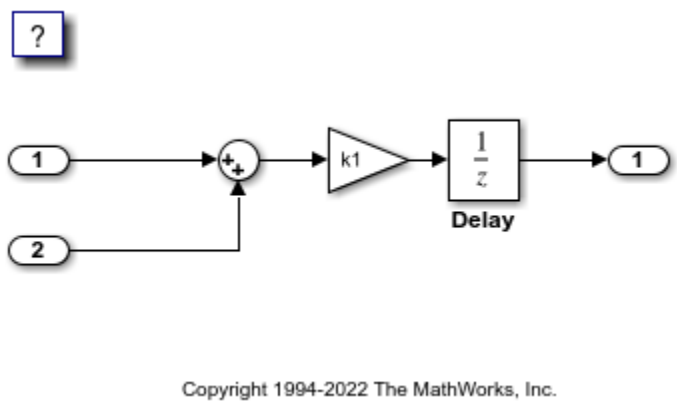
Generate Reentrant, Multi-Instance Code

This example shows you how to configure a model for reentrant, multi-instance code generation. Multiple programs can use reentrant code simultaneously. When you configure a model for reentrancy, the execution (step) entry-point function uses root-level input and output arguments instead of global data structures. After examining the configuration settings, generate and review the generated code.

Open the Model

Open the model `Reusable`. The model contains two root Inport blocks and a root Outport block.

```
model='Reusable';
open_system(model);
```



Examine Relevant Model Configuration Parameter Settings

1. Open the **Embedded Coder** app.

2. Open the Model Configuration Parameters dialog box.

3. Model configuration parameter **System target file** is set to `ert.tlc`. Although you can generate reentrant code for a model configured with **System target file** set to `grt.tlc`, ERT and ERT-based system target files provide more control over how the code passes root-level I/O.

4. Open the **Code Generation > Interface** pane and explore relevant model configuration parameter settings.

- **Code interface packaging** is set to `Reusable function`. This parameter setting instructs the code generator to produce reusable, multi-instance code.
- Setting of `Reusable function` displays parameter **Multi-instance code error diagnostic**. That parameter is set to `Error`, indicating that the code generator abort if the model violates requirements for generating multi-instance code.
- **Pass root-level I/O as** is set to `Part of model data structure`. This setting packages root-level model input and output into the real-time model data structure (`rtModel`), which is an optimized data structure that replaces `SimStruct` as the top-level data structure for a model.

- **Remove error status field in real-time model data structure** is selected. This parameter setting reduces memory usage by omitting the error status field from the generated real-time model data structure.

Generate and Review Code

```
slbuild(model);
```

```
### Starting build procedure for: Reusable
### Successful completion of build procedure for: Reusable

Build Summary

Top model targets built:

Model     Action                        Rebuild Reason
========================================================================================
Reusable  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.936s
```

Review the Generated Code

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point function `Reusable_step`. Use this file as a starting point for coding your execution framework.
- `Reusable.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `Reusable.h` declare model data structures and a public interface to the model entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework.

1. Include the generated header file by adding directive `#include Reusable.h`.

2. Write input data to the generated code for model Inport blocks.

3. Call the generated entry-point functions.

4. Read data from the generated code for the model Outport block.

Input ports:

- `<Root>/In1` of data `real_T` with dimension of 1
- `<Root>/In2` of data `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void Reusable_initialize(RT_MODEL *const rtM)`. At startup, call this function once.

- Output and update (step) entry-point function, `void Reusable_step(RT_MODEL *const rtM)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.

Output port:

- `<Root>/Out1` of data type `real_T` with dimension of 1

Examine the Step Function

```
Examine the |Reusable_step| function code in |Reusable.c|.
```

```
cfile = fullfile('Reusable_ert_rtw','Reusable.c');
rtwdemodbtype(cfile,'/* Model step function', '/* Model initialize function ', 1, 0);
```

```
/* Model step function */
void Reusable_step(RT_MODEL *const rtM)
{
  D_Work *rtDWork = rtM->dwork;
  ExternalInputs *rtU = (ExternalInputs *) rtM->inputs;
  ExternalOutputs *rtY = (ExternalOutputs *) rtM->outputs;

  /* Outport: '<Root>/Out1' incorporates:
   *  UnitDelay: '<Root>/Delay'
   */
  rtY->Out1 = rtDWork->Delay_DSTATE;

  /* Gain: '<Root>/Gain' incorporates:
   *  Sum: '<Root>/Sum'
   *  UnitDelay: '<Root>/Delay'
   */
  rtDWork->Delay_DSTATE = (rtU->In1 + rtU->In2) * rtP.k1;
}
```

The code generator passes model data to the `Reusable_step` function as part of the real-time model data structure. Try different settings for model configuration parameters **Code interface packaging** and **Pass root-level I/O** and regenerate code. Observe how the function prototype changes.

Close the Model and Report

Close the model and the code generation report.

```
bdclose(model)
```

See Also

Related Examples

- “Generate Reentrant Code from Top Models” on page 8-13

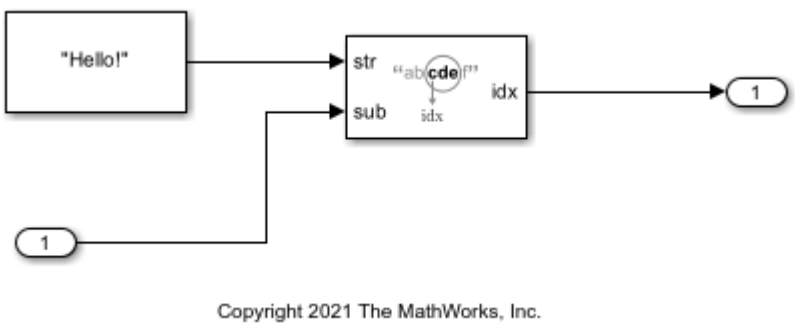
Generate Code for String Blocks by Using the Standard C++ String Library

Optimize generated code for Simulink® string blocks by using standard C++ library string functions and string objects. The C++ string library provides consistent C++ code and improves functionality, such as string length retrieval, concatenation, string copy, string swapping, string comparison, and substring computations.

Example Model

In the model matlab:mStrfindSubStr, a string constant block that has the value "Hello!" connects to the `str` port of a String Find block. An inport connects to the `sub` port of the same String Find block.

```
model = 'mStrfindSubStr';
open_system(model);
```



In your system's temporary folder, create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Generate Code by Using C++ String Libraries

In the Configuration Parameters dialog box, set the **Language** parameter to `C++`.

```
set_param(model, 'TargetLang', 'C++');
```

In the Configuration Parameters dialog box, set the **Code interface packaging** parameter to `C++ class`.

```
set_param(model, 'CodeInterfacePackaging', 'C++ class');
```

In the Configuration Parameters dialog box, set the **Standard math library** parameter to `C++03 (ISO)`.

```
set_param(model, 'TargetLangStandard', 'C++03 (ISO)');
```

Build the model.

```
evalc('slbuild(model)');
```

View the generated code with C++ string library objects. Here is a portion of `mStrfindSubStr.h`. The code instantiates a std:string object `In1`.

```
cfile = fullfile(cgDir,'mStrfindSubStr_ert_rtw','mStrfindSubStr.h');
rtwdemodbtype(cfile,'External inputs','External outputs',0,0);
```

```
struct ExtU_mStrfindSubStr_T {
  std::string In1;                      // '<Root>/In1'
};
```

View the generated code with C++ string library functions. Here is a portion of `mStrfindSubStr.cpp`. The code uses the `find` function to look for the first character match, which is present in the C++ string library.

```
cfile = fullfile(cgDir,'mStrfindSubStr_ert_rtw','mStrfindSubStr.cpp');
rtwdemodbtype(cfile,'Model step function','Model initialize function',0,0);
```

```
void untitled1ModelClass::step()
{
  uint32_T tmpOut;

  // StringFind: '<Root>/String Find' incorporates:
  //   Inport: '<Root>/In1'
  //   StringConstant: '<Root>/String Constant'

  tmpOut = mStrfindSubStr_P.StringConstant_String.find(mStrfindSubStr_U.In1);
  if (tmpOut == (uint32_T)std::string::npos) {
    // Outport: '<Root>/Out1'
    mStrfindSubStr_Y.Out1 = -1;
  } else {
    // Outport: '<Root>/Out1'
    mStrfindSubStr_Y.Out1 = static_cast<int32_T>(tmpOut) + 1;
  }

  // End of StringFind: '<Root>/String Find'
}
```

This code executes faster and is easier to read. Close the model and the code generation report.

```
bdclose(model);
rtwdemoclean;
cd(currentDir);
```

Limitations:

An array of a bus that contains std:string is not supported in software-in-loop (SIL) and processor-in-loop (PIL) simulations.

See Also

More About

- “Document Generated Code with Simulink Report Generator” on page 54-38
- Use dynamic memory allocation for model initialization

- Use dynamic memory allocation for model block instantiation
- Remove root level I/O zero initialization
- Remove internal data zero initialization

Select Code Generation Output for Target Platform Deployment

In this section...

"Example Model Hierarchy" on page 43-25
"Select Output for Simulation Harness" on page 43-25
"Specify Code Generation Output" on page 43-26
"Select C Code Interface Configuration" on page 43-27

When you want to generate code for a specific target platform, you configure the model at a high level by selecting the code output and, for C code, the code interface configuration. These high-level code output options determine which configuration parameters and code interface options are available for your model.

Example Model Hierarchy

For this example, open the model `RollAxisAutopilotHarness`.

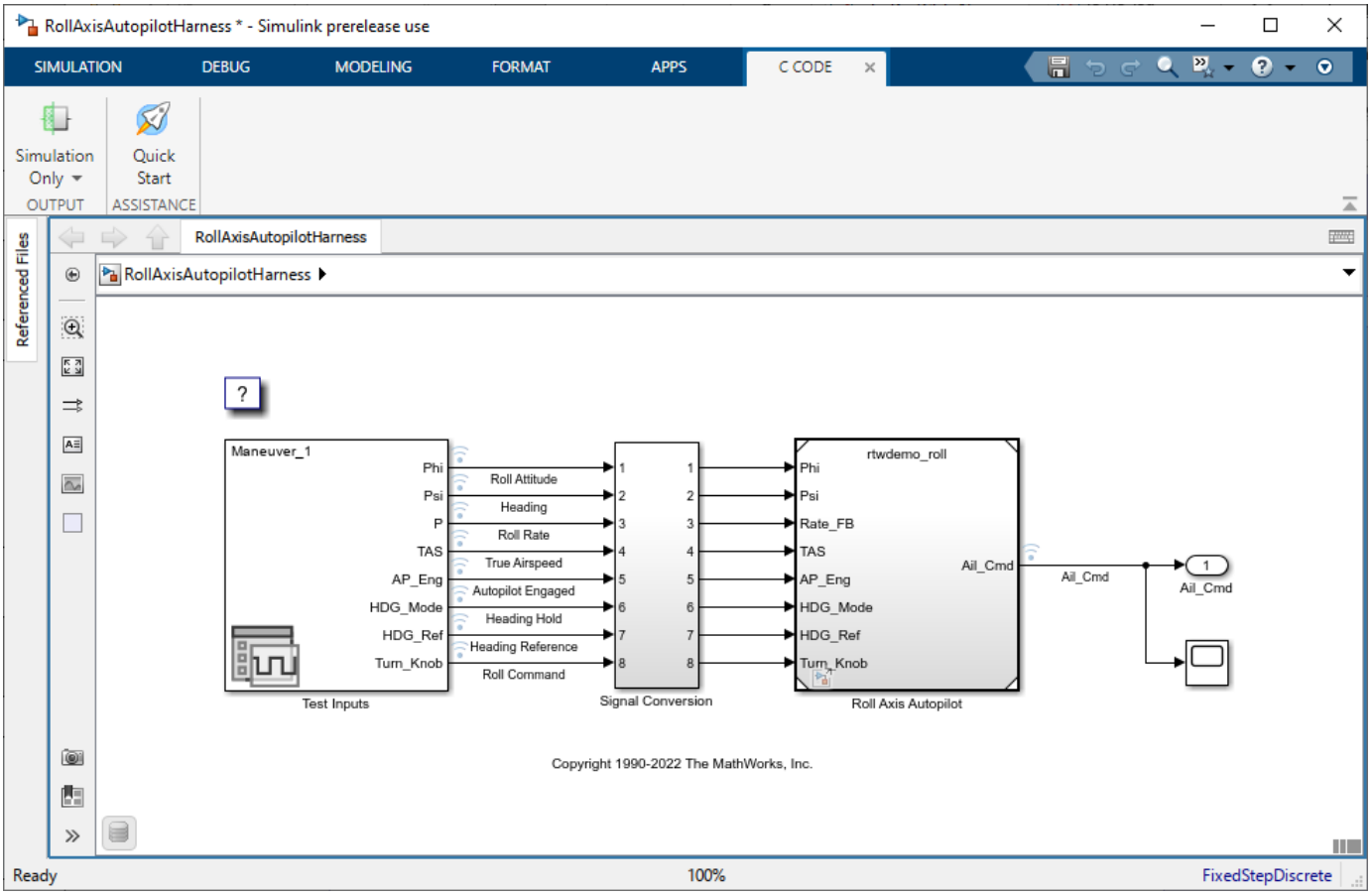
```
openExample('RollAxisAutopilotHarness')
```

The top model is a harness model that generates test inputs for the component model `RollAxisAutopilot`. The harness model is for simulation only. You generate code for the model `RollAxisAutopilot`.

Select Output for Simulation Harness

For the simulation harness model, specify the output as simulation only. This specification indicates to the code generator to not generate code for this model.

1. Navigate to the top model `RollAxisAutopilotHarness`.
2. Open the Embedded Coder app.
3. On the **C Code** tab, from the **Output** drop-down list, click **Simulation Only**.



The **C Code** tab hides the code generation actions because you do not generate code for this model. The code generation actions appear when you navigate into a model that is configured for code generation.

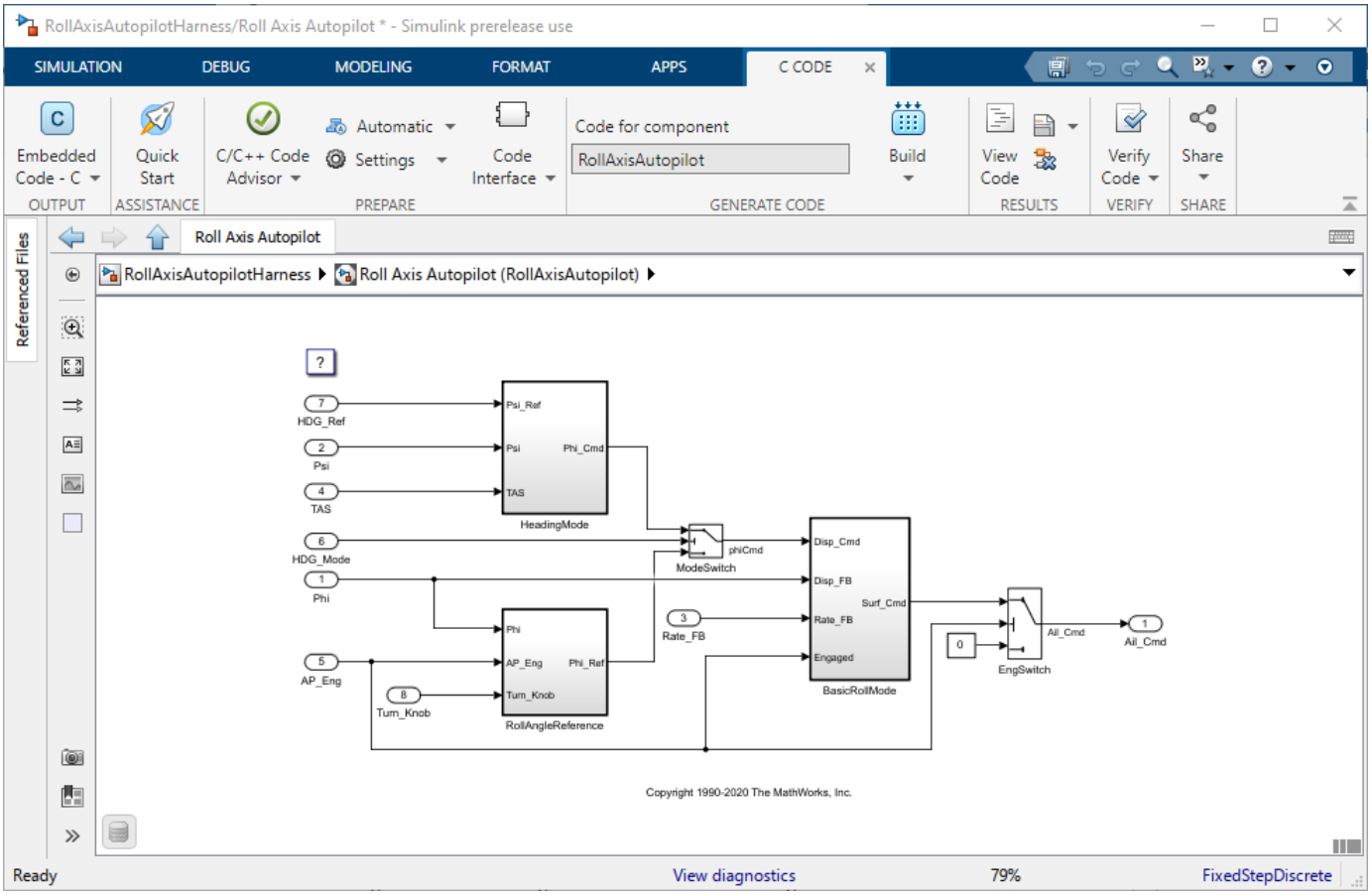
Specify Code Generation Output

To generate code for a target platform, specify the target and language as the code generation output for a model. You can specify these code generation output types:

- **Embedded Code − C** — Generate a C to run on the target device. The generated code is either a standalone program that runs on the target or a component that the target middleware runs, depending on the code interface configuration.
- **Embedded Code − C++** — Generate C++ code to run on the target device.
- **Shared Library** — Generate shared library code.

For the model `RollAxisAutopilot`, select the output **Embedded Code − C**.

1. Navigate to the referenced model `RollAxisAutopilot`.
2. On the **C Code** tab, from the **Output** drop-down list, click **Embedded Code − C**.



Next, configure the model to use a code interface configuration to generate code for the target platform.

Select C Code Interface Configuration

When you generate C code, you can generate code for a specific target platform by selecting a code interface configuration. A code interface configuration defines the interfaces in the generated code and controls how the code interacts with the target platform on which you run the code. You select one of these types of code interface configurations:

- Data interface configuration — The code generator creates a standalone executable program intended to run on the target device. The generated code includes the scheduler and services for the target device and uses the data interfaces that you define.
- Service interface configuration — The code generator creates an algorithm that you intend to deploy within a larger application. The platform middleware calls the algorithm code and provides the services for the target device. The generated code calls the services according to your definitions in the service interface configuration. To create a service interface configuration, you must use an Embedded Coder Dictionary that is stored in a Simulink data dictionary.

To apply a code interface configuration, select the Embedded Coder Dictionary that contains the configuration. For this example, configure the component model `RollAxisAutopilot` to use the dictionary and service code interface configuration that you create in “Create a Service Interface Configuration” on page 32-11.

1. Navigate to the referenced model `RollAxisAutopilot`.
2. On the **C Code** tab, from the **Code Interface** drop-down list, click **Set up shared Embedded Coder Dictionary**.
3. Select an .SLDD file that contains your Embedded Coder Dictionary. For this example, select `InterfaceCoderDictionary.sldd`, which you create in “Create a Service Interface Configuration” on page 32-11. Click **Open** and **Next**.

The Embedded Coder Dictionary provides the code interface configuration for the model. You can configure the code interface in more detail by mapping model elements to code definitions from the interface configuration.

See Also

Related Examples

- “Configure C Code Deployment Types for Model Hierarchy” on page 43-29
- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

Configure C Code Deployment Types for Model Hierarchy

When you develop a complex system by using Model-Based Design, models have different requirements for the resulting code to run on the target platform. A *deployment type* indicates how the generated code for a model interacts with the target platform and code generated from other models. The deployment type also indicates the role of a model in the system and determines the composition rules for using the model in a hierarchy with other models. You can set the deployment type of each model in a system by using the Embedded Coder app or the code mappings programmatic interface. The deployment types are:

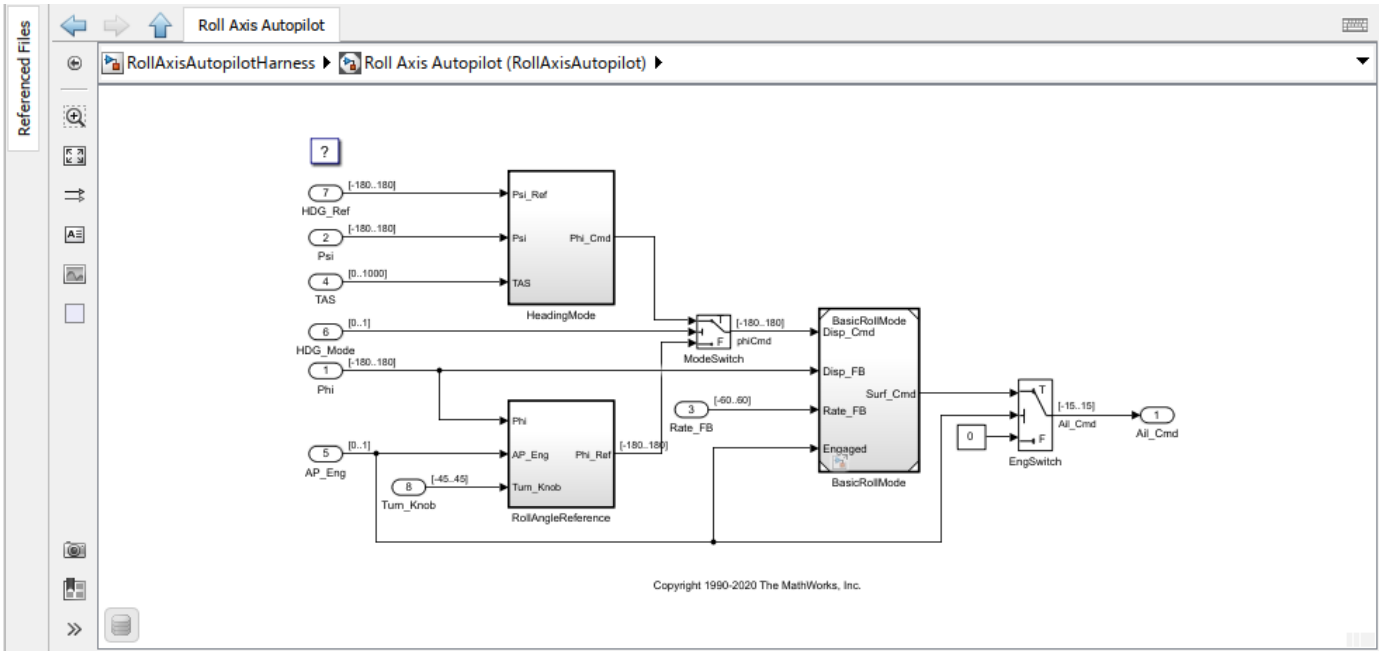
- Component — The top model from which the code generator produces code. The code generator produces a standalone algorithm that runs on the target platform. The component code exposes its interface to other components in the system and to platform services.
- Subcomponent — A model reference that a component model uses. The generated code entry points are symbolically scoped to the parent component and are not exposed to the platform.
- Automatic — Embedded Coder determines the deployment type based on the model hierarchy context.

Embedded Coder uses the deployment type of a model to map elements of the model to interfaces in the code interface configuration. This example shows you how to configure the deployment types for several models in a system and generate code to run on the target platform.

Set Up Test Harness and Model Hierarchy

The model `RollAxisAutopilotHarness` is a harness model that generates test inputs for the component model `RollAxisAutopilot`. The harness model is for simulation only. You generate code for the model `RollAxisAutopilot`. For this example, you add a referenced model to the `RollAxisAutopilot` model to represent a subcomponent in the generated code.

1. Open the model `RollAxisAutopilotHarness`. At the command line, enter `openExample('RollAxisAutopilotHarness')`.
2. View the component model `RollAxisAutopilot` by double-clicking the Model block.
3. Convert the subsystem `BasicRollMode` to a referenced model. Right-click the Subsystem block and click **Subsystem & Model Reference** > **Convert to** > **Referenced Model**. The Model Reference Conversion Advisor opens. Click **Convert**.



The model hierarchy now contains these models:

- `RollAxisAutopilotHarness` — The test harness that generates inputs for `RollAxisAutopilot`. You simulate this model. You do not generate code from this model.
- `RollAxisAutopilot` — The component model from which the code generator generates code for the target platform.
- `BasicRollMode` — A subcomponent for which the code generator produces code in the context of the component model `RollAxisAutopilot`.

To generate code according to this hierarchy, configure the deployment types for these models.

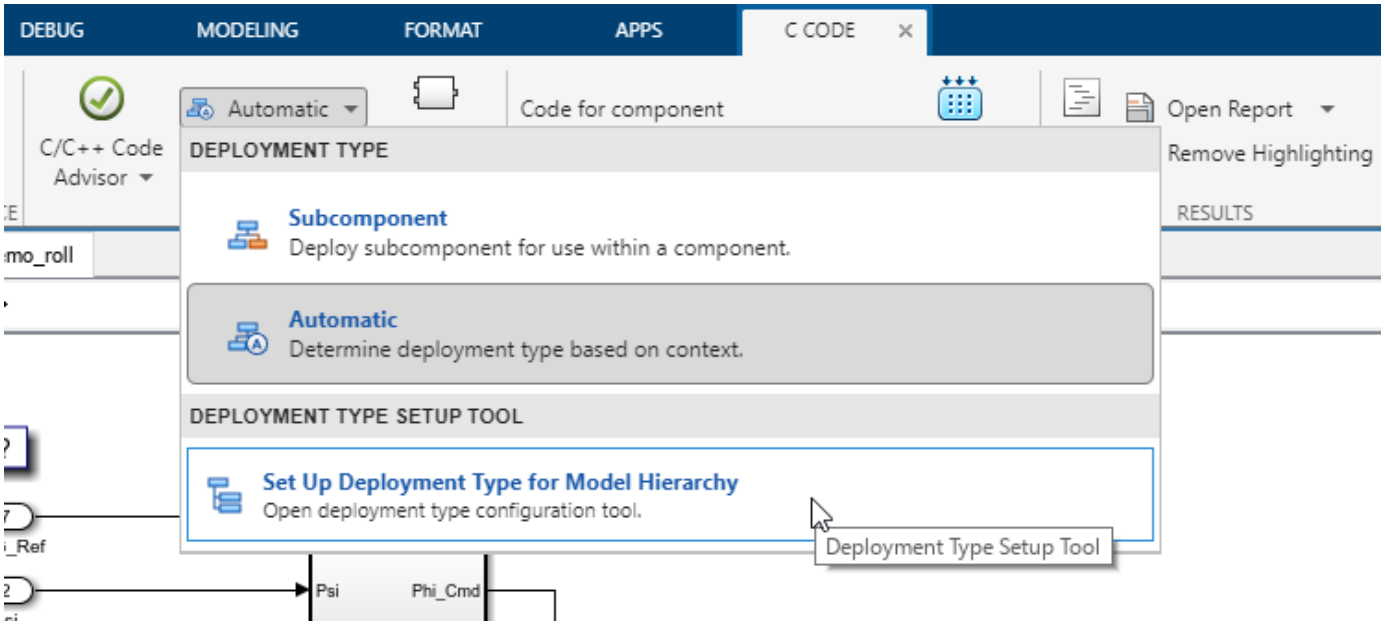
Specify Deployment Types for the Models

You specify the deployment types of the models in one of these ways:

- For each model, on the **C Code** tab, select the deployment type from the deployment type options. Select a deployment type from the **Deployment Type** menu or, for non-deployable models, select the **Simulation only** option from the **Output** menu.
- Configure the models in a hierarchy by using the deployment configuration tool.
- Use the code mappings programmatic interface.

For this example, use the deployment configuration tool and set the deployment types for the whole model hierarchy.

1. With the `RollAxisAutopilotHarness` model open, open the **Embedded Coder** app.
2. On the **C Code** tab, in the **Prepare** section, click the **Automatic** button. This button shows the deployment type for the model that is in the canvas. **Automatic** indicates that Embedded Coder determines the deployment type based on the model context.



3 From the **Deployment Type** menu, select **Set Up Deployment Type for Model Hierarchy**.

4 In the Set up deployment type for model hierarchy dialog box, expand the `RollAxisAutopilotHarness` and `RollAxisAutopilot` models to see the model hierarchy.

5 Specify the `RollAxisAutopilotHarness` model as simulation only. In the row for the model, clear the check box for the **Deployable** column. This configures the model for simulation only. The code generator does not produce code from this model.

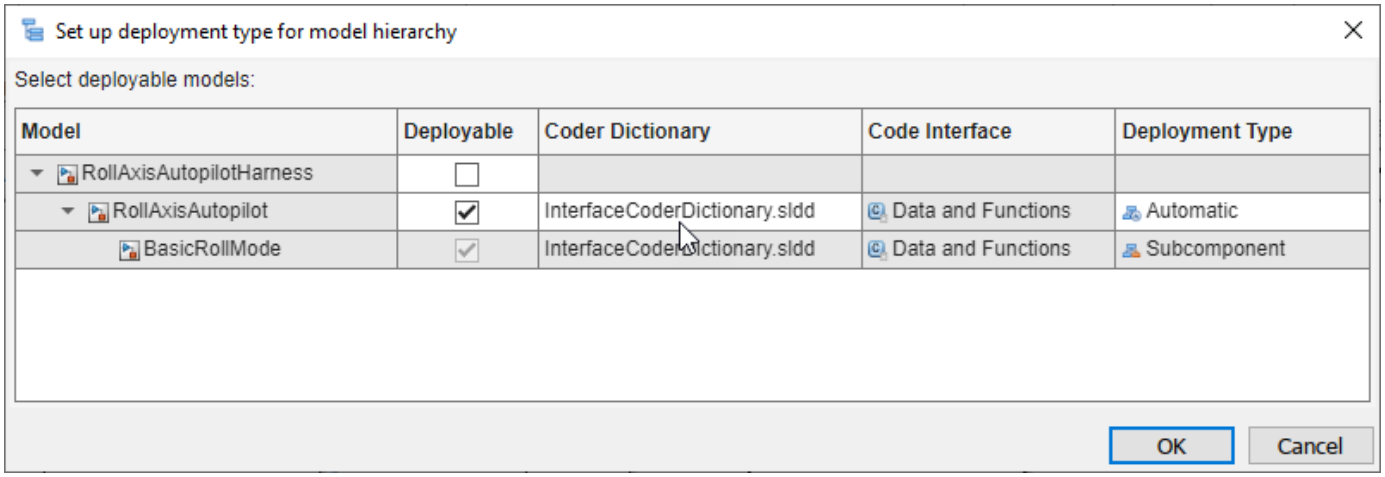
6 For the deployable models, the code interface configuration in the specified Coder Dictionary determines what deployment types are available.

- Data interface configuration — the **Automatic** and **Subcomponent** data types are available.
- Service interface configuration —the **Component** and **Subcomponent** data types are available.

For the `RollAxisAutopilot` model, specify the **Coder Dictionary** as an .SLDD file that contains a coder dictionary with a data interface configuration.

7 Specify the `RollAxisAutopilot` model as an automatically determined deployment type. In the row for the model, in the **Deployment Type** column, double-click the box and from the drop-down select **Automatic**.

8 Because `RollAxisAutopilot` is a top model, the `BasicRollMode` model is automatically configured as a subcomponent. For this model, the code generator produces subcomponent code, which is scoped to the component code for `RollAxisAutopilot`.



9 Click **OK**. On the **C Code** tab, the **Code for component** field shows the model for which you generate code.

Generate and Inspect Code

Generate code for the models. On the **C Code** tab, click **Build**. The Code view displays the generated code next to the model. To view code for a specific model, navigate to the model in the canvas. The Code view is updated to show the code for the currently displayed model. If you navigate to a simulation only model, such as `RollAxisAutopilotHarness`, the Code view continues to show code from the component model.

View the code for the component model `RollAxisAutopilot`. This code snippet shows some of the component interface that is generated for the model.

```
#include "RollAxisAutopilot.h"
#include <math.h>
#include "rtwtypes.h"
#include "BasicRollMode.h"

/* Block signals and states (default storage) */
DW rtDW;

/* External inputs (root inport signals with default storage) */
ExtU rtU;

/* External outputs (root outports fed by signals with default storage) */
ExtY rtY;

/* Model step function */
void RollAxisAutopilot_step(void)
{
  /* local block i/o variables */
  real32_T rtb_BasicRollMode;
  real32_T rtb_FixPtUnitDelay1;

  /* Outputs for Atomic SubSystem: '<Root>/RollAngleReference' */
  /* UnitDelay: '<S3>/FixPt Unit Delay1' */
  rtb_FixPtUnitDelay1 = rtDW.FixPtUnitDelay1_DSTATE;
```

View the code for the subcomponent model `BasicRollMode`. This code snippet shows some of the subcomponent interface that is generated for the model.

```
#include "BasicRollMode.h"
#include "rtwtypes.h"

/* Disable for referenced model: 'BasicRollMode' */
void BasicRollMode_Disable(BasicRollMode_DW_f *localDW)
{
  /* Disable for DiscreteIntegrator: '<Root>/Integrator' */
```

```
  localDW->Integrator_DSTATE = localDW->Integrator;
}

/* Output and update for referenced model: 'BasicRollMode' */
void BasicRollMode(const real32_T *rtu_Disp_Cmd, const real32_T *rtu_Disp_FB,
                   const real32_T *rtu_Rate_FB, const boolean_T *rtu_Engaged,
                   real32_T *rty_Surf_Cmd, BasicRollMode_DW_f *localDW)
{
  real32_T rtb_Sum;
  real32_T u0;
  boolean_T rtb_NotEngaged;
```

To further customize the interface of the generated code, use the Code Mappings editor to map model element categories and individual model elements to code definitions. The Code Mappings editor shows code definitions that are available to the model based on the deployment type.

See Also

Code Mappings Editor – C

Related Examples

- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15

Configure C++ Class Interface

- “Interactively Configure C++ Interface” on page 44-2
- “Programmatically Configure C++ Interface” on page 44-25
- “Configure C++ Class Interface for Rate-Based Models” on page 44-46
- “Configure C++ Class Interface for Export-Function Models” on page 44-50

Interactively Configure C++ Interface

To streamline the integration of C++ code generated from Simulink applications, components, and subsystems with external C or C++ code, configure a customized C++ class interface. When you generate C++ code from a model, the model appears as a class, the data elements appear as class members, and the model functions appear as class methods in the generated code. Configuring a C++ class interface enables you to customize the following aspects of the generated C++ code:

- Class information — Class name and namespace
- Class member information — Class member access and visibility
- Class method information — Class method names and arguments

A customized C++ class interface enables the generated classes to meet specific code standards or interface requirements so that the generated code can compile and integrate into larger architectures with minimal post-generation customization.

Interactive Workflow Overview

The interactive workflow provides quick, intuitive, and complete configuration of a customized C++ class interface by using tools accessible directly from the **C++ Code** tab. From the tab, you can click **Code Interface** and access the **Class Name & Namespace** configuration dialog box and Code Mappings editor (with Property Inspector) to configure an interface.

To interactively configure a customized C++ interface:

1. “Open Environment” on page 44-2
2. “Configure Model as Class” on page 44-4
3. “Configure Model Data Elements as Class Members” on page 44-5
4. “Configure Model Functions as Class Methods” on page 44-15
5. “Generate C++ Class Interface” on page 44-22

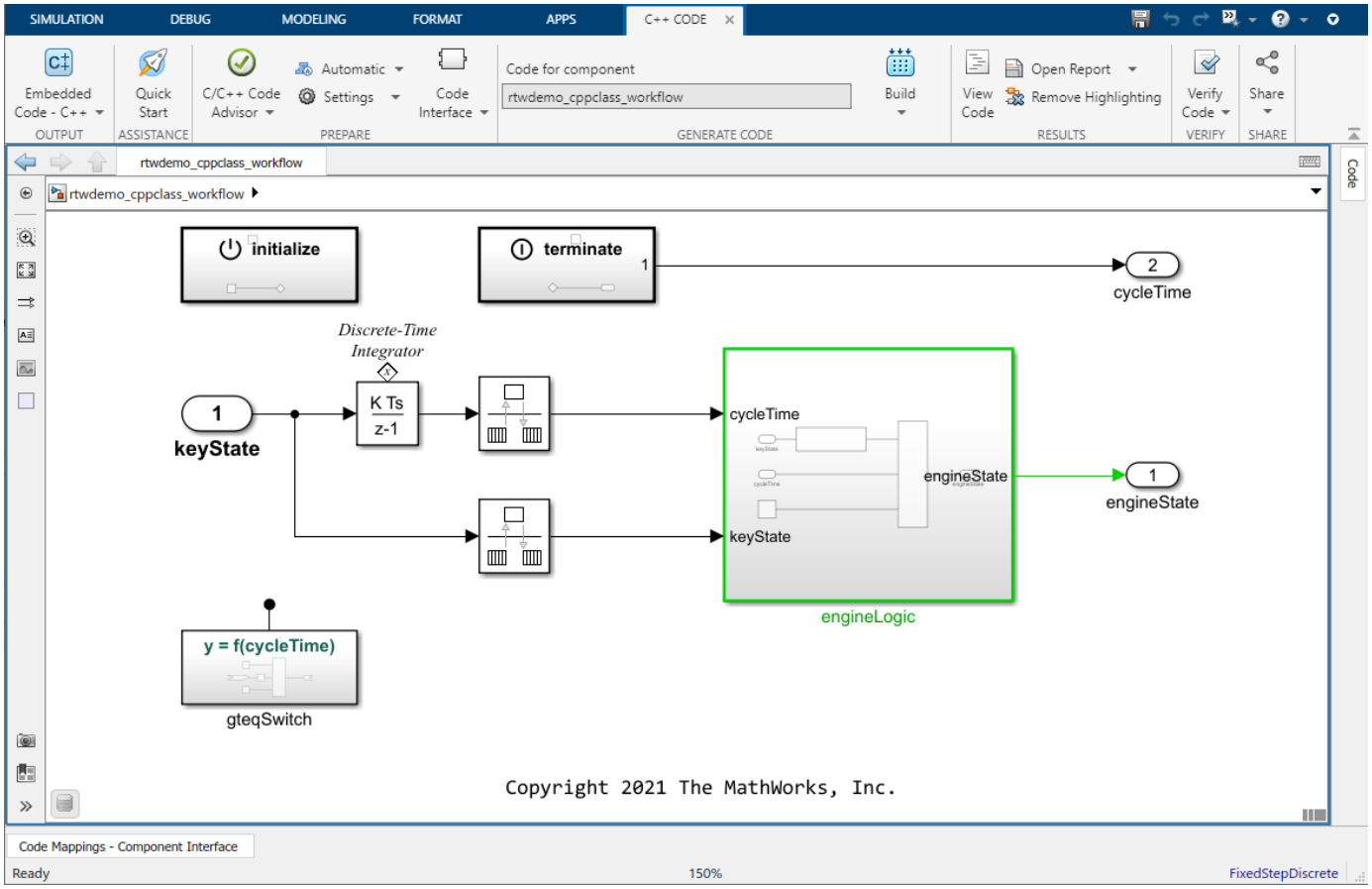
Open Environment

Open the environment for configuring a customized C++ class interface:

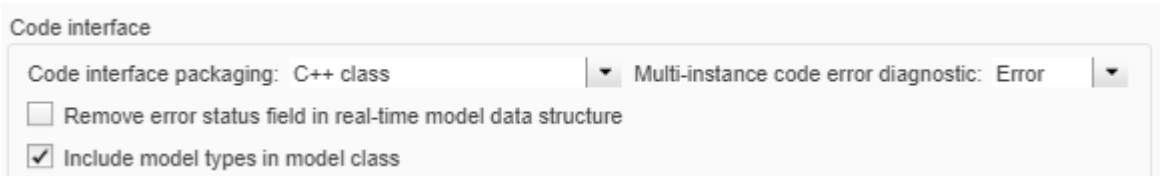
1. Open the model `CppClassWorkflowKeyIgnition`.

   ```
   openExample('ecoder/CppCodeGenerationWorkflowForKeyIgnitionSystemExample', ...
               supportingFile='CppClassWorkflowKeyIgnition.slx')
   ```

2. Open the Embedded Coder app. In the Apps gallery, click **Embedded Coder**.
3. Set the language to C++. On the **C++ Code** tab, click **Output** and select **Embedded C++ Code**.



4 Set the model configuration parameters. To open the Configuration Parameters dialog box, on the **C++ Code** tab, click **Settings**, and select **C/C++ Code generation settings**. These parameters configure model-wide code generation behavior. To configure parameters specific to generating a customized C++ class interface, you can set these parameters (located on the **Interface** pane):



Configuration Parameter	Description
Code interface packaging	Selects output language for generated code. For more information, see Code interface packaging.
Multi-instance code error diagnostic	Specifies the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code. For more information, see Multi-instance code error diagnostic.

Configuration Parameter	Description
Remove error status field in real-time model data structure	Specifies whether to omit the error status field from the generated real-time model data structure `rtModel`. For more information, see Remove error status field in real-time model data structure.
Include model types in model class	Specifies to include model type definitions within the class namespace of the model. For more information, see Include model types in model class.

Interface parameters that are related but are less commonly used appear under **Advanced parameters**.

Configuration Parameter	Description
Terminate function required	Specifies whether to generate the *`model`*`_terminate` method. For more information, see Terminate function required.
Combine signal/state structures	Specifies whether to combine global block signals and global state data into one data structure in the generated code. For more information, see Combine signal/state structures.
Generate destructor	Specifies whether to generate a destructor for the C++ model class. For more information, see Generate destructor.
Use dynamic memory allocation for model block instantiation	Specifies memory allocation for model hierarchies. For more information, seeUse dynamic memory allocation for model block instantiation.

Configure Model as Class

Configure the class name and namespace. When you generate C++ code from a model, that model appears as a class in the generated code. To ease integration and comply with code and interface requirements, you can customize the generated class name. Optionally, you can scope the generated code and prevent symbol clashes within a project by specifying a namespace for the generated class. In modeled systems constructed as a model hierarchy, you can specify a different namespace for each model in the hierarchy.

The generated C++ class interface, declared in the model header file, incorporates the customized name and namespace:

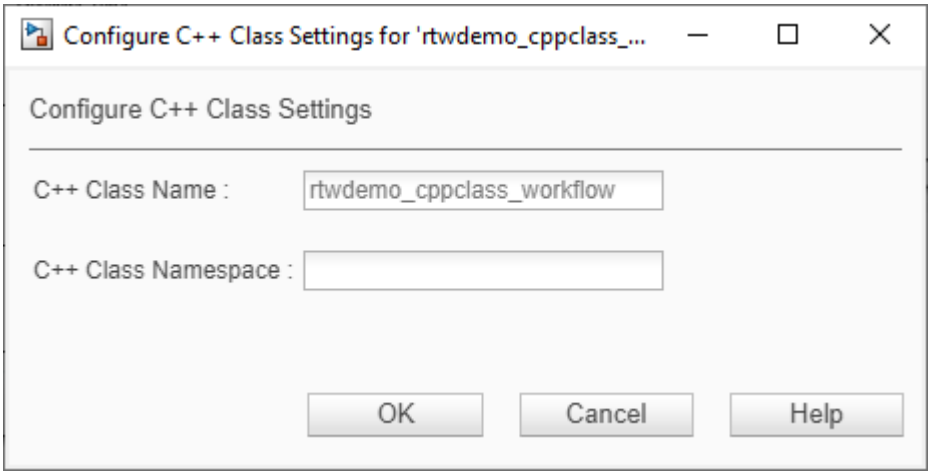
```
// File: CppClassWorkflowKeyIgnition.h

namespace CustomizedNamespace
{
  class customized_ModelClass {
    // public data and function members
```

```
    public:

     // private data and function members
    private:

  };
}
```

Workflow

To configure the class name and namespace for a customized C++ class interface:

1. On the **C++ Code** tab, click **Code Interface** and select **Class Name & Namespace**.



2. To configure the class name, edit the **C++ Class Name** field.
3. To configure the class namespace, edit the **C++ Class Namespace** field.

   You can configure a nested namespace for the model by delimiting the namespaces by using the scope resolution operator `::` in the form `ns1::ns2::ns3`.
4. Click **OK**. Validation is performed interactively with field warnings that alert you if you enter an invalid name or namespace.

Configure Model Data Elements as Class Members

Configure the visibility and access of class members. When you generate C++ code from a model, Simulink data elements appear as class members in the generated code. To adjust the encapsulation of the class data to meet code standards, security, or performance requirements, you can customize the visibility and access of the generated class members. Simulink data elements can be grouped into these categories of modeling elements:

Model Element Category	Description
Inports	Root-level data input ports of a model, such as Inport and In Bus Element blocks. For more information, see Inport.
Outports	Root-level data output ports of a model, such as Outport and Out Bus Element blocks. For more information, see Outport.

Model Element Category	Description
Model parameter arguments	Workspace variables that appear as instance (nonstatic) class data members.
Model parameters	Workspace variables that are shared across instances of the model class that are generated as static class data members.
Signals, states, and internal data	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

For each model element category, you can configure the data visibility to control the access modifier (access specifier) of the generated class members. Each option and its impact on the generated C++ class interface, declared in the model header file, is outlined in this table.

Data Visibility Options	Description

public	If you configure data elements as `public`, they appear as public members of the generated class:
	
	`File: CppClassWorkflowKeyIgnition.h`
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	
	`  // example inport`
	`  ExtU rtU;`
	
	`  // example outport`
	`  ExtY rtY;`
	
	`  // example block signals and states`
	`  DW rtDW;`
	
	`  // Other data, methods, and types (not shown)`
	`  private:`
	`  …`
	`  };`
	`}`

Data Visibility Options	Description

private	If you configure elements as `private`, they appear as private members of the generated class:
	`File: CppClassWorkflowKeyIgnition.h`
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	`  private:`
	`  // example inport`
	`  ExtU rtU;`
	`  // example outport`
	`  ExtY rtY;`
	`  // example block signals and states`
	`  DW rtDW;`
	`  // Other data, methods, and types (not shown)`
	`  };`
	`}`

Individual arguments (Model parameter arguments only)	If you configure data elements as `Individual arguments`, the elements do not appear as members of the class, but are instead passed as arguments to class methods:
	`File: CppClassWorkflowKeyIgnition.h`
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	`  // model step function`
	`  void step1(real_T rty_engineState,`
	`  const real_T modelParameterArgument);`
	`  // Other data, methods, and types (not shown)`
	`  private:`
	`  …`
	`  };`
	`}`

After you set the data visibility of a model element category, you can configure the model access method to determine how the get and set methods are generated for the data elements. This configuration controls how application code can view and modify the class member data. Each option and its impact on the generated C++ class interface, declared in the model header file, is outlined in this table.

Member Access Method Options	Description

Method	If you configure inports as `Method`, a set method appears for each inport. If you configure outports as `Method`, a get method appears for each outport in the model. If you configure the other model element categories as `Method`, an aggregate structure-based get and set method appears for the data element category.
	`File: CppClassWorkflowKeyIgnition.h`
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`   public:`
	`   // example inport`
	`   void setInport(real_T localArgInput);`
	
	`   // example outport`
	`   real_T getOutport() const;`
	
	`   // example of model parameter arguments configured with`
	`   // ‘DataAccess’ ‘Direct'`
	`   real_T const &getInstP() const;`
	
	`   // example of model parameter arguments configured with`
	`   // ‘DataAccess’ ‘Pointer’`
	`   // real_T const *getInstP() const;`
	
	`   // example Block parameters get and set methods`
	`   const P &getBlockParameters() const;`
	
	`   void setBlockParameters(const P *pP) const;`
	
	`   // example Block states get and set methods`
	`   const DW &getDWork() const;`
	
	`   void setDWork(const DW *pDW);`
	
	`   private:`
	`    …`
	`  };`
	`}`

Member Access Method Options	Description

Inlined method	If you configure inports as `Inlined method`, a set method defined in its declaration appears for each inport. If you configure outports as `Inlined method`, a get method defined in its declaration appears for each outport in the model. If you configure the other model element categories as `Inlined method`, an aggregate structure-based get and set method defined in its declaration appears for the data element category.
	
	File: CppClassWorkflowKeyIgnition.h
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`   public:`
	`   // example inport set method`
	`   void setkeyState(real_T localArgInput) {`
	`     rtU.keyState = localArgInput;`
	`   }`
	
	`   // example outport get method`
	`   const real_T *getengineState() const {`
	`     return rtY.engineState;`
	`   }`
	
	`   // example Block parameters get and set method`
	`   const P &getBlockParameters() const {`
	`     return rtP;`
	`   }`
	
	`   void setBlockParameters(const P *pP) const {`
	`     rtP = *pP;`
	`   }`
	
	`  // example Block states get and set methods`
	`   const DW &getDWork() const {`
	`     return rtDW;`
	`   }`
	
	`   void setDWork(const DW *pDW) {`
	`     rtDW = *pDW;`
	`   }`
	
	`   private:`
	`    …`
	`  };`
	`}`

Member Access Method Options	Description

Structure-based method	If you configure the inports as `Structure-based method`, the inports appear as an aggregate structure-based set method for the model element category. If you configure the outports as `Structure-based method`, the outports appear as an aggregate structure-based get method for the model element category. This configuration option does not apply to the other model element categories.
	
	File: CppClassWorkflowKeyIgnition.h
	
	namespace CustomizedNamespace
	{
	  class customized_ModelClass {
	
	  // public data and function members
	  public:
	
	  // External inputs (root inport signals with default storage)
	  struct ExtU {
	  real_T keyState; // '<Root>/keyState'
	  };
	
	  // External outputs (root outports fed by signals with default storage)
	  struct ExtY {
	  real_T engineState[3]; // '<Root>/engineState'
	  real_T cycleTime; // '<Root>/cycleTime'
	  };
	  …
	
	  // example inport
	  void setExternalInputs(const ExtU *pExtU);
	
	  // example outport
	  const ExtY &getExternalOutputs() const;
	
	  // Other model element categories may not be configured as ‘Structure-based metho
	
	  private:
	  …
	  };
	}

Member Access Method Options	Description

Inlined structure-based method	If you configure the inports as `Inlined structure-based method`, the inports appear as an aggregate structure-based set method defined in its declaration for the model element category. If you configure the outports to `Inlined structure-based method`, the outports appear as an aggregate structure-based get method defined in their declaration for the model element category. This configuration option does not apply to the other model element categories.
	
	File: CppClassWorkflowKeyIgnition.h
	
	namespace CustomizedNamespace
	{
	class customized_ModelClass {
	public:
	
	// External inputs (root inport signals with default storage)
	struct ExtU {
	real_T keyState; // '<Root>/keyState'
	};
	
	// External outputs (root outports fed by signals with default storage)
	struct ExtY {
	real_T engineState[3]; // '<Root>/engineState'
	real_T cycleTime; // '<Root>/cycleTime'
	};
	…
	
	// example inport
	void setExternalInputs(const ExtU *pExtU);
	
	// example outport
	const ExtY &getExternalOutputs() const;
	
	// example inport
	void setExternalInputs(const ExtU *pExtU)
	{
	rtU = *pExtU;
	}
	
	// example outport
	const ExtY &getExternalOutputs() const
	{
	return rtY;
	}
	// Other model element categories may not be configured as ‘Inlined structure-base
	
	private:
	…
	};
	}

Member Access Method Options	Description

None	If you configure the access of a model element category as `None`, get and set methods do not appear in the generated class and data can be accessed directly by application code.
	
	File: CppClassWorkflowKeyIgnition.h
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	`  // External inputs (root inport signals with default storage)`
	`  struct ExtU {`
	`  real_T keyState;  // '<Root>/keyState'`
	`  };`
	
	`  // External outputs (root outports fed by signals with default storage)`
	`  struct ExtY {`
	`  real_T engineState[3];  // '<Root>/engineState'`
	`  real_T cycleTime;  // '<Root>/cycleTime'`
	`  };`
	`  …`
	
	`  // External inputs`
	`  ExtU rtU;`
	
	`  // External outputs`
	`  ExtY rtY;`
	
	`  private:`
	`  …`
	`  };`
	`}`

For each model element category, valid data visibility and member access method combinations are outlined in this table.

Model Element Category	Data Visibility	Member Access Method
Inports	private	Method Inlined method Structure-based method Inlined structure-based method
	public	None Method Inlined method Structure-based method Inlined structure-based method

Model Element Category	Data Visibility	Member Access Method
Outports	private	Method Inlined method Structure-based method Inlined structure-based method
	public	None Method Inlined method Structure-based method Inlined structure-based method
Model parameter arguments	Individual arguments	None
	private	Method Inlined method
Model parameters	private	None Method Inlined method
	public	None Method Inlined method
Signals, states, and internal data	private	None Method Inlined method
	public	None Method Inlined method

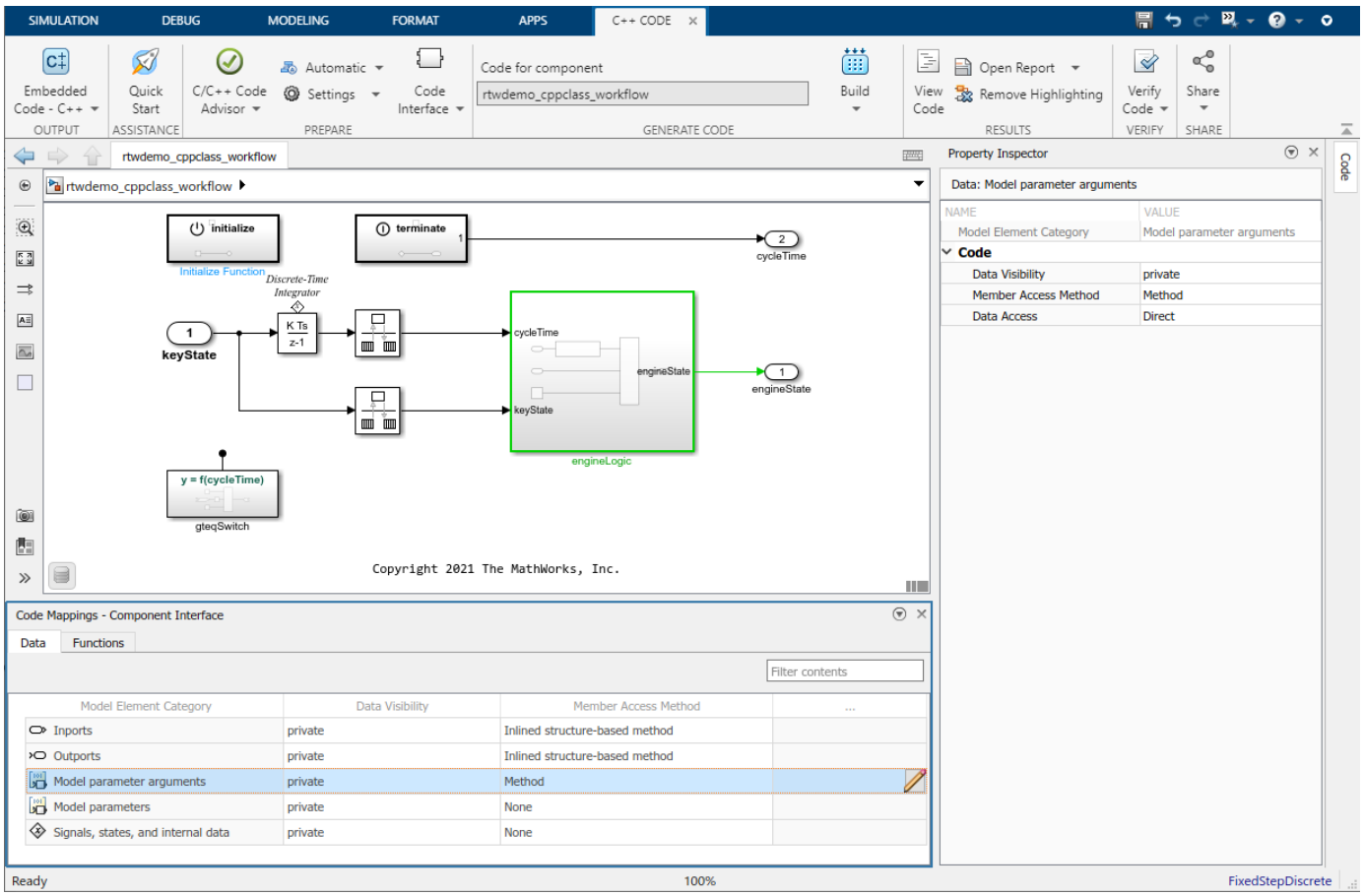
For inports, outports, and model parameter arguments, you can additionally specify whether the data for the model element category is stored by value (`'Direct'`) or pointer (`'Pointer'`) in the generated code.

When configuring data access for **Inports** or **Outports** as pointer members, the model must have Model Configuration Parameters set to either generate an example ERT main program (`ert_main.cpp`) or generate code only. Additionally, the data access method for the inports or outports must be structure-based. To configure the data access method for **Model Parameter Arguments**, data visibility cannot be set to individual arguments.

Workflow

To configure the class members of a customized C++ class interface:

1. Open the Code Mappings editor. On the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.
2. Open the **Data** pane. In the Code Mappings editor, click the **Data** tab.



3. Configure the visibility. To configure the visibility of a category of Simulink data elements in the generated code, from the **Data Visibility** column, select an access specifier from the given options. Options vary depending on the **Model Element Category**.
4. Configure the member access. To configure the access of a category of Simulink data elements in the generated code, from the **Member Access Method** column, select how to generate get and set methods from the given options. Options vary depending on the **Model Element Category** and the selected **Data Visibility**.
5. Configure the data access. To configure data access for inports, outports, or model parameter arguments:
   a. Select the model element that you want to configure (**Inports**, **Outports**, or **Model parameter arguments**).
   b. Verify that **Data Visibility** and **Member Access Method** are set as needed to configure data access.

Data access is configurable for inports, outports, and model parameter arguments with these specifications:

- `'Inports'` whose member access method is set to `'Structure-based method'` or `'Inlined structure-based method'`.
- `'Outports'` whose member access method is set to `'Structure-based method'` or `'Inlined structure-based method'`.
- `'ModelParameterArguments'` whose data visibility is set to `'private'` (not `'Individual Arguments'`).

c Click to open the **Data Access** dialogue box, and select `Pointer` or `Direct`.

Configure Model Functions as Class Methods

Configure the class method names and arguments. When you generate C++ code from a model, model functions appear as class methods in the generated code. To integrate with external code or interface requirements, you can customize the name of the generated class methods. Additionally, for base-rate periodic functions and Simulink Functions, you can configure the name, order, and identifier of the generated arguments.

The generated class methods are referred to as entry-point methods and are locations in code where a transfer of program control (execution) occurs. Entry-point methods vary depending on the type of Simulink model and can be grouped into the following types:

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
Export-Function Models	Exported function	The exported function for a subsystem.	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model	*`function-call-inport-block-name`* or *`signal-label`* (if specified)
	Simulink Function	The exported function for a Simulink Function block.	`Simulink Function:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink Function block in the model	*`function-name`* for a global Simulink Function block or *`model_function-name`* for a scoped Simulink Function block

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
Export-Function or Rate-based Models	Initialize function	Initialization code for a model. At the start of the application code, call the function *once*. Do not use this function to reset the real-time model data structure (`rtM`).	`Initialize`	*`model`*`_initialize`
	Partition function	For a model partition, output and update code. Model configuration parameter **Single output/update function** is selected (the default).	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink® Schedule Editor (for example, P1)	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods
	Periodic multitasking function	For blocks in a rate-based model configured for multitasking, output and update code. The code generator produces a function for each sample period. Model configuration parameter **Single output/update function** is selected (the default).	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multitasking model (for example, D1)	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
	Periodic single-tasking function	For blocks in a rate-based model configured for single-tasking, output and update code. Model configuration parameter **Single output/update function** is selected (the default).	`Periodic`	*`model`*`_step`
	Reset function	If the model includes a Reset Function block, reset code generated. To reset conditions or state, call the function from the application code.	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model	*`model_reset-function-name`*
	Terminate function	Code for turning off a system. For ERT-based models, you can suppress generation of this function by clearing the model configuration parameter **Terminate function required** (set by default).	`Terminate`	*`model`*`_terminate`

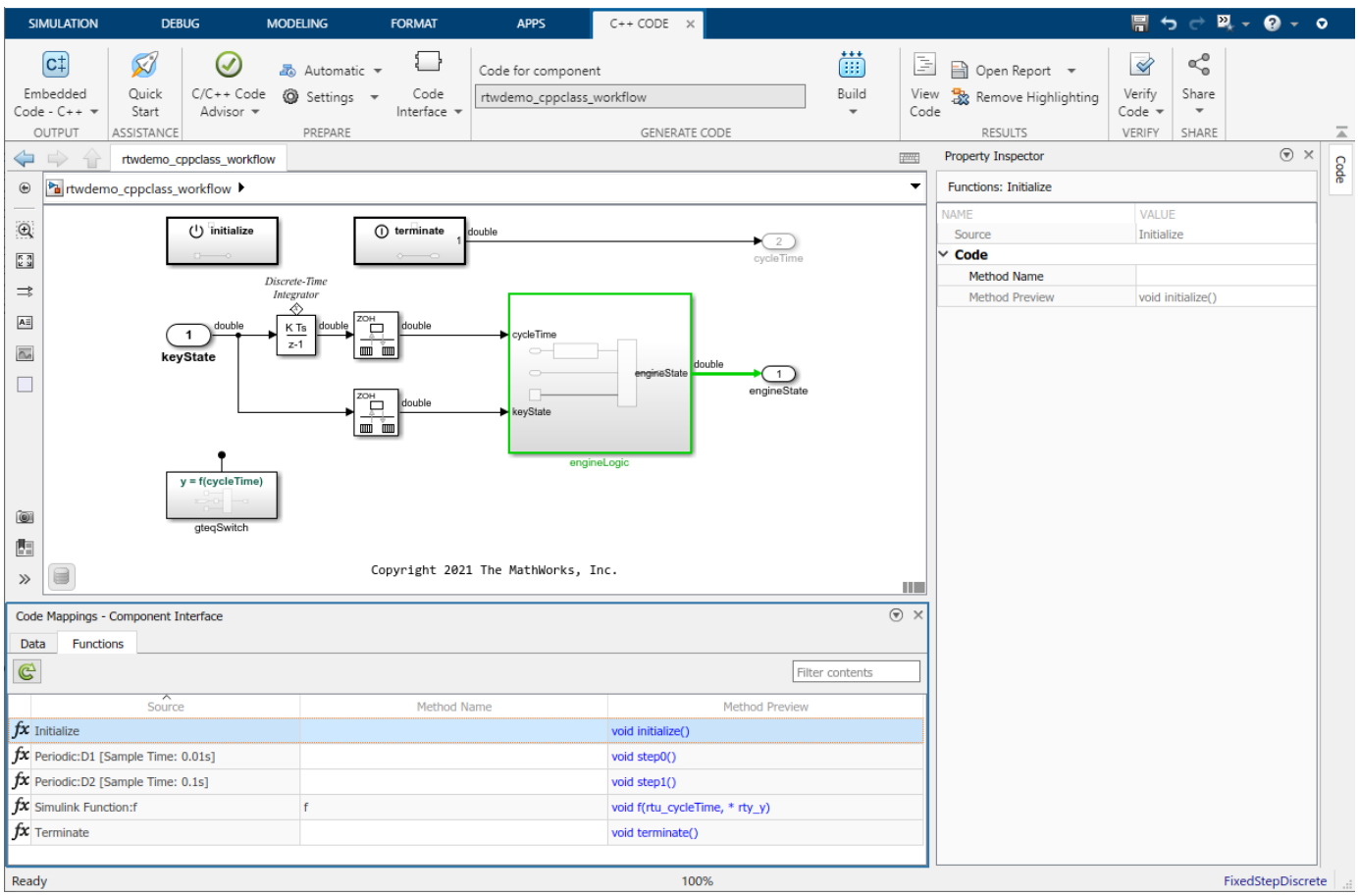
The generated C++ class interface, declared in the model header file, incorporates the function name and argument customizations in the generated entry-point methods:

```
File: CppClassWorkflowKeyIgnition.h

namespace CustomizedNamespace
{
  class customized_ModelClass {
   public:
    // model initialize function- customized name
   void customized_initialize();

    // model step function- customized name & arguments
   void customized_step(customArg1, const* customArg2);
```

```
    // model terminate function- customized name
   void customized_terminate();

    // Constructor
   customized_ModelClass();

    // Destructor
   ~customized_ModelClass();

   private:
  …
  };
}
```

Workflow

1. Open the Code Mappings editor. On the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.
2. Open the **Functions** pane. In the Code Mappings editor, click the **Functions** tab. To view a complete list of entry-point functions for your model, click the **Update Diagram** button.



3. Configure method names.

   a. To configure the name of an entry-point method in the generated code, from the **Method Name** column, click and directly edit the spreadsheet. You can enter a custom name or you

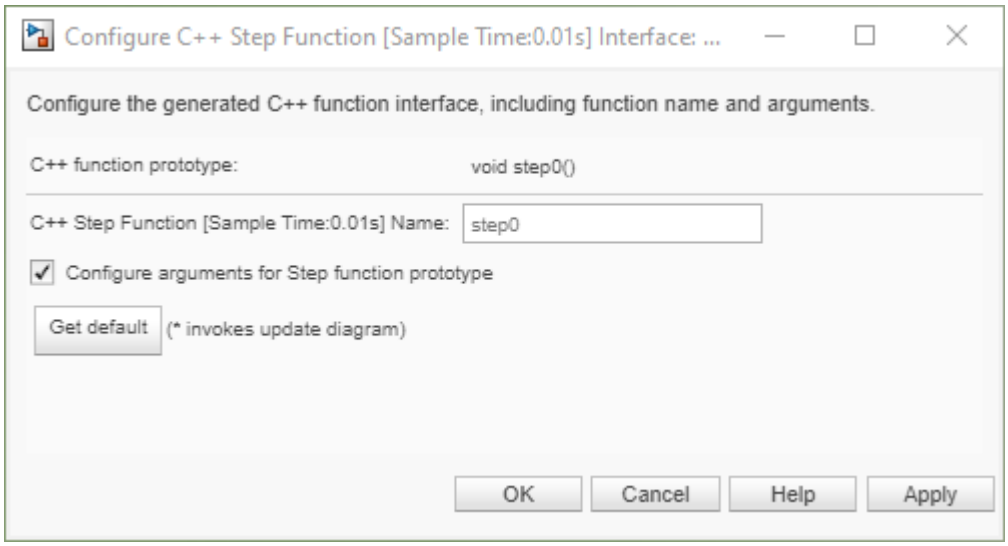
can use identifier format parameters to control a dynamically generated name. For more information about identifier format parameters, see “Identifier Format Control” on page 55-17.

b Verify the name in the **Method Preview** column.

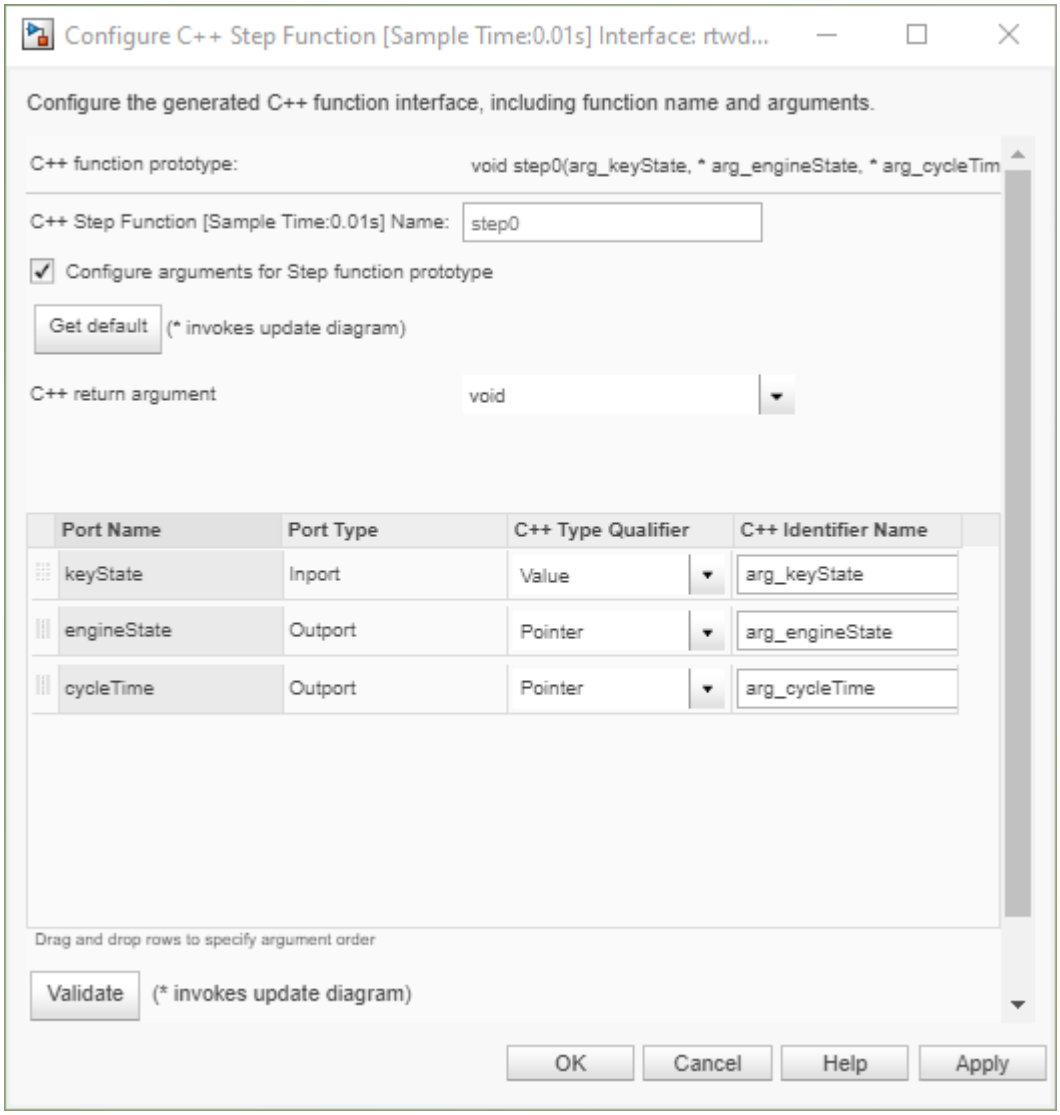
4 Configure method arguments. For base-rate periodic functions and Simulink Functions, you can configure the name, order, and identifiers of the method arguments.

To configure the arguments of a base-rate periodic function:

a Open the Step Function Interface configuration dialog box. From the Code Mappings editor, on the **Functions** tab, in the **Method Preview** column, click the method preview hyperlink of a base-rate periodic function:



b In the dialog box, select **Configure arguments for Step function prototype** and click **Get default**. This action launches a view of the method arguments.



c To change the order of the arguments, click and drag the argument rows in the viewer.

d To change the identifier, in the **C++ Type Qualifier** column, select the appropriate identifier from the drop-down list:

Identifier Option	Preview
Value (Inports only)	myPeriodicFunction(argInport)
Const Reference (Inports only)	myPeriodicFunction(const & argInport)
Pointer to Const	myPeriodicFunction(const * argInport)
Pointer	myPeriodicFunction(* argInport)
Const Pointer to const	myPeriodicFunction(const * const argInport)

e To change the name of an argument, in the **C++ Identifier Name** column, click and directly edit the name. You can enter a custom name or you can use identifier format parameters to

control a dynamically generated name. For more information about identifier format parameters, see “Identifier Format Control” on page 55-17.

i If you enter the same name for an inport argument and an outport argument the inport and outport appear as one parameter in the generated method. For example, if you specify an inport as `argInport` and an outport argument as `argOutport`, the generated method would be:

```
myStepFunction(argInport, argOutport);
```

If you specify an inport argument as `argIO` and an outport argument as `argIO`, the generated method would be:

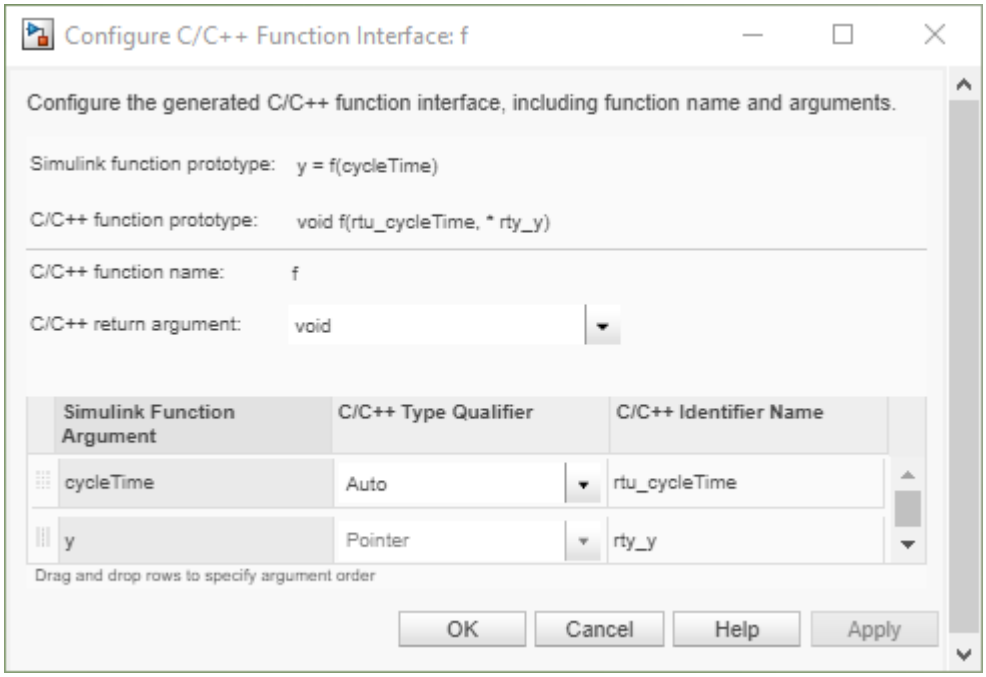
```
myStepFunction(argIO);
```

f To verify the arguments selections, click **Validate**.

g To apply changes and exit the dialog box, click **OK**.

To configure the arguments of a Simulink Function:

a Open the Simulink Function Interface configuration dialog box. From the Code Mappings editor, on the **Functions** tab, in the **Method Preview** column, click the method preview hyperlink of a Simulink Function:



b To change the return argument, select a return argument option from the **C/C++ return argument** drop-down list.

c To change the order of the arguments, click and drag the argument rows in the viewer.

d To change the identifier, in the **C++ Type Qualifier** column, select the identifier from the drop-down list:

Identifier Option	Preview
Auto	mySimulinkFunction(argInport)
Const Reference	mySimulinkFunction(const & argInport)
Pointer to Const	mySimulinkFunction(const * argInport)
Pointer	mySimulinkFunction(* argInport)
Const Pointer to const	mySimulinkFunction(const * const argInport)

e To change the name of the argument, in the **C++ Identifier name** column, click and directly edit the name. You can enter a custom name or you can use identifier format parameters to control a dynamically generated name. For more information about identifier format parameters, see “Identifier Format Control” on page 55-17.

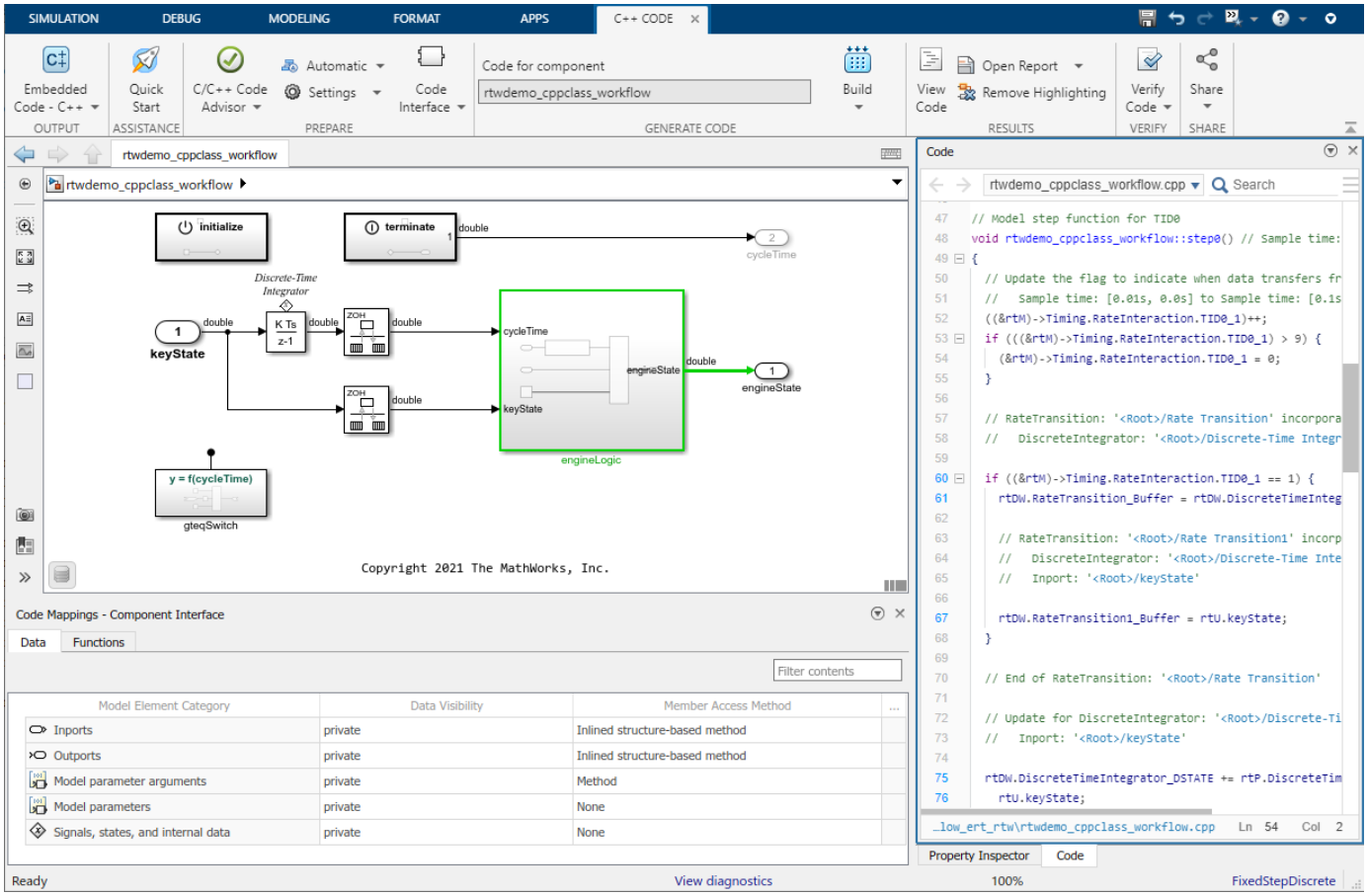
f To verify changes to the Simulink Function arguments, review the **Simulink function prototype** field to view the expected method prototype.

g To apply changes and exit the dialog box, click **OK**.

Generate C++ Class Interface

Generate the customized C++ class interface by generating code from the model. To verify your C++ class interface configurations, build the model and view the generated class, class member, and class method representations.

1. Generate code. To generate a C++ class interface, on the **C++ Code** tab, click **Build**.
2. View code. To view the generated code, on the tab click **View Code**. The generated code appears beside the model in the model workspace.



3 Iterate. If the generated interface does not meet code requirements, make configuration adjustments until requirements are satisfied.

If the generated code representation does not meet your requirements, reconfigure the interface and generate code again until code generations requirements are satisfied. For guidance on understanding the generated code, see “Analyze Generated Data Code Interface” on page 54-13.

Considerations and Limitations

- Instance-Specific Parameter Support — You can use model parameter arguments to configure workspace variables that have been specified as arguments. You can configure these arguments as private members of your class or as individual arguments defined outside the class. To configure within the class, you can set the data visibility to private and configure the generation of the get and set methods. Optionally, you can configure the class member values as defined within the class (in the Property Inspector, set **Data Access** to `Direct`) or passed-by-reference through the class constructor (in the Property Inspector, set **Data Access** to `Pointer`). The data access setting is supported for top model builds. For reference model builds, the code is generated as references in the model class. To configure instance-specific parameters outside the class, set the data visibility to individual arguments.

  Instance-specific parameter limitations include:

  - MATLAB variables marked as model arguments are not able to be configured as private class members.

  - Model parameter arguments are not supported by right-click builds.
- Interface Code Generation Behavior — The I/O arguments style of step method specification supports single-rate models and multirate single-tasking models. Multirate multitasking models are not supported. Additionally, the C++ encapsulation interface is not the default, the value is ignored for the **Pass fixed-size scalar root inputs by value for code generation** parameter.
- Stateflow considerations — If you have a Stateflow license, for a Stateflow chart that resides in a root model configured to use the `I/O arguments step method` function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, do one of the following to generate code:
  - In the Stateflow chart, clear the **Execute (enter) Chart At Initialization** check box.
  - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- Simscape Considerations — If a model root inport value connects to a Simscape block, insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- Referenced Model Considerations — When building a referenced model that is configured to generate a C++ class interface:
  - Do not use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that has a continuous sample time or saves states.
  - Do not use virtual buses as inputs or outputs to the referenced model when the referenced model uses the I/O arguments step method. When bus signals cross referenced model boundaries, either use nonvirtual buses or use the Default step method.

See Also

More About

- “Programmatically Configure C++ Interface” on page 44-25
- “Analyze Generated Data Code Interface” on page 54-13
- “Identifier Format Control” on page 55-17

Programmatically Configure C++ Interface

To streamline the integration of C++ code generated from Simulink applications, components, and subsystems with external C or C++ code, configure a customized C++ class interface. When you generate C++ code from a model, the model appears as a class, the data elements appear as class members, and the model functions appear as class methods in the generated code. Configuring a C++ class interface enables you to customize the following aspects of the generated C++ code:

- Class information — Class name and namespace
- Class member information — Class member access and visibility
- Class method information — Class method names and arguments

A customized C++ class interface enables the generated classes to meet specific code standards or interface requirements so that the generated code can compile and integrate into larger architectures with minimal post-generation customization.

Programmatic Workflow Overview

The programmatic workflow uses a programmatic interface that you can call into by using the MATLAB command line or a MATLAB script to configure a customized C++ class interface. Additionally, you can use the functions to facilitate scripting, automated testing, and developing or utilizing downstream features. The programmatic interface is composed of the following functions:

- To create or retrieve the Code Mappings object to configure a customized C++ Class interface, you can use the functions `coder.mapping.utils.create` and `coder.mapping.api.get`.
- To configure the class name and namespace, you can use the functions `getClassName`, `setClassName`, `getClassNamespace`, and `setClassNamespace`.
- To configure how data elements appear as class members, you can use the functions `getData` and `setData`.
- To configure how model functions appear as class methods, you can use the functions `find`, `getFunction`, and `setFunction`.

To programmatically configure a customized C++ interface, follow the outlined workflow:

1. “Open Environment” on page 44-25
2. “Configure Model as Class” on page 44-27
3. “Configure Model Data Elements as Class Members” on page 44-28
4. “Configure Model Functions as Class Methods” on page 44-38
5. “Generate C++ Class Interface” on page 44-43

Open Environment

Open the environment for configuring a customized C++ class interface:

1. Open the model and the MATLAB command prompt.

   This workflow uses the model `CppClassWorkflowKeyIgnition`, which is a supporting file in an existing example. Alternatively, you can create a MATLAB configuration script.

   ```
   openExample('ecoder/CppCodeGenerationWorkflowForKeyIgnitionSystemExample', ...
               supportingFile='CppClassWorkflowKeyIgnition.slx')
   ```

```
model='CppClassWorkflowKeyIgnition'
open_system(model)
```

2 Set model configuration parameters. These parameters configure model-wide code generation behavior. To configure parameters specific to generating a customized C++ class interface, you can use the `set_param` function to set these parameters:

Configuration Parameter	Description
Code interface packaging	Selects output language for generated code. For more information, see Code interface packaging.
Multi-instance code error diagnostic	Specifies the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code. For more information, see Multi-instance code error diagnostic.
Remove error status field in real-time model data structure	Specifies whether to omit the error status field from the generated real-time model data structure `rtModel`. For more information, see Remove error status field in real-time model data structure.
Include model types in model class	Specifies to include model type definitions within the class namespace of the model. For more information, see Include model types in model class.

Interface parameters that are related, but are less commonly used:

Configuration Parameter	Description
Terminate function required	Specifies whether to generate the *`model`*`_terminate` method. For more information, see Terminate function required.
Combine signal/state structures	Specifies whether to combine global block signals and global state data into one data structure in the generated code. For more information, see Combine signal/state structures.
Generate destructor	Specifies whether to generate a destructor for the C++ model class. For more information, see Generate destructor.
Use dynamic memory allocation for model block instantiation	Specifies memory allocation for model hierarchies. For more information, seeUse dynamic memory allocation for model block instantiation.

3 To configure a customized C++ class interface for a model, create a Code Mappings object. To create a new Code Mappings object for a model, use the function `coder.mapping.utils.create`. If a Code Mappings object already exists for the model, the function returns the existing object.

```
% Create a new Code Mappings object for a model
cm = coder.mapping.utils.create(model);
```

Configure Model as Class

Configure the class name and namespace. When you generate C++ code from a model, that model appears as a class in the generated code. To ease integration and comply with code and interface requirements, you can customize the generated class name. Optionally, you can scope the generated code and prevent symbol clashes within a project by specifying a namespace for the generated class. In modeled systems that use a model hierarchy, you can specify a different namespace for each model in the hierarchy.

The generated C++ class interface, declared in the model header file, incorporates the customized name and namespace:

```
// File: CppClassWorkflowKeyIgnition.h

namespace CustomizedNamespace
{
  class customized_ModelClass {
    // public data and function members
   public:

    // private data and function members
   private:

  };
}
```

Workflow

To programmatically configure the class name and namespace, get the Code Mappings object for your model and set the name and namespace:

1. Get the Code Mappings object for your model. To get the object, use the function `coder.mapping.api.get`.

   ```
   % Get the Code Mappings object for a model
   cm = coder.mapping.api.get(model);
   ```

2. Configure the class name. To set the class name, use the function `setClassName`.

   ```
   % Set the class name
   setClassName(cm, 'customized_ModelClass');
   ```

   To verify the new class name before generating code, use the function `getClassName`.

   ```
   % Get the class name
   getClassName(cm);
   ```

3. Configure the class namespace. To set the namespace, use the function `setClassNamespace`.

   ```
   % Set the class namespace
   setClassNamespace(cm, 'customized_namespace');
   ```

   To configure a nested namespace, use the scope resolution operator `::` to specify scope.

   ```
   % Set a nested namespace
   setClassNamespace(cm, 'customized_namespace1::customized_namespace2');
   ```

   To verify the new namespace before generating code, use the function `getClassNamespace`.

```
% Get the class namespace
getClassNamespace(cm);
```

Configure Model Data Elements as Class Members

Configure the visibility and access of class members. When you generate C++ code from a model, Simulink data elements appear as class members in the generated code. To adjust the encapsulation of the class data to meet code standards, security, or performance requirements, you can customize the visibility and access of the generated class members. Simulink data elements can be grouped into the following categories of modeling elements:

Model Element Category	Description
Inports	Root-level data input ports of a model, such as Inport and In Bus Element blocks. For more information, see Inport.
Outports	Root-level data output ports of a model, such as Outport and Out Bus Element blocks. For more information, see Outport.
Model parameter arguments	Workspace variables that appear as instance (nonstatic) class data members.
Model parameters	Workspace variables that are shared across instances of the model class that are generated as static class data members.
Signals, states, and internal data	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

For each model element category, you can configure the data visibility to control the access modifier (access specifier) of the generated class members. Each option and its impact on the generated C++ class interface, declared in the model header file, is outlined in this table.

Data Visibility Options	Description

public	If you configure data elements as `public`, they appear as public members of the generated class:
	
	File: CppClassWorkflowKeyIgnition.h
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	
	`  // example inport`
	`  ExtU rtU;`
	
	`  // example outport`
	`  ExtY rtY;`
	
	`  // example block signals and states`
	`  DW rtDW;`
	
	`  // Other data, methods, and types (not shown)`
	`  private:`
	`  …`
	`  };`
	`}`

protected	If you configure data elements as `protected`, they appear as protected members of the generated class:
	
	File: CppClassWorkflowKeyIgnition.h
	
	`// Class declaration for model example_model`
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	
	`  protected:`
	
	`  // example inport`
	`  ExtU rtU;`
	
	`  // example outport`
	`  ExtY rtY;`
	
	`  // example block signals and states`
	`  DW rtDW;`
	
	`  // Other data, methods, and types (not shown)`
	`  private:`
	`  …`
	`  };`
	`}`

Data Visibility Options	Description

private	If you configure elements as `private`, they appear as private members of the generated class:
	`File: CppClassWorkflowKeyIgnition.h`
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`   public:`
	`   private:`
	`    // example inport`
	`    ExtU rtU;`
	`    // example outport`
	`    ExtY rtY;`
	`    // example block signals and states`
	`    DW rtDW;`
	`   // Other data, methods, and types (not shown)`
	`  };`
	`}`

Individual arguments (Model parameter arguments only)	If you configure data elements as `Individual arguments`, the elements do not appear as members of the class, but are instead passed as arguments to class methods:
	`File: CppClassWorkflowKeyIgnition.h`
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`   public:`
	`   // model step function`
	`   void step1(real_T rty_engineState,`
	`     const real_T modelParameterArgument);`
	`   // Other data, methods, and types (not shown)`
	`   private:`
	`    …`
	`  };`
	`}`

After you set the data visibility of a model element category, you can configure the member access method to determine how the get and set methods are generated for the data elements. This configuration controls how application code can view and modify the class member data. Each option and its impact on the generated C++ class interface, declared in the model header file, is outlined in this table.

Member Access Method Options	Description

Method	If you configure inports as `Method`, a set method appears for each inport. If you configure outports as `Method`, a get method appears for each outport in the model. If you configure the other model element categories as `Method`, an aggregate structure-based get and set method appears for the data element category.
	
	File: CppClassWorkflowKeyIgnition.h
	
	namespace CustomizedNamespace
	{
	  class customized_ModelClass {
	  public:
	  // example inport
	  void setInport(real_T localArgInput);
	
	  // example outport
	  real_T getOutport() const;
	
	  // example of model parameter arguments configured with
	  // ‘DataAccess’ ‘Direct'
	  real_T const &getInstP() const;
	
	  // example of model parameter arguments configured with
	  // ‘DataAccess’ ‘Pointer’
	  // real_T const *getInstP() const;
	
	  // example Block parameters get and set methods
	  const P &getBlockParameters() const;
	
	  void setBlockParameters(const P *pP) const;
	
	  // example Block states get and set methods
	  const DW &getDWork() const;
	
	  void setDWork(const DW *pDW);
	
	  private:
	  …
	  };
	}

Member Access Method Options	Description

Inlined method	If you configure inports as `Inlined method`, a set method defined in its declaration appears for each inport. If you configure outports as `Inlined method`, a get method defined in its declaration appears for each outport in the model. If you configure the other model element categories as `Inlined method`, an aggregate structure-based get and set method defined in its declaration appears for the data element category.
	
	File: CppClassWorkflowKeyIgnition.h
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`   public:`
	`   // example inport set method`
	`   void setkeyState(real_T localArgInput) {`
	`     rtU.keyState = localArgInput;`
	`   }`
	
	`   // example outport get method`
	`   const real_T *getengineState() const {`
	`     return rtY.engineState;`
	`   }`
	
	`   // example Block parameters get and set method`
	`   const P &getBlockParameters() const {`
	`     return rtP;`
	`   }`
	
	`   void setBlockParameters(const P *pP) const {`
	`     rtP = *pP;`
	`   }`
	
	`  // example Block states get and set methods`
	`   const DW &getDWork() const {`
	`     return rtDW;`
	`   }`
	
	`   void setDWork(const DW *pDW) {`
	`     rtDW = *pDW;`
	`   }`
	
	`   private:`
	`    …`
	`  };`
	`}`

Member Access Method Options	Description

Structure-based method	If you configure the inports as `Structure-based method`, the inports appear as an aggregate structure-based set method for the model element category. If you configure the outports as `Structure-based method`, the outports appear as an aggregate structure-based get method for the model element category. This configuration option does not apply to the other model element categories.
	
	File: CppClassWorkflowKeyIgnition.h
	
	namespace CustomizedNamespace
	{
	  class customized_ModelClass {
	
	  // public data and function members
	  public:
	
	  // External inputs (root inport signals with default storage)
	  struct ExtU {
	  real_T keyState; // '<Root>/keyState'
	  };
	
	  // External outputs (root outports fed by signals with default storage)
	  struct ExtY {
	  real_T engineState[3]; // '<Root>/engineState'
	  real_T cycleTime; // '<Root>/cycleTime'
	  };
	  …
	
	  // example inport
	  void setExternalInputs(const ExtU *pExtU);
	
	  // example outport
	  const ExtY &getExternalOutputs() const;
	
	  // Other model element categories may not be configured as ‘Structure-based metho
	
	  private:
	  …
	  };
	}

Member Access Method Options	Description

Inlined structure-based method	If you configure the inports as `Inlined structure-based method`, the inports appear as an aggregate structure-based set method defined in its declaration for the model element category. If you configure the outports to `Inlined structure-based method`, the outports appear as an aggregate structure-based get method defined in their declaration for the model element category. This configuration option does not apply to the other model element categories.
	
	File: CppClassWorkflowKeyIgnition.h
	
	namespace CustomizedNamespace
	{
	class customized_ModelClass {
	public:
	
	// External inputs (root inport signals with default storage)
	struct ExtU {
	real_T keyState; // '<Root>/keyState'
	};
	
	// External outputs (root outports fed by signals with default storage)
	struct ExtY {
	real_T engineState[3]; // '<Root>/engineState'
	real_T cycleTime; // '<Root>/cycleTime'
	};
	…
	
	// example inport
	void setExternalInputs(const ExtU *pExtU);
	
	// example outport
	const ExtY &getExternalOutputs() const;
	
	// example inport
	void setExternalInputs(const ExtU *pExtU)
	{
	rtU = *pExtU;
	}
	
	// example outport
	const ExtY &getExternalOutputs() const
	{
	return rtY;
	}
	// Other model element categories may not be configured as ‘Inlined structure-base
	
	private:
	…
	};
	}

Member Access Method Options	Description

None	If you configure the access of a model element category as `None`, get and set methods do not appear in the generated class and data can be accessed directly by application code.
	
	File: CppClassWorkflowKeyIgnition.h
	
	`namespace CustomizedNamespace`
	`{`
	`  class customized_ModelClass {`
	`  public:`
	`  // External inputs (root inport signals with default storage)`
	`  struct ExtU {`
	`  real_T keyState;  // '<Root>/keyState'`
	`  };`
	
	`  // External outputs (root outports fed by signals with default storage)`
	`  struct ExtY {`
	`  real_T engineState[3];  // '<Root>/engineState'`
	`  real_T cycleTime;  // '<Root>/cycleTime'`
	`  };`
	`  …`
	
	`  // External inputs`
	`  ExtU rtU;`
	
	`  // External outputs`
	`  ExtY rtY;`
	
	`  private:`
	`  …`
	`  };`
	`}`

For each model element category, valid data visibility and member access method combinations are outlined in this table.

Model Element Category	Data Visibility	Member Access Method
Inports	private	Method Inlined method Structure-based method Inlined structure-based method
	public	None Method Inlined method Structure-based method Inlined structure-based method

Model Element Category	Data Visibility	Member Access Method
Outports	private	Method Inlined method Structure-based method Inlined structure-based method
	public	None Method Inlined method Structure-based method Inlined structure-based method
Model parameter arguments	Individual arguments	None
	private	Method Inlined method
Model parameters	private	None Method Inlined method
	public	None Method Inlined method
Signals, states, and internal data	private	None Method Inlined method
	public	None Method Inlined method

For inports, outports, and model parameter arguments, you can additionally specify whether the data for the model element category is stored by value (`'Direct'`) or pointer (`'Pointer'`) in the generated code.

When configuring data access for **Inports** or **Outports** as pointer members, the model must have Model Configuration Parameters set to either generate an example ERT main program (`ert_main.cpp`) or generate code only. Additionally, the data access method for the inports or outports must be structure-based. To configure the data access method for Model Parameter Arguments, data visibility cannot be set to individual arguments.

Workflow

To programmatically configure class member visibility and access, get the Code Mappings object for the model and set the access and visibility properties:

1. Get the Code Mappings object for the model. To get the object, use the function `coder.mapping.api.get`.

   ```
   % Get the Code Mappings object
   cm = coder.mapping.api.get(model);
   ```

2. Configure the visibility. To set the data visibility of a category of Simulink data elements in the generated code, use the function `setData`. When you use the `setData` function, specify the Code Mappings object, model element category, data visibility property, and data visibility option. Valid data visibility options vary depending on the model element category.

   ```
   % Set the data visibility of Simulink model data elements by category
   % setData(Code Mappings object, model element category, 'DataVisibility', data visibility opt
   setData(cm, 'Outports', 'DataVisibility', 'private');
   ```

   To verify the data visibility setting before generating code, use the function `getData`:

   ```
   % Get the data visibility of Simulink model data elements by category
   % getData(Code mappings object, model element category, 'DataVisibility')
   getData(cm, 'Outports', 'DataVisibility');
   ```

3. Configure the member access. To set the member access method of a category of Simulink data elements in the generated code, use the function `setData`. When you use the `setData` function, specify the Code Mappings object, model element category, member access method property, and member access method option. Valid member access method options vary depending on the model element category and selected data visibility option.

   ```
   % Set the access of Simulink model data elements by category
   % setData(Code Mappings object, model element category,...
   % 'MemberAccessMethod', member access method option)
   setData(cm, 'Inports', 'MemberAccessMethod', 'Structure-based method');
   ```

   To verify the member access method setting before generating code, use the function `getData`:

   ```
   % Get the member access method of Simulink model data elements by category
   % getData(Code Mappings object, model element category, 'MemberAccessMethod')
   getData(cm, 'Inports', 'MemberAccessMethod');
   ```

4. Configure the data access for model element inports, outports, or model parameter arguments.

   - `'Inports'` or `'Outports'` – Can be configured as pointer or direct access for inports or outports whose member access method is set to `'Structure-based method'` or `'Inlined structure-based method'`. The default configuration is direct access.
   - `'ModelParameterArguments'` – Can be configured as pointer or direct access for model parameter arguments whose data visibility is set to `'private'` (not `'Individual Arguments'`).

   To configure data access, use the function `setData`.

   For example:

   ```
   % For inports, verify the member access method is structure-based
   getData(cm, 'Inports', 'MemberAccessMethod');
   ```

```
% Set the data access method for inports
% setData(Code Mappings object, model element category, 'DataAccess', value)
setData(cm, 'Inports', 'DataAccess', 'Pointer');

% For model parameter arguments, verify the visibility is private
getData(cm, 'ModelParameterArguments', 'DataVisibility');
% Set the data access method for model parameter arguments
setData(cm, 'ModelParameterArguments', 'DataAccess', 'Direct');

% Get the data access method of Simulink model data elements by category
% getData(Code Mappings object, model element category, 'DataAccess')
getData(cm, 'Inports', 'DataAccess');
```

Configure Model Functions as Class Methods

Configure the class method names and arguments. When you generate C++ code from a model, model functions appear as class methods in the generated code. To integrate with external code or interface requirements, you can customize the name of the generated class methods. Additionally, for base-rate periodic functions and Simulink Functions, you can configure the name, order, and identifier of the generated arguments.

The generated class methods are referred to as entry-point methods and are locations in code where a transfer of program control (execution) occurs. Entry-point methods vary depending on the type of Simulink model and can be grouped into the following types:

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
Export-Function Models	Exported function	The exported function for a subsystem.	`ExportedFunction:`*`slIdentifier`*, where *`slIdentifier`* is the name of the function-call Inport block in the model	*`function-call-inport-block-name`* or *`signal-label`* (if specified)
	Simulink Function	The exported function for a Simulink Function block.	`Simulink Function:`*`slIdentifier`*, where *`slIdentifier`* is the name of the Simulink Function block in the model	*`function-name`* for a global Simulink Function block or *`model_function-name`* for a scoped Simulink Function block

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
Export-Function or Rate-based Models	Initialize function	Initialization code for a model. At the start of the application code, call the function *once*. Do not use this function to reset the real-time model data structure (`rtM`).	`Initialize`	*`model`*`_initialize`
	Partition function	For a model partition, output and update code. Model configuration parameter **Single output/update function** is selected (the default).	`Partition:`*`slIdentifier`*, where *`slIdentifier`* is a partition that was created explicitly from a block in the model and shown in the Simulink® Schedule Editor (for example, P1)	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods
	Periodic multitasking function	For blocks in a rate-based model configured for multitasking, output and update code. The code generator produces a function for each sample period. Model configuration parameter **Single output/update function** is selected (the default).	`Periodic:`*`slIdentifier`*, where *`slIdentifier`* is an annotation that corresponds to the sample time period for a periodic or continuous rate of a multitasking model (for example, D1)	*`model`*`_step`*`n`*, where *`n`* uniquely identifies the function generated for one of the model sample periods

Model Type	Type of Model Function	Description	Name of Model Function	Expected Method Name
	Periodic single-tasking function	For blocks in a rate-based model configured for single-tasking, output and update code. Model configuration parameter **Single output/update function** is selected (the default).	`Periodic`	*`model`*`_step`
	Reset function	If the model includes a Reset Function block, reset code generated. To reset conditions or state, call the function from the application code.	`Reset:`*`slIdentifier`*, where *`slIdentifier`* is the name of the reset function in the model	*`model_reset-function-name`*
	Terminate function	Code for turning off a system. For ERT-based models, you can suppress generation of this function by clearing the model configuration parameter **Terminate function required** (set by default).	`Terminate`	*`model`*`_terminate`

The generated C++ class interface, declared in the model header file, incorporates the function name and argument customizations in the generated entry-point methods:

```
File: CppClassWorkflowKeyIgnition.h

namespace CustomizedNamespace
{
  class customized_ModelClass {
   public:
    // model initialize function- customized name
   void customized_initialize();

    // model step function- customized name & arguments
   void customized_step(customArg1, const* customArg2);
```

```
    // model terminate function- customized name
   void customized_terminate();

    // Constructor
   customized_ModelClass();

    // Destructor
   ~customized_ModelClass();

   private:
  …
  };
}
```

Workflow

To programmatically configure class method names and arguments, get the Code Mappings object for the model and set the function properties:

1. Get the Code Mappings object for the model. To get the object, use the function `coder.mapping.api.get`.

   ```
   % Get the Code Mappings object
   cm = coder.mapping.api.get(model);
   ```

2. To view the available entry-point methods for the model, use the function `find`. When you use the `find` function, specify the Code Mappings object and the function category you would like to view for the model. Valid function category options are the following:

   - Functions
   - PeriodicFunctions
   - PartitionFunctions
   - ExportedFunctions
   - ResetFunctions
   - SimulinkFunctions

   ```
   % View entry-point methods for a model by category
   % This example shows how to view the entry-point functions for a model
   % find(Code Mappings object, function category')
   find(cm, 'Functions');
   ```

3. Configure method names. To configure the method name of an entry-point method, use the function `setFunction`. When you use the `setFunction` function, specify the Code Mappings object, model function name, the method name property, and the custom name. Valid model function names are specified in the following way:

   - `Initialize` — Initialize function.
   - `Terminate` — Terminate function.
   - `Periodic:slIdentifier` — Periodic function (implicit task), where `slIdentifier` is the annotation (for example, D1) corresponding to the sample time period. `Periodic` is sufficient for a single-tasking periodic function.
   - `Partition:slIdentifier` — Partition function (explicit task), where `slIdentifier` is the partition name in Schedule Editor.
   - `Reset:slIdentifier` — Reset function where `slIdentifier` is name of the reset function in model.

- `ExportedFunction:slIdentifier` — Exported function where `slIdentifier` is the name of function-call inport block.
- `ExportedFunction:slIdentifier` — Simulink function where `slIdentifier` is the name of Simulink function in model.

```
% Set the name of individual entry-point methods
% This example shows how to customize the name of a periodic function
% setFunction(Code Mappings object, model function, 'MethodName', custom name)
setFunction(cm, 'Periodic', 'MethodName', 'custom_Periodic');
```

To verify the method name before generating code, use the function `getFunction`:

```
% Get method name
% getFunction(Code mappings object, model function, 'MethodName')
getData(cm, 'Periodic', 'MethodName');
```

4 Configure method arguments. For base-rate periodic functions and Simulink Functions, you can use the function `setFunction` to configure the generated method arguments. When you use the `setFunction` function, specify the Code Mappings object, model function name, the arguments property, and the method prototype. Valid model function names are specified in the following way:

- `Initialize` — Initialize function.
- `Terminate` — Terminate function.
- `Periodic:slIdentifier` — Periodic function (implicit task), where `slIdentifier` is the annotation (for example, D1) corresponding to the sample time period. `Periodic` is sufficient for a single-tasking periodic function.
- `Partition:slIdentifier` — Partition function (explicit task), where `slIdentifier` is the partition name in Schedule Editor.
- `Reset:slIdentifier` — Reset function where `slIdentifier` is name of the reset function in model.
- `ExportedFunction:slIdentifier` — Exported function where `slIdentifier` is the name of function-call inport block.
- `ExportedFunction:slIdentifier` — Simulink function where `slIdentifier` is the name of Simulink function in model.

Valid identifier options for parameters are the following:

Identifier Option	Preview
Value (Inports only)	myPeriodic(argInport)
Const Reference (Inports only)	myPeriodic(const & argInport)
Pointer to Const	myPeriodic(const * argInport)
Pointer	myPeriodic(* argInport)
Const Pointer to const	myPeriodic(const * const argInport)

In the below example, the periodic function, `foo`, is configured to generate a method where `y` is the return argument and its parameter `u1` is passed as a scalar and `u2` is passed as a constant pointer:

```
% Configure method arguments
% setFunction(Code Mappings object, function, 'Arguments', function prototype)
setFunction(cm, 'Periodic:foo', 'Arguments', 'y=(u1, const *u2)');
```

To verify the method arguments before generating code, use the function `getFunction`:

```
% Get method name
% getFunction(Code Mappings object, model function, 'Arguments')
getData(cm, 'Periodic:foo', 'Arguments');
```

Generate C++ Class Interface

Generate the customized C++ class interface by generating code from the model. To verify your customized C++ class interface configurations, build the model and view the generated class, class member, and class method representations.

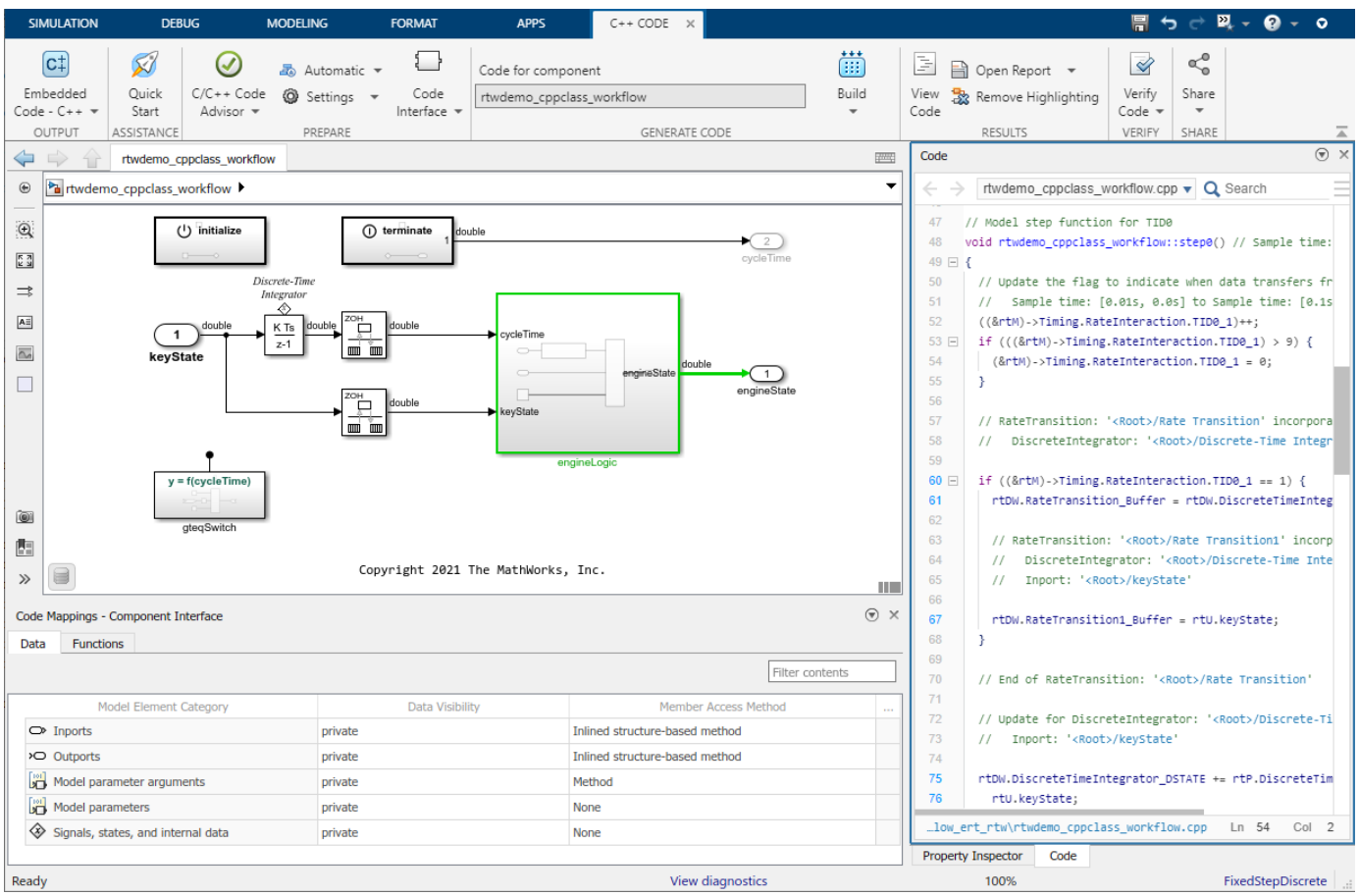
1. Generate code. To generate a C++ class interface, build the model with the command `slbuild`.

   ```
   slbuild example_model
   ```

2. View code. To view the generated code, open the model in the Embedded Coder app.

   ```
   open example_model
   ```

   To open the **Code** view, on the **C++ Code** tab, click **View Code**. The generated code appears beside the model in the model workspace.



3. Iterate. If the generated interface does not meet code requirements, make configuration adjustments until requirements are satisfied.

If the generated code representation does not meet your requirements, reconfigure the interface and generate code again until code generations requirements are satisfied. For guidance on understanding the generated code, see “Analyze Generated Data Code Interface” on page 54-13.

Considerations and Limitations

- Instance-Specific Parameter Support — You can use model parameter arguments to configure workspace variables that have been specified as arguments. You can configure these arguments as private members of your class or as individual arguments defined outside the class. To configure within the class, you can set the data visibility setting to private and configure the generation of the get and set methods. Optionally, you can configure the class member values as defined within the class (`setData(cm,'ModelParameterArguments','DataAccess','Direct')`) or passed-by-reference through the class constructor (`setData(cm,'ModelParameterArguments','DataAccess','Pointer')`). The data access setting is supported for top model builds. For reference model builds, the code is generated as references in the model class. To configure instance-specific parameters outside the class, set the data visibility to individual arguments.

  Instance-specific parameter limitations include:

  - MATLAB variables marked as model arguments are not able to be configured as private class members.
  - Model parameter arguments are not supported by right-click builds.
- Interface Code Generation Behavior — The I/O arguments style of step method specification supports single-rate models and multirate single-tasking models. Multirate multitasking models are not supported. Additionally, the C++ encapsulation interface is not the default, the value is ignored for the **Pass fixed-size scalar root inputs by value for code generation** parameter.
- Stateflow considerations — If you have a Stateflow license, for a Stateflow chart that resides in a root model configured to use the `I/O arguments step method` function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, do one of the following to generate code:

  - In the Stateflow chart, clear the **Execute (enter) Chart At Initialization** check box.
  - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- Simscape Considerations — If a model root inport value connects to a Simscape block, insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- Reference Model Considerations — When building a referenced model that is configured to generate a C++ class interface:

  - Do not use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that has a continuous sample time or saves states.

- Do not use virtual buses as inputs or outputs to the referenced model when the referenced model uses the I/O arguments step method. When bus signals cross referenced model boundaries, either use nonvirtual buses or use the Default step method.

See Also

`coder.mapping.api.CodeMappingCPP` | `coder.mapping.api.get` | `coder.mapping.utils.create` | `find` | `getClassName` | `setClassName` | `getClassNamespace` | `setClassNamespace` | `getData` | `setData` | `getFunction` | `setFunction`

Related Examples

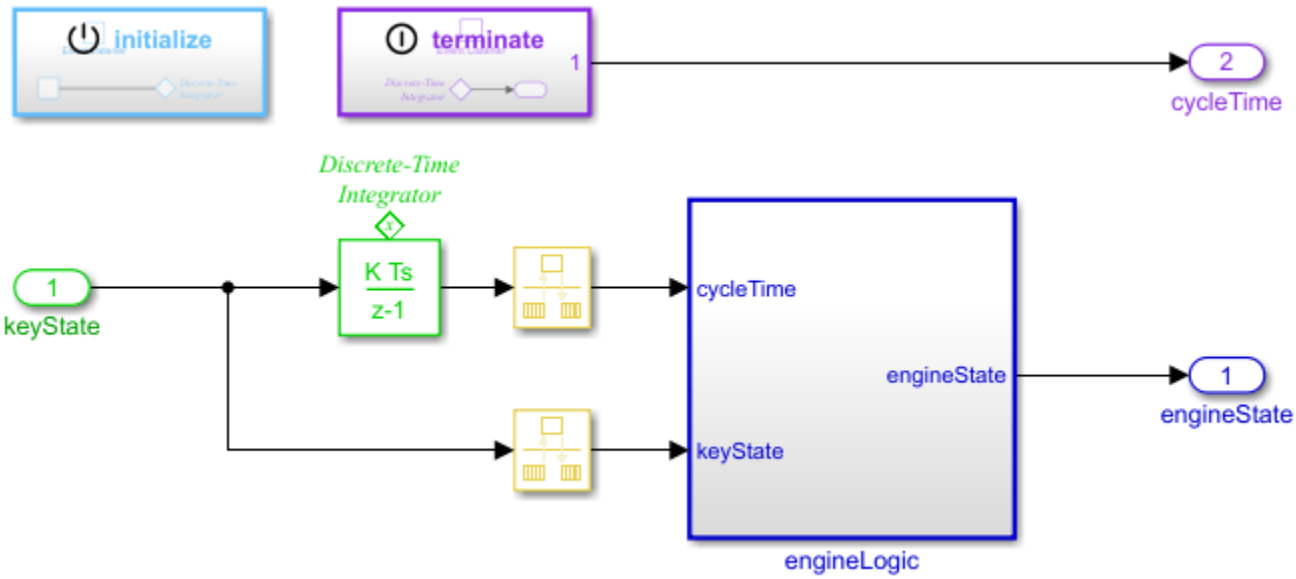
- “Configure C++ Class Interface for Rate-Based Models” on page 44-46
- “Configure C++ Class Interface for Export-Function Models” on page 44-50

More About

- “Interactively Configure C++ Interface” on page 44-2
- “Analyze Generated Data Code Interface” on page 54-13

Configure C++ Class Interface for Rate-Based Models

This example shows how to configure a C++ class interface for a rate-based model. In Simulink you can create and generate code for a rate-based modeling system that enables you to control the scheduling of model components. This example shows you how to customize the generated class name, namespace, class members, and class methods for an example model. This example, CppClassRateBased, simulates whether a vehicle engine is on or off depending if the ignition has been turned on.



Interface Goals and Requirements

A primary goal for configuring a C++ class interface for a rate-based model is to customize the generated code so that it easily integrates with your task scheduler. For this example, the goal is to update the name and namespace of the class to more meaningful values, distinguish the varied sample rates, and to simplify and encapsulate the remaining data and functions in the interface.

- To better reflect that this model is an engine status model, update the class name to `engine_status`.
- To prevent clashes with other symbols in a project and indicate that the generated code is from Simulink, update the class namespace to `sl`.
- To enable the configuration of the name and arguments of the base-rate periodic function, configure the inports and outports of the model as public class members without a get/set method.
- To enable the calibrate of shared values across multiple instances of the generated model class, configure the model parameters as private class members with a get/set method.
- To simplify the interface, configure parameter arguments and internal data as private without a get/set method.
- To make the other entry-point names more meaningful, update the names of the initialize and terminate entry-rate methods.

Interactive Example

Configure Model Class Name and Namespace

1. On the **C++ Code** tab, click **Code Interface** and select **Class Name & Namespace**.

2. To configure the class name, in the **C++ Class Name** field, update the name to `engine_status`.

3. To configure the class namespace, in the **C++ Class Namespace** field, update the name to `sl`.

4. Click **OK**.

Configure Model Data Elements as Class Members

1. Open the Code Mappings editor. On the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.

2. Configure the data visibility. For the model element categories:

- Set the **Inports** and **Outports** to `public` and `None`.
- Set the **Model parameters** to `private` and `Method`.
- Set the **Model parameter arguments** and **Signals, states, and internal data** to `private` and `None`.

Configure Model Functions as Class Methods

1. Open the **Functions** pane. In the Code Mappings editor, click the **Functions** tab.

2. To view a complete list of entry-point functions for your model, click the **Update Diagram** button.

3. Configure the periodic functions to distinguish the sample rates.

- To configure the base-rate periodic function, in the **Method Preview** column, click its method preview hyperlink.
- Configure the function name. In the Step Function Interface configuration dialog box **C++ Step Function Name** field, update the function name to `EngineEntrypoint`.
- Open the arguments viewer. In the dialog box, select **Configure arguments for Step function prototype** and click **Get default**.
- Configure the argument names. In the **C++ Identifier Name** column, click and edit the names to `keyState arg_keyState`, `engineState arg_engineState`, `cycleTime arg_cycleTime`.
- Configure the argument identifiers. In the **C++ Type Qualifier** column, select the appropriate identifier from the drop-down options. Configure `keyState arg_keyState` as `value`, `engineState arg_engineState` as `Pointer`, and `cycleTime arg_cycleTime` as `Pointer`.
- Change the order of the arguments. Click and drag the argument rows in the viewer to the following order: `keyState arg_keyState`, `engineState arg_engineState`, and `cycleTime arg_cycleTime`.
- Verify the arguments selections. Click **Validate**.
- To apply changes and exit the dialog box, click **OK**.

4. Configure the other entry-point methods.

- Configure the Initialize function name. In the **Method Name** column, click and edit the spreadsheet to change the name to `initIntegrator`.

- Configure the Terminate function name. In the **Method Name** column, click and edit the spreadsheet to change the name to `terminateReadIntegrator`.

5. Verify the method prototypes for all entry-point functions in the **Method Preview** column.

Generate the Interface

1. Generate code. To generate a C++ class interface, on the **C++ Code** tab, click **Build**.

2. View code. To view the generated code, on the tab click **View Code**. The generated code appears beside the model in the model workspace.

Programmatic Example

To programmatically configure a C++ class interface for the CppClassRateBased model, use this workflow:

Open the example model

```
model ='CppClassRateBased';
open_system(model);

% Get the C++ mapping object
cm = coder.mapping.api.get(model);
```

Configure Model Class Name and Namespace

```
% Configure the C++ class name
setClassName(cm,'engine_status');
% Configure the enclosing namespace
setClassNamespace(cm,'sl');
```

Configure Model Data Elements as Class Members

```
% Configure the inports and outport to pass external data directly to
% the base-rate periodic function in the modeled application. This
% configuration enables you to later configure the name and arguments for
% the base-rate periodic function
setData(cm, 'Inports', 'DataVisibility', 'public');
setData(cm, 'Outports', 'DataVisibility', 'public');
setData(cm, 'Inports', 'MemberAccessMethod', 'None');
setData(cm, 'Outports', 'MemberAccessMethod', 'None');

% Configure model parameters so that you can adjust calibration values
setData(cm, 'ModelParameters','DataVisibility','private');
setData(cm,'ModelParameters', 'MemberAccessMethod','Method');

% Configure Model Parameter Arguments and Iternal data as encapsulated and simple
setData(cm, 'ModelParameterArguments', 'DataVisibility', 'private');
setData(cm, 'ModelParameterArguments', 'MemberAccessMethod', 'None');
setData(cm, 'InternalData', 'DataVisibility', 'private');
setData(cm, 'InternalData', 'MemberAccessMethod', 'None');
```

Configure Model Functions as Class Methods

```
% To configure the periodic functions, use the find function to
% retrieve the periodic functions for the model
periodic_functions = find(cm, 'PeriodicFunctions');
```

```
% Configure the base-rate periodic function method name
setFunction(cm, periodic_functions, 'MethodName', 'EngineEntrypoint');

% Configure the base-rate periodic function arguments
setFunction(cm, periodic_functions,'Arguments',...
    '(const & keyState arg_keyState, * engineState arg_engineState, * cycleTime arg_cycleTime)')

% Configure the initialize and terminate function names
setFunction(cm,'Initialize','MethodName', 'initIntegrator');
setFunction(cm,'Terminate', 'MethodName', 'terminateReadIntegrator');
```

Generate The Interface

Build the application model to generate the C++ class interface.

See Also

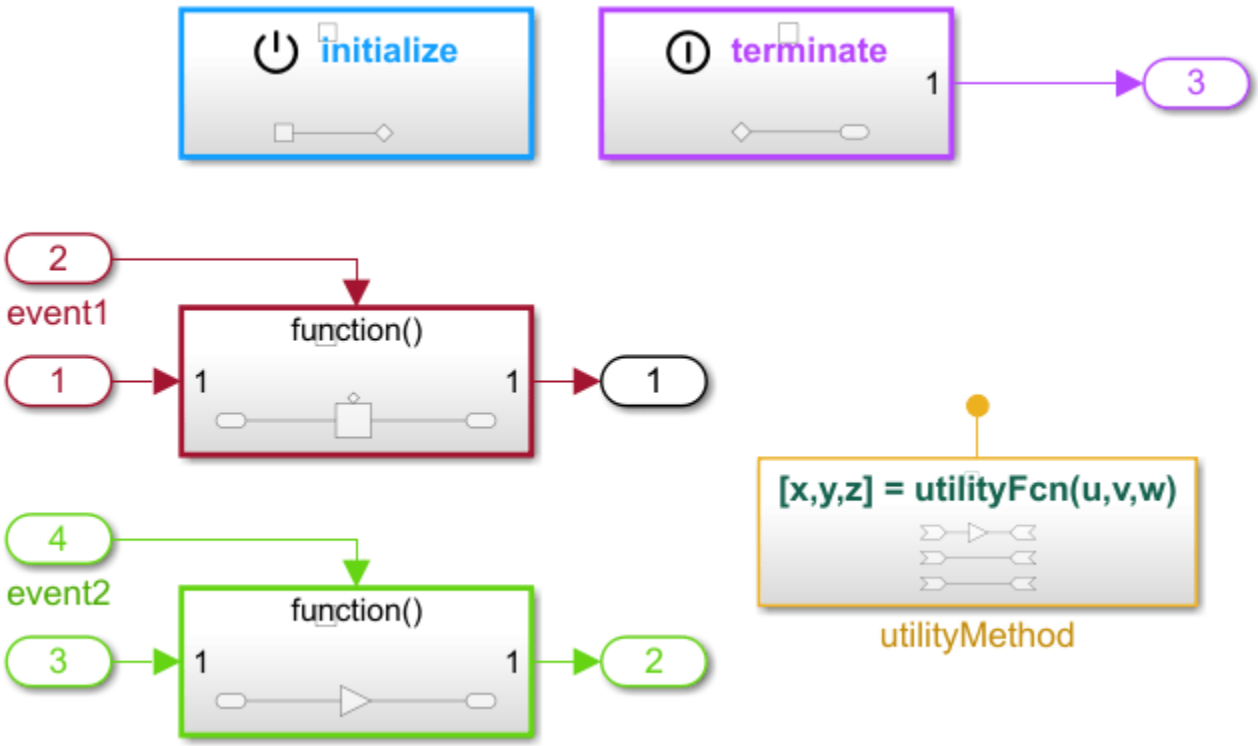
Code Mappings - C++ Editor | `coder.mapping.api.CodeMappingCPP`

Related Examples

- “Interactively Configure C++ Interface” on page 44-2
- “Programmatically Configure C++ Interface” on page 44-25

Configure C++ Class Interface for Export-Function Models

This example shows how to configure a C++ class interface for an export-function model. Export-function modeling enables you to generate code for event-triggered functions that can be integrated with an external environment or a scheduler. Functions are defined using Function-Call Subsystem, function-call Model, Simulink Function, and S-Function blocks. This example shows you how to customize the generated class name, namespace, class members, and class methods for an example model. This example, `CppClassExportFunction`, provides a set of functions you might customize in a typical export-function model.



Interface Goals and Requirements

A primary goal for configuring the generated C++ code from an export-function model is to be able to customize event-driven behavior. For this example, the goal is to update the name and namespace of the class to more meaningful values, customize the export functions, and to simplify and encapsulate the remaining data and functions in the interface.

- To better reflect the event-driven behavior in the model, update the class name to `triggered_events`.
- To prevent clashes with other symbols in a project and to indicate that the generated code is from Simulink, update the class namespace to `sl`.
- To simplify and encapsulate the data, set the model elements to private with a get/set method.
- To customize the functions, customize the names of the export-functions.

- To make the other entry-point names more meaningful, update the names of the initialize and terminate entry-point methods.

Interactive Example

Configure Model Class Name and Namespace

1. On the **C++ Code** tab, click **Code Interface** and select **Class Name & Namespace**.

2. To configure the class name, in the **C++ Class Name** field, update the name to `triggered_events`.

3. To configure the class namespace, in the **C++ Class Namespace** field, update the name to `sl`.

4. Click **OK**.

Configure Model Data Elements as Class Members

1. Open the Code Mappings editor. On the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.

2. Configure the data visibility. For the model element categories:

- Set the **Inports** and **Outports** to `private` and `Method`.
- Set the **Model parameters** to `private` and `Method`.
- Set the **Model parameter arguments** to `private` and `Method`.
- Set the **Signals, states, and internal data** to `private` and `Method`.

Configure Model Functions as Class Methods

1. Open the **Functions** pane. In the Code Mappings editor, click the **Functions** tab.

2. Configure the export function names.

- For the export function, `Export Function: event 1`, from the **Method Name** column, click and edit the spreadsheet to update the name to `subsystemEvent1`.
- For the export function, `Export Function: event 2`, from the **Method Name** column, click and edit the spreadsheet to update the name to `subsystemEvent2`.

3. Configure the Initialize function name. In the **Method Name** column, click and edit the spreadsheet to change the name to `initIntegrator`.

4. Configure the Terminate function name. In the **Method Name** column, click and edit the spreadsheet to change the name to `terminateReadIntegrator`.

5. Verify the method prototypes for all entry-point functions in the **Method Preview** column.

Generate the Interface

1. Generate code. To generate a C++ class interface, on the **C++ Code** tab, click **Build**.

2. View code. To view the generated code, on the tab, click **View Code**. The generated code appears beside the model in the model workspace.

Programmatic Example

To programmatically configure a C++ class interface for the `CppClassExportFunction` model, use this workflow:

Open the example model

```
model ='CppClassExportFunction';
open_system(model);

% Get the C++ mapping object
cm = coder.mapping.api.get(model);
```

Configure Model Class Name and Namespace

```
% Configure the C++ class name
setClassName(cm,'event_triggered');
% Configure the enclosing namespace
setClassNamespace(cm,'sl');
```

Configure Model Data Elements as Class Members

```
% Configure the inports and outports for the model
setData(cm, 'Inports', 'DataVisibility', 'private');
setData(cm, 'Outports', 'DataVisibility', 'private');
setData(cm, 'Inports', 'MemberAccessMethod', 'Method');
setData(cm, 'Outports', 'MemberAccessMethod', 'Method');

% Configure the model parameters
% static model parameters (shared across all instances)
setData(cm, 'ModelParameters','DataVisibility','private');
setData(cm,'ModelParameters', 'MemberAccessMethod','Method');

% Configure the model parameter agruments
setData(cm, 'ModelParameterArguments', 'DataVisibility', 'private');
setData(cm, 'ModelParameterArguments', 'MemberAccessMethod', 'Method');

% Configure the internal data
setData(cm, 'InternalData', 'DataVisibility', 'private');
setData(cm, 'InternalData', 'MemberAccessMethod', 'Method');
```

Configure Model Functions as Class Methods

```
% To configure the export functions, use the find function to
% retrieve all the export functions for the model
exported_functions = find(cm, 'ExportedFunctions');

% Configure the Exported Function names
setFunction(cm, exported_functions{1}, 'MethodName', 'subsystemEvent1');
setFunction(cm, exported_functions{2}, 'MethodName', 'subsystemEvent2');

% Configure the initialize and terminate function names
setFunction(cm,'Initialize','MethodName', 'initIntegrator');
setFunction(cm,'Terminate', 'MethodName', 'terminateReadIntegrator');
```

Generate The Interface

Build the application model to generate the C++ class interface.

See Also

Code Mappings – C++ Editor | `coder.mapping.api.CodeMappingCPP`

Related Examples

- “Interactively Configure C++ Interface” on page 44-2
- “Programmatically Configure C++ Interface” on page 44-25

Memory Sections in Embedded Coder

- “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2
- “Protect Global Data with const and volatile Type Qualifiers” on page 45-13

Control Data and Function Placement in Memory by Inserting Pragmas

For some applications, you can use pragmas and other code decorations to control the placement of data (global variables) and function definitions in memory. For example, a linker configuration file can define named sections in a `SECTIONS` directive and map each section to a range of memory addresses. In the C code, you include pragmas that assign global variables and functions to these named sections and, by extension, to the memory ranges. By controlling memory placement, you can:

- Generate code that is more efficient for your hardware.
- Modularize your application code for easier maintenance and modification later in the development process and after deployment.

With Embedded Coder and memory sections, you can:

- Apply default pragmas or other decorations to categories of model data and entry-point functions. To configure these defaults, use the Code Mappings editor. For example, you can:
  - Apply a default pragma to internal data, which includes block states that the code generator cannot eliminate through optimizations. You can apply a different default pragma to constant parameters, such as nonscalar parameters that the generated code must store in memory.
  - Apply a default pragma to categories of generated functions, including entry-point functions such as *`model`*`_step`.
- Override the default pragmas for individual data items such as block parameters, states, and signals. To do so, create your own storage class.
- Override the default pragmas for individual functions that correspond to atomic subsystems, which you can configure to appear in the generated code as separate functions with separate data. Use the subsystem parameters dialog box.

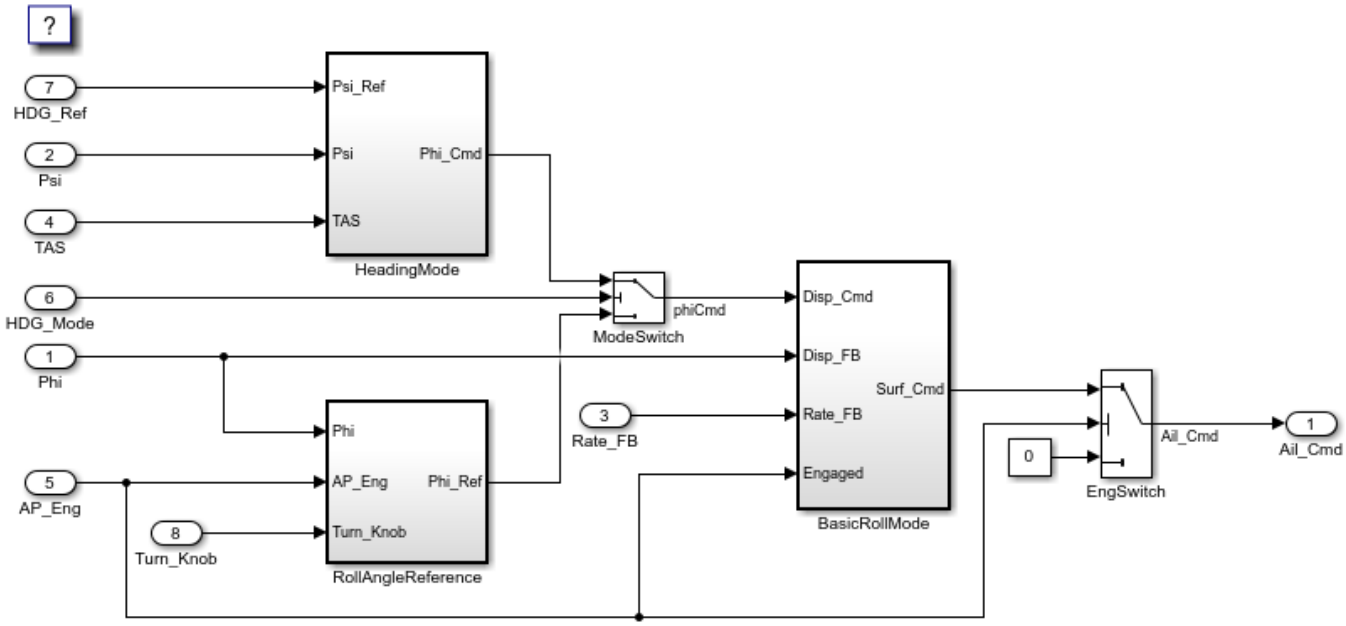
Insert Pragmas by Using Memory Sections

In this example, you configure the default memory placement for all of the data and functions of the algorithm represented by the example model `RollAxisAutopilot`. Then, for some signal data, you override the default placement.

Explore Example Model and Inspect Default Generated Code

1. Open the model `RollAxisAutopilot` by typing this command:

```
openExample('RollAxisAutopilot');
```



The model is configured to generate efficient production code. For example, the configuration parameter **Default parameter behavior** is set to `Inlined`.

2. In the **Apps** gallery, under **Code generation**, click **Embedded Coder**. The **C Code** tab opens. Generate code from the model.
3. In the code generation report, inspect the file `RollAxisAutopilot.h`. The file defines structure types that represent data that the algorithm needs. For example, the file defines a structure type that represents block states such as the states of Discrete-Time Integrator blocks.

   ```
   /* Block signals and states (default storage) for system '<Root>' */
   typedef struct {
     real32_T FixPtUnitDelay1_DSTATE;     /* '<S7>/FixPt Unit Delay1' */
     real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
     int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
   } DW;
   ```

   The file also declares entry-point functions for the model.

   ```
   /* Model entry point functions */
   extern void RollAxisAutopilot_initialize(void);
   extern void RollAxisAutopilot_step(void);
   ```
4. Inspect `RollAxisAutopilot.c`. This file defines global structure variables to store the data. The file also defines the functions.

In this example, assume your linker configuration file defines named sections `MYALGORITHM_DATA` and `MYALGORITHM_CODE` in a `SECTIONS` directive. You can configure the model so that the generated code includes pragmas, placing the memory allocated for the data and functions in these named sections.

- For a global variable named `myVar`, the pragma syntax is:

  ```
  #pragma SEC_MYALGORITHM_DATA("myVar")

  double myVar;
  ```

- For a function named `myFunction`, the pragma syntax is the same except for the section name:

  ```
  #pragma SEC_MYALGORITHM_CODE("myFunction")

  void myFunction(void)
  ```

Create Memory Sections

In this example, you use two pragmas with different syntaxes, so you must create two memory sections.

1. Open the Embedded Coder app. On the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.
2. In the **Embedded Coder Dictionary** dialog box, select the **Memory Sections** tab and click the **Add** button.
3. For the new memory section, set these options:
   - **Name** to `MYALGORITHM_DATA`.
   - **Statements Surround** to `Each variable`.
   - **Pre Statement** to `#pragma SEC_MYALGORITHM_DATA("$N")`. The token `$N` stands for the name of each variable that uses the memory section.
4. Create another, similar memory section that corresponds to `MYALGORITHM_CODE`.
5. Close the Embedded Coder Dictionary.

Configure Default Pragmas for Data and Functions

1. On the **C Code** tab, select **Code Interface** > **Default Code Mappings** or **Code Interface** > **Individual Element Code Mappings**
2. In the **Data Defaults** tab, expand **Inports and Outports**, **Signals** and **Parameters**.
3. In the table, select the **Inports** row.
4. Click the icon and set **Memory Section** to `MYALGORITHM_DATA`.
5. For the other rows in the table, set **Memory Section** to `MYALGORITHM_DATA`.
6. Under **Function Defaults**, for each row in the table, set **Memory Section** to `MYALGORITHM_CODE`.
7. In your current folder, delete the existing `slprj` folder.
8. Configure the model to generate only code. Select the configuration parameter **Configuration Parameters** > **Generate code only**.
9. Click **Generate Code**.

Now, the `RollAxisAutopilot.c` file applies the pragmas to the definitions of the structure variables and the functions. For each category of data and functions that you configured in the Code Mappings editor, the code applies a pragma. For example, the code applies the `MYALGORITHM_DATA` pragma to each of the structures that store block states, root-level inputs, and root-level outputs.

```
/* Block signals and states (default storage) */
#pragma SEC_MYALGORITHM_DATA("rtDW")

DW rtDW;

/* External inputs (root inport signals with default storage) */
#pragma SEC_MYALGORITHM_DATA("rtU")

ExtU rtU;

/* External outputs (root outports fed by signals with default storage) */
#pragma SEC_MYALGORITHM_DATA("rtY")

ExtY rtY;
```

Retain Default Memory Section After Applying a Default Storage Class or Function Template

In the Code Mappings editor, you can use the **Storage Class** and **Function Customization Template** columns to control the default appearance of data and functions in the generated code. When you use these columns to apply a setting other than `Default`, the Code Mappings editor discards the memory section that you applied in the Property Inspector. To retain the memory section, use the Embedded Coder Dictionary to create a storage class or function template, then apply the memory section to that storage class or function template.

In this example, you configure the nonparameter data of the model, such as signals and states, to appear in the same structure by creating a storage class. To retain the `MYALGORITHM_DATA` memory section, you apply the memory section to the storage class.

1. In the Embedded Coder Dictionary for the model, select the **Storage Classes** tab and click the **Add** button.
2. For the new storage class, set:
   - **Name** to `STRUCT_DATA`.
   - **Storage Type** to `Structured`.
   - **Memory Section** to `MYALGORITHM_DATA`.
3. In the Code Mappings editor, under **Data Defaults**, in the **Storage Class** column, select `STRUCT_DATA` for these rows:
   - **Inports**
   - **Outports**
   - **Signals, states and internal data**
4. Generate code from the model.
5. Inspect `RollAxisAutopilot.c`. Now, the file defines a single structure variable that contains the nonparameter data, applying the pragma to that variable.

```
/* Storage class 'STRUCT_DATA' */
#pragma SEC_MYALGORITHM_DATA("STRUCT_DATA_RollAxisAutopilot")

RollAxisAutopilot_STRUCT_DATA STRUCT_DATA_RollAxisAutopilot;
```

Incorporate Default Memory Section into a Storage Class for Individual Mapping

To override the default storage classes that you specify in **Code Mappings** > **Data Defaults**, apply a storage class directly to a data item by using the Code Mappings editor. Directly applying a storage class other than `Auto` or `Model default` bypasses the default memory section that you specify in **Data Defaults**. To retain the memory section, create a storage class in the Embedded Coder Dictionary and apply the desired memory section to it. Then, apply the storage class to individual data items in the Code Mappings editor. Other models cannot access the storage class and memory section that you define in the Embedded Coder Dictionary. To share the storage class and memory section sections among other models, store the Embedded Coder Dictionary in a Simulink Data Dictionary data (`sldd`) outside of your model. Then, share the dictionary between the target models. For more information, see “Migrate Data Interface Configuration from Model File to Shared Data Dictionary” on page 32-21.

In `RollAxisAutopilot`, in the `BasicRollMode` subsystem, the three Gain blocks represent the parameters of a PID control algorithm. In the “Retain Default Memory Section After Applying a Default Storage Class or Function Template” on page 45-5 section, configure the nonparameter data of the model, such as signals and states, to appear in the same structure by applying the default

storage class STRUCT_DATA. In this example, you configure the output signals of the Gain blocks so that the generated code allocates memory for them in the MYALGORITHM_DATA section but does not appear in the default storage class structure. Define a new storage class myStore in the Embedded Coder Dictionary and apply the desired memory section MYALGORITHM_DATA to it. Then, directly apply the storage class to the output signals of the Gain blocks.

1. In the Embedded Coder Dictionary of the model, select the **Storage Classes** tab and click the **Add** button.
2. For the new storage class, set:
   - **Name** to myStore.
   - **Storage Type** to Structured.
   - **Memory Section** to MYALGORITHM_DATA.
3. In the model, navigate into the BasicRollMode subsystem. The signals of the subsystem are not automatically populated into the Code Mappings editor. You need to manually add signals by pausing on the ellipsis that appears above or below a signal line to open the action bar. Click the **Add Signal** button. The button is also available in the Code Mappings editor on the **Signals/States** tab.
4. After you add the output signals of the Gain blocks, the signal rows appear in the **Code Mappings** > **Signals/States** tab. Open the Property Inspector by selecting the signal rows. In the Code section, set these properties:
   - **Storage Class** to myStore.
   - **Identifier** to the block names.
5. Generate code from the model.
6. Inspect the generated file RollAxisAutopilot.c. The file defines a structure variable myStore_RollAxisAutopilot and applies the pragma to the variable.

```
/* Storage class 'myStore' */
#pragma SEC_MYALGORITHM_DATA("myStore_RollAxisAutopilot")

RollAxisAutopilot_myStore myStore_RollAxisAutopilot;
```

7. Inspect the generated file RollAxisAutopilot.h. The structure variable contains the output signals of the Gain blocks. The output signals are now in a different structure but stored in the same MYALGORITHM_DATA memory section.

```
/* Storage class 'myStore', for system '<Root>' */
typedef struct {
  real32_T DispGain;                   /* '<S1>/DispGain' */
  real32_T RateGain;                   /* '<S1>/RateGain' */
  real32_T IntGain;                    /* '<S1>/IntGain' */
} RollAxisAutopilot_myStore;
```

Configure Pragma to Surround Groups of Definitions

If your build toolchain requires that a pragma or other decoration surround multiple definitions of variables or functions at once, in an Embedded Coder Dictionary or the Custom Storage Class Designer, set **Statements surround** to Group of variables (the default in the Custom Storage Class Designer).

Override Default Memory Placement for Individual Data Elements

After you configure memory section defaults in the Code Mappings editor (see “Configure Default C Code Generation for Categories of Data Elements and Functions” on page 33-15), to override these

default settings for individual data elements (signals, parameters, and states), create a storage class and any required memory sections by using the Embedded Coder Dictionary. For more information see, “Choose Where to Create and Store Memory Section Definition” on page 45-7. When you create the storage class, set the **Memory section** property to the appropriate memory section. Then, use the Code Mappings editor to apply the storage class to individual data elements.

Choose Where to Create and Store Memory Section Definition

To define a memory section, you must choose where to create it: in an Embedded Coder Dictionary or in a Simulink Shared Dictionary.

- If you need to use the memory section only in the Code Mappings editor, define the memory section in an Embedded Coder Dictionary.

  If you need to use the memory section only in the Code Mappings editor and want to share it with other models, define the memory section in a Simulink® data dictionary (sldd). For more information, see “Share Code Interface Configuration Between Models” on page 32-19.
- If you need to use the memory section outside of the Code Mappings editor, define the memory section in a package. For example, define memory sections in a package to configure memory sections of an atomic Subsystem. For more information, see “Override Memory Section for Atomic Subsystem” on page 45-7.

Share Memory Section Definition Between Models

- If you define the memory section in the Embedded Coder Dictionary of a model, you cannot use the memory section in other models. To share the memory section, migrate the definition to a Simulink data dictionary (`sldd`). Then, share the dictionary between the target models. For more information, see “Migrate Data Interface Configuration from Model File to Shared Data Dictionary” on page 32-21.
- If you define the memory section in a package by using the Custom Storage Class Designer, any model can use the memory section. For more information, see “Create Storage Class by Using Custom Storage Class Designer” on page 40-23. Add the folder containing the package folder to the MATLAB path (see “What Is the MATLAB Search Path?”).

Override Default Memory Placement for Subsystem Functions and Data

When you use atomic subsystems to partition the generated code into functions (see “Generate Subsystem Code as Separate Function and Files” on page 4-7), you can apply different memory sections to the functions and data of each subsystem. You can also specify that a subsystem not use a memory section.

- To use different memory sections to override the model-level defaults that you set in the Code Mappings editor, see “Override Memory Section for Atomic Subsystem” on page 45-7.
- To specify that a subsystem not use a memory section (in other words, to prevent the subsystem from inheriting the model-level defaults), see “Specify That Atomic Subsystem Does Not Use a Memory Section” on page 45-8.

Override Memory Section for Atomic Subsystem

The memory sections that you specify for a subsystem override the model-level defaults that you set in the Code Mappings editor. Use this technique to aggregate the data and instruction code for

subroutines or subcomponents (represented by subsystems) into different regions of memory. To apply a memory section directly to an atomic subsystem, define the memory section in a package by using the Custom Storage Class Designer. You cannot use a memory section that you define in an Embedded Coder Dictionary. Then, load the package into the model and configure the subsystem block parameters to specify the memory section.

To create the memory section:

1. In your current folder, create a folder named `+myPackage`. The folder defines a package named `myPackage`. For more information, see “Create Data Class Package” on page 40-14.

   To make the package available outside of your current folder, optionally, you can add the folder containing the `+myPackage` folder to the MATLAB path.
2. Open the Custom Storage Class designer.

   ```
   cscdesigner('myPackage');
   ```
3. In the Custom Storage Class Designer, select the **Memory Section** tab.
4. Click **New**.
5. For the new memory section, set these options:
   - **Name** to `MYALGORITHM_DATA`.
   - **Statements surround** to `Each variable`.
   - **Pre statement** to `#pragma SEC_MYALGORITHM_DATA("$N")`.
6. Click **Apply** and **Save**.
7. Set your current folder to the folder that contains the `+myPackage` folder.

To apply the memory section in an atomic subsystem:

1. Configure the Embedded Coder Dictionary of the model to load the target package as described in “Refer to Code Generation Definitions in a Package”.
2. To configure the target subsystem to specify the memory section, open the subsystem parameters dialog box. On the **Code Generation** tab:
   - Set **Function packaging** to `Nonreusable function` or `Reusable function` (for reentrant code).
   - If you set **Function packaging** to `Nonreusable function`, to enable configuration of memory sections for the subsystem data, select **Function with separate data**. If you do not select **Function with separate data**, the subsystem data inherits memory sections from the model or, if applicable, a parent subsystem.
   - To apply the `MYALGORITHM_DATA` memory section directly to the subsystem functions and data, use these block parameters:
     - **Memory section for initialize/terminate functions**
     - **Memory section for execution functions**
     - **Memory section for constants**
     - **Memory section for internal data**
     - **Memory section for parameters**

Specify That Atomic Subsystem Does Not Use a Memory Section

By default, subsystem functions and data inherit the model-level memory sections that you specify for relevant function and data categories in the Code Mappings editor. For example, if you specify a

function customization template for the **Execution** category, and that template carries a memory section, the memory section applies to subsystem execution functions as well as model entry-point execution functions.

To specify that a subsystem does not use a memory section, open the subsystem parameters dialog box. On the **Code Generation** tab, set these parameters to `Default`:

- **Memory section for initialize/terminate functions**
- **Memory section for execution functions**
- **Memory section for constants**
- **Memory section for internal data**
- **Memory section for parameters**

With these settings, the subsystem does not use a memory section for the data or functions that each parameter represents.

Limitations and Other Considerations

- The settings that you specify for an atomic, nonreusable subsystem with separate data apply only to the data and functions of that subsystem, not to data in similarly configured child subsystems. Atomic, nonreusable child subsystems with separate data can inherit memory sections from the containing model, not from the parent subsystem.
- If you use **Build This Subsystem** or **Build Selected Subsystem** to generate code for an atomic subsystem that specifies memory sections, the code generator ignores the subsystem-level specifications and uses the model-level specifications instead. For information about building subsystems, see “Generate Code and Executables for Individual Subsystems”.

Share Memory Section Between Packages (Package Memory Sections Only)

Packages can access and use memory sections that are defined in other packages, including custom packages and built-in packages such as `Simulink`. Only one copy of the memory section exists, in the package that defines it. Other packages refer to the memory section by pointing to it in its original location. Changes to the memory section, including changes to a built-in memory section in later MathWorks product releases, are immediately available in every referencing package.

To configure a package to refer to a memory section that is defined in another package:

1. Open the Custom Storage Class Designer. At the command prompt, enter `cscdesigner`.
2. Select the **Memory Section** tab.
3. Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.
4. In the **Memory section definitions** pane, select the existing definition below which you want to insert the reference.
5. Click **New Reference**.

   A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties.

6. Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.
7. Set **Refer to memory section in package** to specify the package that contains the memory section that you want to reference.
8. Set **Memory section to reference** to specify the memory section to be referenced.
9. Click **OK** or **Apply** to save the changes to memory. To save the changes permanently, click **Save**.

Control Appearance of Memory Section Drop-Down List (Package Memory Sections Only)

When you apply a package memory section, you select the memory section from a drop-down list. To control the order of the memory sections in the list, in the Custom Storage Class Designer, use the **Up** and **Down** buttons. The order of memory sections in drop-down lists matches the order in the Custom Storage Class Designer.

Protect Definitions of Package Memory Sections (Package Memory Sections Only)

When you click **Save** in the Custom Storage Class Designer, the Designer saves memory section and custom storage class definitions into the `csc_registration.m` file in the package folder. To determine the location of this file, in the Custom Storage Class Designer, inspect the value of **Filename**.

You can prevent changes to the memory section definitions of an entire package by converting the `csc_registration.m` file from a MATLAB file to a P-file. Use the `pcode` function.

A best practice is to keep `csc_registration.m` and `csc_registration.p` in your package folder. That way, if you need to modify the memory sections by using the Designer, you can delete `csc_registration.p` and later regenerate it after you finish the modifications. Because the P-coded version of the file takes precedence, while both files exist in the package, the memory sections are protected.

Limitations

- The code generator does not apply memory sections to data that uses these built-in storage classes:
  - `ExportedGlobal`
  - `ImportedExtern`
  - `ImportedExternPointer`
- In the Custom Storage Class Designer, the storage type qualifier that you specify for a memory section by using the **Qualifier** text box affects only data items that use a storage class setting other than these built-in storage classes:
  - `ExportedGlobal`
  - `ImportedExtern`
  - `ImportedExternPointer`

  The code generator omits the qualifier from other data categories.

- When you create a subsystem in a custom block library, you cannot specify memory sections for the subsystem definition in the library. Instead, specify memory sections for the subsystem instances that you place in your models.

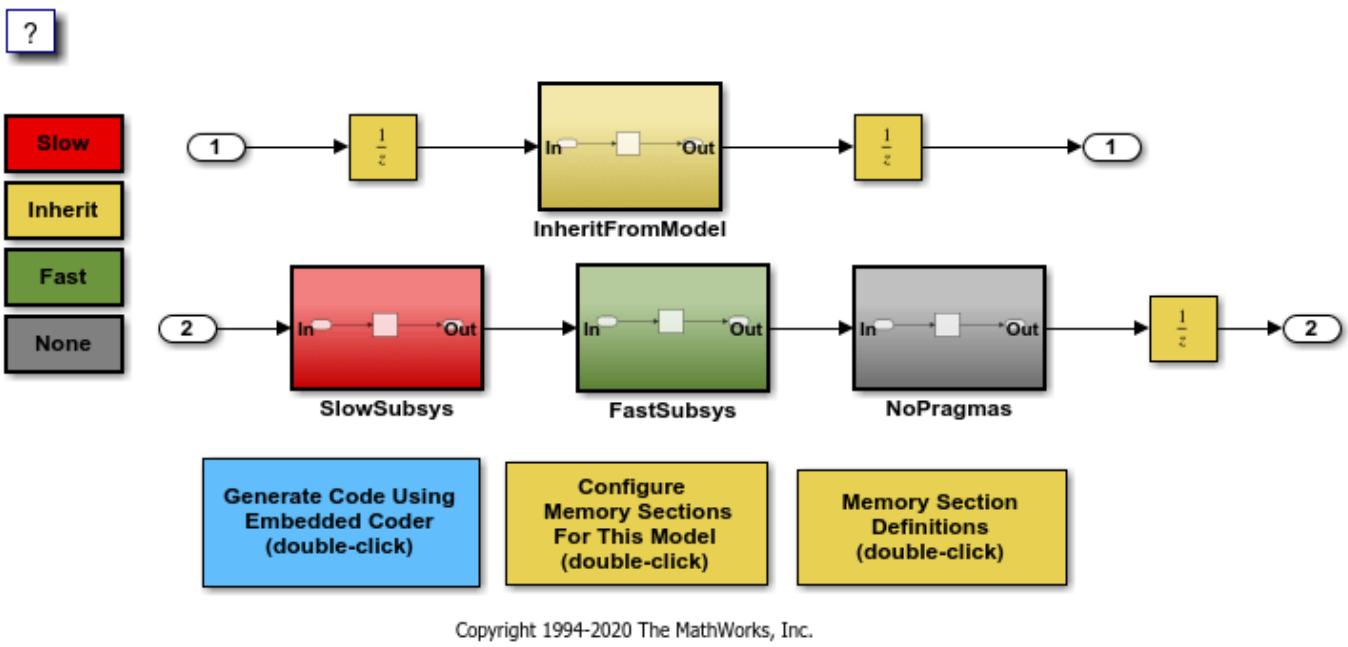
Insert Pragmas for Functions and Data in Generated Code

This example shows how to insert pragmas in generated code to control the placement of data and functions in memory. To configure memory placement for subsystem functions, use the memory sections that you define in a package. This example illustrates how to use memory sections defined in a package to insert pragmas for functions and data in the generated code.

Explore Example Model

Open the example model.

```
open_system('rtwdemo_memsec')
```



1. To view the memory sections in the ECoderDemos package, click the **Memory Section Definitions** button in the model and then select the **Memory Sections** tab.
2. To configure the memory sections for functions and data, click the **Configure Memory Sections For This Model** button. In the Code Mappings editor, select the appropriate row, and then specify memory sections in the Property Inspector.
3. The model-level settings are the default settings for atomic subsystems. Open the subsystem parameter drop-down lists to see the memory section settings for each of the atomic subsystems in the model. You can override the default model-level setting by configuring the subsystem block parameters.

4 To generate code, click the **Generate Code Using Embedded Coder** button. A code generation report opens automatically. Inspect the data and function definitions in the `.c` files and observe how the generated pragmas correspond to the specified memory sections.

See Also

Related Examples

- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2
- “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14
- “Protect Global Data with const and volatile Type Qualifiers” on page 45-13
- “Data Structures in the Generated Code” on page 36-31

Protect Global Data with const and volatile Type Qualifiers

In C, you use the type qualifier `const` to prevent code in an application from assigning a new value to a variable. In an application where an external actor (for example, a hardware device) can manipulate the value of a variable, you use the keyword `volatile` to prevent a compiler from optimizing the assembly code in a way that compromises the integrity of the variable value. You can also use `volatile` to prevent a compiler from eliminating storage for `const` data, such as a parameter that has a value that you want to tune during execution.

- To apply the qualifiers to an individual data item in a model, including a custom structure that you create by using a nonvirtual bus or a parameter structure, apply the appropriate predefined storage class directly to the data item. The storage class prevents optimizations such as **Default parameter behavior** from eliminating storage for the data item. For an example, see “Type Qualifiers” on page 26-12. For information about optimizations that a directly applied storage class prevents, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49.

  If the predefined storage classes do not meet your requirements, you can create your own custom storage class. To make your custom storage class apply the qualifiers, in the Custom Storage Class Designer, set **Memory section** to the appropriate built-in memory section or to a memory section that you create. For more information, see “Create Storage Classes by Using the Custom Storage Class Designer” on page 40-14.
- You can apply the qualifiers to a category of model data by default, such as parameters or states. As you add blocks to a model, new data items in these categories carry the qualifiers that you specify. For more information, see “Configure Default Code Generation for Data” on page 33-16.

  If the predefined storage classes do not meet your requirements, you can create your own by using an Embedded Coder Dictionary. In the Dictionary, for your new storage class, select the appropriate check boxes under **Qualifiers**. For more information, see “Create Code Definitions for Use in the Code Mappings Editor” on page 32-2.

Maintain const Correctness for Arguments of Entry-Point Functions

When your external code calls a generated entry-point function and passes `const` data through an argument (formal parameter) of the function, to make the corresponding argument in the function definition `const`, customize the execution (step) entry-point function interface.

- To configure the step entry-point function interface for a model, see “Configure Name and Arguments for Individual Step Functions” on page 33-120.
- To configure the step entry-point function interface for a Simulink Function block, see “Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2.

Incorrect Results or Undefined Behavior When Passing Volatile Data to a Generated Function

The generated code can define and call functions other than model entry-point functions. For example, you can configure an atomic subsystem to appear in the code as a separate function. Also, lookup table blocks, such as n-D Lookup Table, typically yield separate utility functions.

When the generated code defines a function that has an argument (formal parameter), the function definition does not apply `volatile` to the argument. Therefore, when other generated code or your

external code calls the function and passes a volatile variable as the value of the argument, the called function implicitly casts away the volatility.

If your application executes the called function while the value of the volatile data changes, the function can yield incorrect results or undefined behavior. In particular, for lookup table data that you prepare for calibration, by applying `const` and `volatile`, make sure that you do not calibrate the data while your application executes the lookup utilities.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67

Array Layout

Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks

Simulink® Coder™ supports row-major array layout for code generation. You can integrate existing applications that use row-major array layout with the generated code in row-major array layout. When you switch an existing model with lookup table (LUT) blocks from the column-major array layout to the row-major array layout, it is recommended to convert the LUT blocks from the column-major algorithm to the row-major algorithm. The code generated by using row-major algorithm performs with the best speed and memory usage when operating on table data with row-major array layout. The code generated by using column-major algorithm performs best with column-major array layout.

This example shows the workflow of converting a model with LUT blocks from column-major layout to row-major layout to achieve the best performance on a row-major array layout.

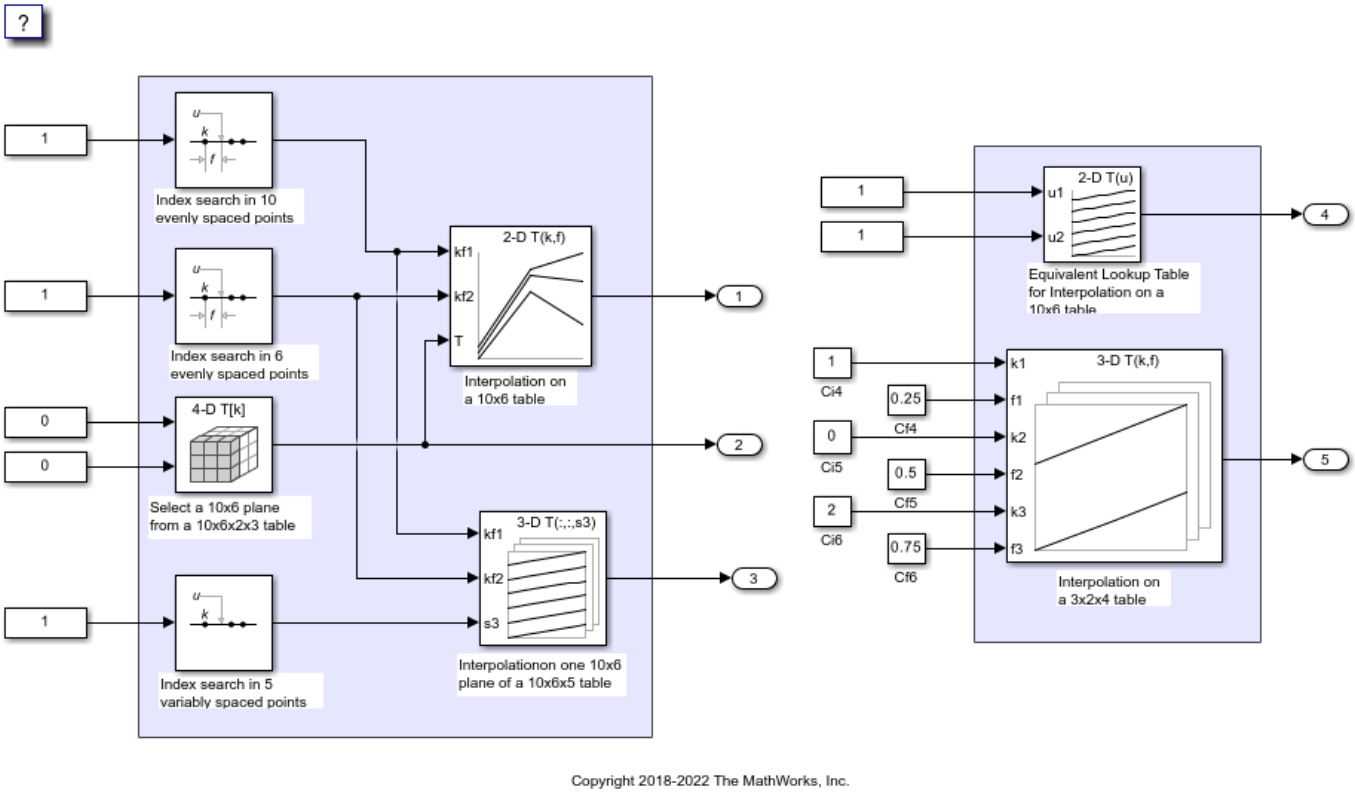
In this example, you:

- Identify the array layout and select the optimized algorithm.
- Preserve semantics through table permutation.
- Generate code by using a row-major algorithm and an array layout.

Simulate and Generate Code by Using Column-Major Algorithms

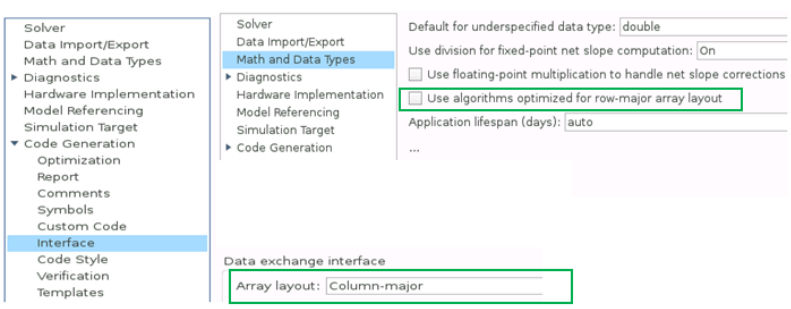
1. Open the example model `RowLUTColToRow`.

```
open_system('RowLUTColToRow');
```



By default, Simulink configures a model to use column-major algorithms and a column-major array layout. These parameters are the configuration parameters in the Configuration Parameters dialog box.

- Math and Data Types > “Use algorithms optimized for row-major array layout” — This parameter affects simulation and code generation.
- Code Generation > Interface > Array layout — This parameter affects only code generation.



2. On the **Modeling** tab, click **Run** to simulate the model and observe the output logged in workspace variable `yout`.

3. Change your current folder in MATLAB® to a writable folder. On the **C Code** tab, click **Build** to generate C code.

Select Optimized Algorithms for Row-Major Array Layout

Table data with row-major array layout is frequently used in the field of calibration. To interface the row-major table data with the existing model, update the column-major model to operate efficiently on row-major table data.

Use the algorithm that is optimized for the specified array layout to achieve the best performance. For example, use row-major algorithms when **Array layout** is set as `Row-major` during code generation.

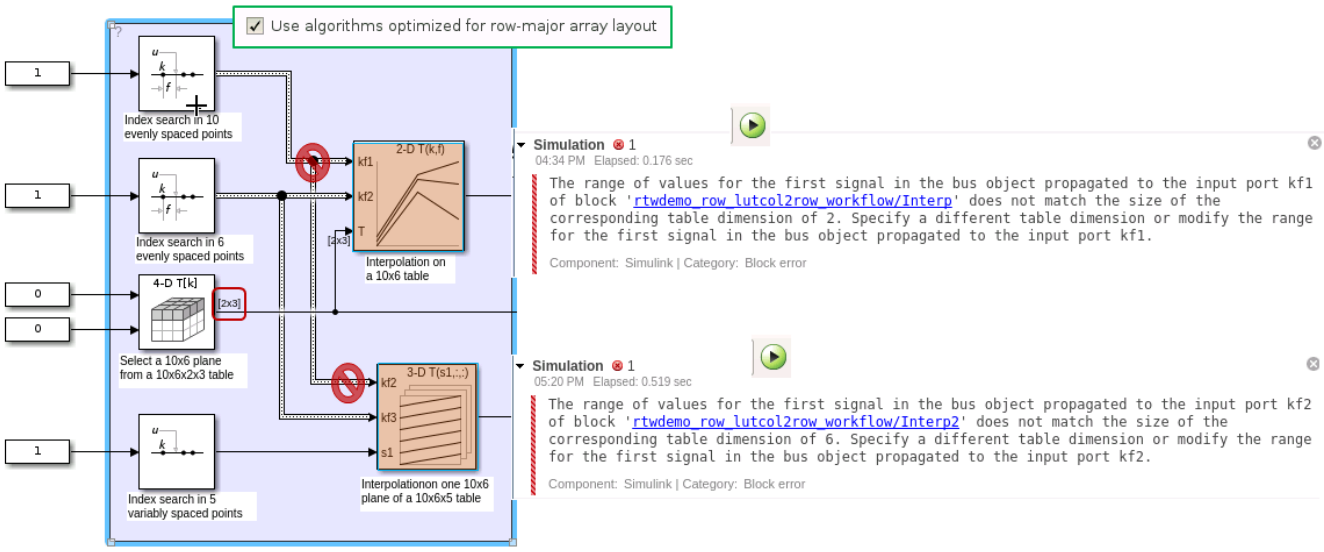
Array Layout	Algorithm	Cache-Friendly Algorithm
Column-major	Column-major	✓ (Recommended)
Row-major	Row-major	✓ (Recommended)
Row-major	Column-major	⚠ (Not recommended)
Column-major	Row-major	⚠ (Not recommended)

1. To enable row-major algorithms, open the Configuration Parameters dialog box. On the **Math and Data Types** pane, select the configuration parameter **Use algorithms optimized for row-major array layout**. Alternatively, in the MATLAB Command Window, enter:

```
set_param('RowLUTColToRow','UseRowMajorAlgorithm','on');
```

2. Click **Run** to simulate the model. Simulink reports errors because it encounters inconsistent breakpoint and table data between prelookup and interpolation blocks. The causes of this error are the two semantic changes that occur when you switch from column-major algorithms to row-major algorithms, that is, when you:

- Select a plane from a 3-D table in interpolation block.
- Select a plane from a 4-D table through a direct lookup table block.



Dimension of the Selected Plane (Semantic Changes)

	Column-major algorithm	Row-major algorithm
Select a plane from a 4-D table (10x6x2x3)	10x6	2x3
Select a plane from a 3-D table (10x6x5)	10x6	6x5

Preserve Semantics by Using Table Permutation

1. For subtable selection before interpolation, or direct lookup that outputs a vector or 2-D matrix, the model semantics change when you switch from a column-major algorithm to a row-major algorithm by selecting the configuration parameter **Use algorithms optimized for row-major array layout**. To preserve the semantics and fix the previous errors, permute the table data by using these commands:

```
T4d_str = get_param('RowLUTColToRow/Direct LUT','Table');
set_param('RowLUTColToRow/Direct LUT','Table',...
['permute(',T4d_str,',[3,4,1,2])']);

T3d_str = get_param('RowLUTColToRow/Interp2','Table');
set_param('RowLUTColToRow/Interp2','Table',...
['permute(',T3d_str,',[3,1,2])']);
```

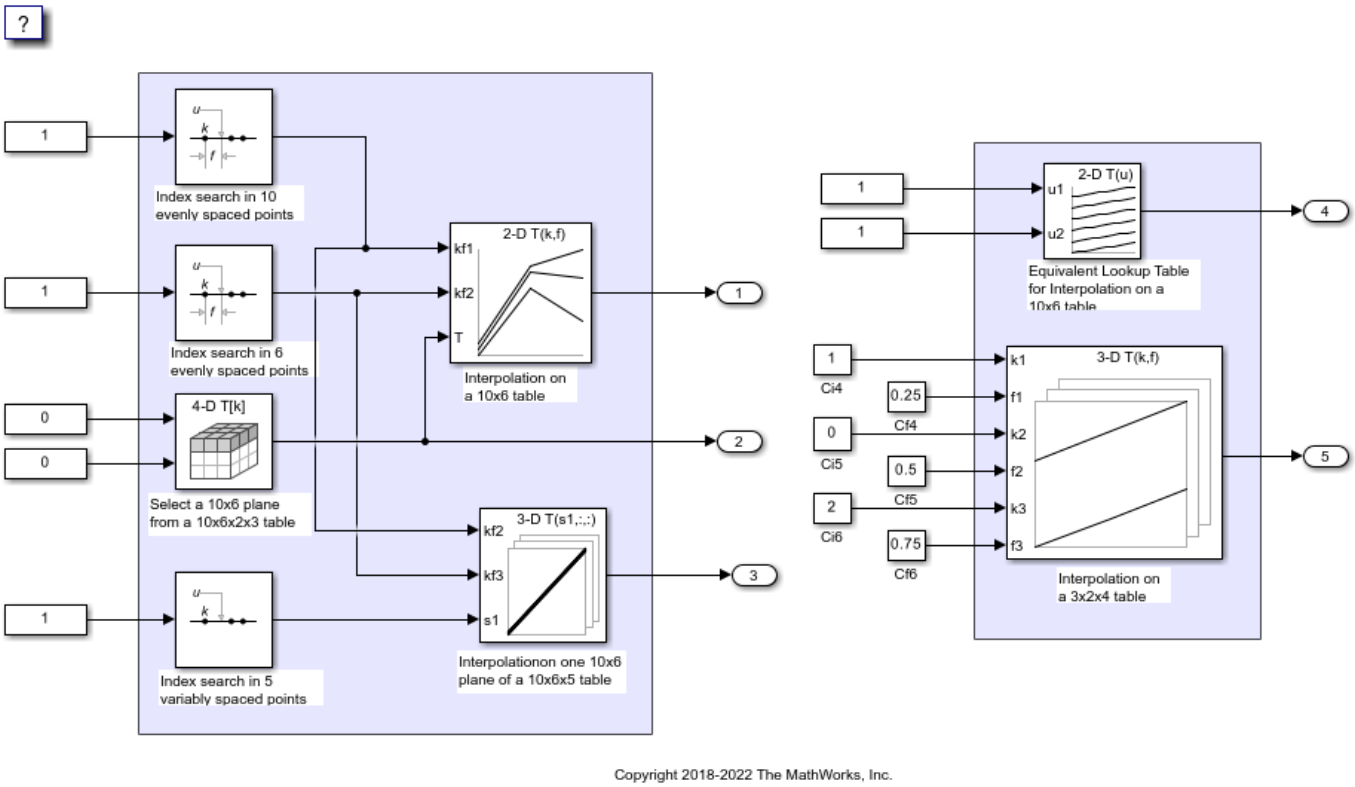
2. Before you import table data from a file, you must permute the table data in the file. This permutation keeps the table tunable throughout the simulation and code generation workflow.

Code Generation by Using Row-Major Algorithm and Array Layout

After permuting the table data, Simulink configures the model `RowLUTColToRow` for row-major simulation. The model is equivalent to the preconfigured model `RowLUTColToRowPreconfigured` that has permuted table data and uses a row-major algorithm.

1. Open the example model `RowLUTColToRowPreconfigured`.

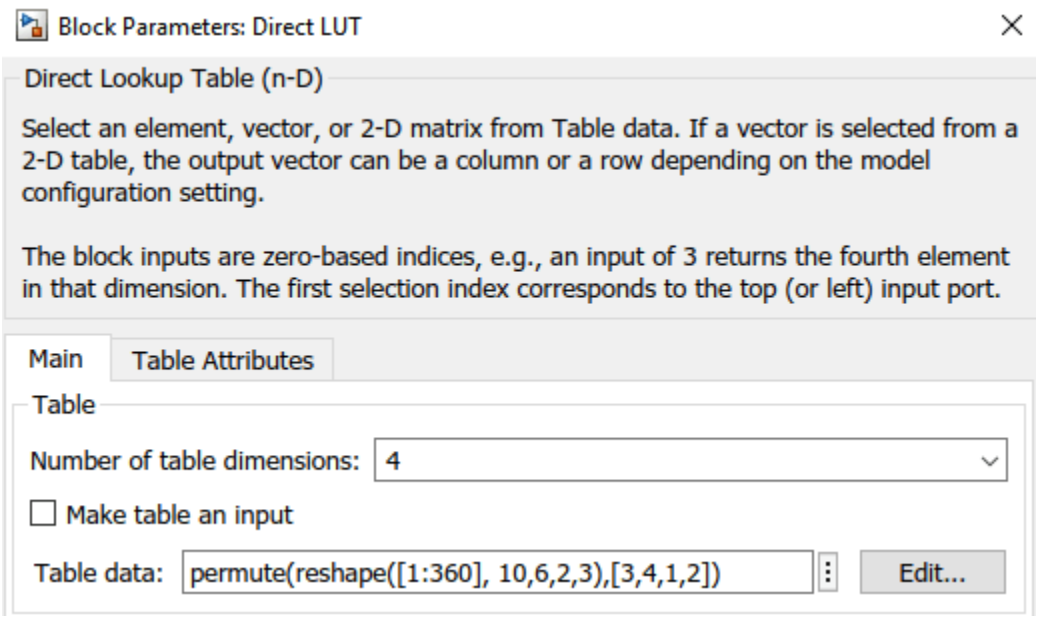
```
open_system('RowLUTColToRowPreconfigured');
```

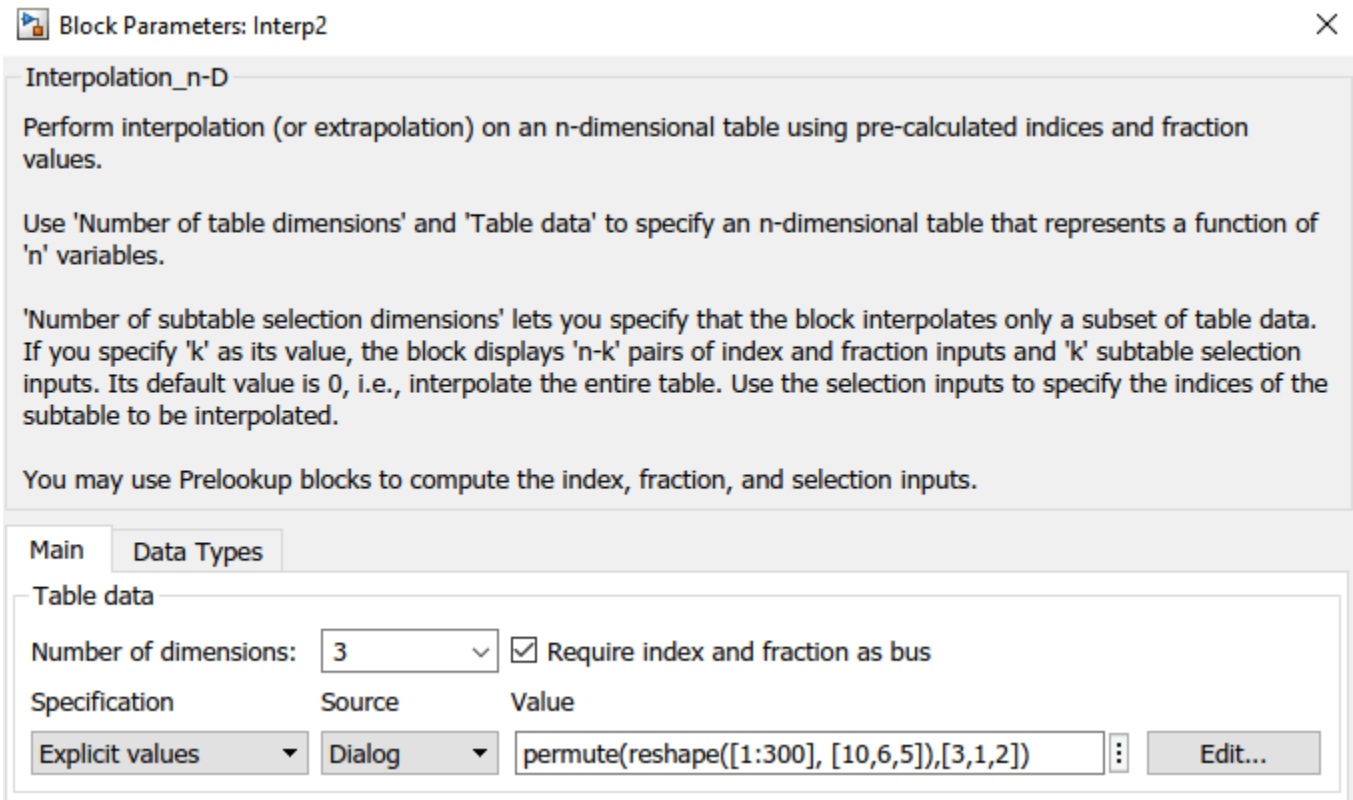


2. To set up these models for row-major code generation, open the Configuration Parameters dialog box. In addition to enabling the **Use algorithms optimized for row-major array layout** configuration parameter, on the **Code Generation > Interface** pane, set the configuration parameter **Array Layout** to `Row-Major` option. The **Array Layout** parameter enables the model for row-major code generation. Alternatively, in the MATLAB Command Window, enter:

```
% For model 'RowLUTColToRowPreconfigured'
set_param('RowLUTColToRowPreconfigured', 'ArrayLayout','Row-major');
% For model 'RowLUTColToRow'
set_param('RowLUTColToRow', 'ArrayLayout','Row-major');
```

3. In the block dialog boxes, examine the permuted 3-D table.





4. Change your current folder in MATLAB to a writable folder. On the **C Code** tab, click **Build** to generate C code. In the generated code, observe the table data with row-major array layout

```
21 /* Block parameters (default storage) */
22 P rtP = {
23   /* Variable: Tbl_1
24    * Referenced by: '<Root>/2-D Lookup Table'
25    */
26   { 1.0, 11.0, 21.0, 31.0, 41.0, 51.0, 2.0, 12.0, 22.0, 32.0, 42.0, 52.0, 3.0,
27     13.0, 23.0, 33.0, 43.0, 53.0, 4.0, 14.0, 24.0, 34.0, 44.0, 54.0, 5.0, 15.0,
28     25.0, 35.0, 45.0, 55.0, 6.0, 16.0, 26.0, 36.0, 46.0, 56.0, 7.0, 17.0, 27.0,
29     37.0, 47.0, 57.0, 8.0, 18.0, 28.0, 38.0, 48.0, 58.0, 9.0, 19.0, 29.0, 39.0,
30     49.0, 59.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0 }
31 };
```

In the generated code, the `memcpy` function replaces the `for` loops. Using `memcpy` reduces the amount of memory for storing data. This optimization improves execution speed.

```
83 /* Model step function */
84 void rtwdemo_row_lutcol2row_workflow_rowrow_step(void)
85 {
86   uint32_T rtb_Prelookup_Index;
87   real_T rtb_Prelookup_Fraction;
88   uint32_T rtb_Prelookup1_Index;
89   real_T rtb_Prelookup1_Fraction;
90   uint32_T rtb_Prelookup2;
91   real_T frac[2];
92   uint32_T bpIndex[2];
93   real_T frac_0[2];
94   uint32_T bpIndex_0[3];
95   real_T frac_1[3];
96   uint32_T bpIndex_1[3];
97
98   /* LookupNDDirect: '<Root>/Direct LUT'
99    *
100   * About '<Root>/Direct LUT':
101   *  4-dimensional Direct Look-Up returning a 2-D Matrix,
102   *  which is contiguous for row-major array
103   */
104   memcpy(&rtY.Out2[0], &rtCP_DirectLUT_table[0], 60U * sizeof(real_T));
105
106   /* PreLookup: '<Root>/PreLookup' incorporates:
107    *  Constant: '<Root>/Constant1'
108    */
109   rtb_Prelookup_Index = plook_evenca(1.0, rtCP_Prelookup_BreakpointsData[0],
110     rtCP_Prelookup_BreakpointsData[1] - rtCP_Prelookup_BreakpointsData[0], 9U,
111     &rtb_Prelookup_Fraction);
```

Observe the algorithms optimized for row-major data.

```
17 real_T intrp2d_la(const uint32_T bpIndex[], const real_T frac[], const real_T
18                   table[], const uint32_T stride, const uint32_T maxIndex[])
19 {
20   real_T y;
21   real_T yR_1d;
22   uint32_T offset_1d;
23
24   /* Row-major Interpolation 2-D
25      Interpolation method: 'Linear point-slope'
26      Use last breakpoint for index at or above upper limit: 'on'
27      Overflow mode: 'wrapping'
28    */
29   offset_1d = bpIndex[1U] * stride + bpIndex[0U];
30   if (bpIndex[0U] == maxIndex[0U]) {
31     y = table[offset_1d];
32   } else {
33     y = (table[offset_1d + 1U] - table[offset_1d]) * frac[0U] + table[offset_1d];
34   }
35
36   if (bpIndex[1U] == maxIndex[1U]) {
37   } else {
38     offset_1d += stride;
39     if (bpIndex[0U] == maxIndex[0U]) {
40       yR_1d = table[offset_1d];
```

```
close_system('RowLUTColToRow',0);
close_system('RowLUTColToRowPreconfigured',0);
```

See Also

Related Examples

- “Code Generation of Matrices and Arrays” on page 52-63
- “Interpolation Algorithm for Row-Major Array Layout” on page 46-10

Interpolation Algorithm for Row-Major Array Layout

This example illustrates the interpolation algorithm in 2-D and 3-D Lookup Table that is optimized for row-major array layout. The interpolation algorithm that is optimized for column-major array layout is also presented as a reference. The code generated by using row-major interpolation algorithm performs with the best speed and memory usage when operating on table data with row-major array layout. The code generated by using column-major interpolation algorithm performs best with column-major array layout.

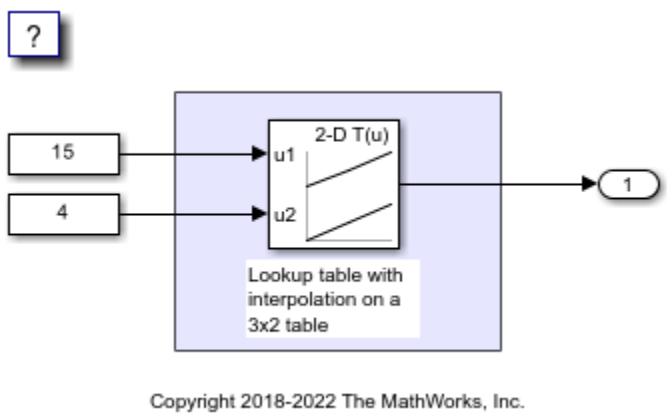
In this example, you:

- Interpolate on a 2-D Lookup Table with column-major and row-major algorithm.
- Generate code with a row-major algorithm and an array layout.
- Identify the array layout and select the optimized algorithm.
- Interpolate on a 3-D Lookup Table with column-major and row-major algorithm.

Simulate with 2-D Row-Major Algorithm

1. Open the example model RowLUT2D.

```
model = 'RowLUT2D';
open_system(model);
```



2. By default, Simulink® configures a model with column-major algorithm and column-major array layout. The model RowLUT2D is configured to use column-major algorithm. Simulate the model. To observe the output, open the Data Inspector from the **Simulation** tab. The output value is 4.

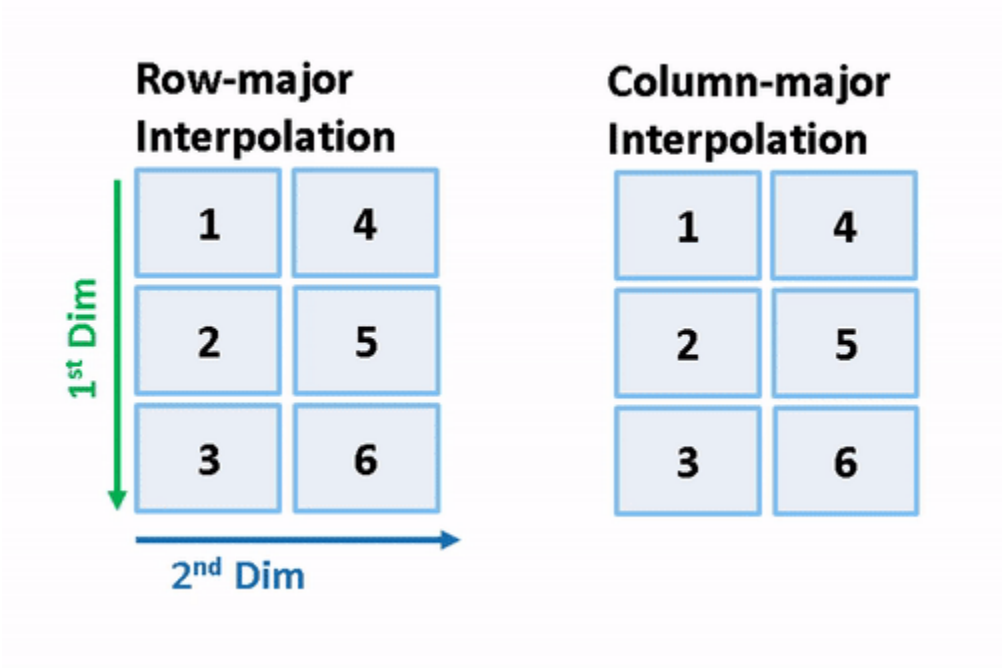
3. To enable row-major algorithms, open the Configuration Parameters dialog box. On the **Math and Data Types** pane, enable the configuration parameter “Use algorithms optimized for row-major array layout”. Alternatively, in the MATLAB® Command Window, enter:

```
set_param(model,'UseRowMajorAlgorithm','on');
```

4. Simulate the model and observe those results in the Data Inspector. The output value is 4.

The column-major and row-major algorithms differ only in the interpolation order. In some cases, due to different operation order on the same data set, you might experience minor numeric differences in

the outputs of column-major and row-major algorithms. For the 2-D Lookup Table used in the example model, the interpolation algorithm is illustrated here.



Generate Code by Using Row-Major Algorithm and Array Layout

The 2-D table data used in model `RowLUT2D` is:

```
Table_3by2.Value
```

```
ans =

     1     4
     2     5
     3     6
```

1. Open the Configuration Parameters dialog box. In addition to enabling the **Use algorithms optimized for row-major array layout** configuration parameter, on the **Code Generation > Interface** pane, set the configuration parameter Array layout to `Row-Major` option. This configuration parameter enables the model for row-major code generation. Alternatively, in the MATLAB Command Window, enter:

```
set_param(model, 'ArrayLayout','Row-major');
```

2. Change your current folder in MATLAB to a writable folder. On the **C Code** tab, click **Build** to generate C code.

3. In the generated code, observe the table data with row-major array layout. For comparison, here is the table data in the generated code for column-major array layout.

2-D Table with row-major array layout

```
/* Block parameters (default storage) */
P_rtwdemo_row_lut2d_T rtwdemo_row_lut2d_P = {
  /* Variable: Table_3by2
   * Referenced by: '<Root>/2-D Lookup Table'
   */
  { 1.0, 4.0, 2.0, 5.0, 3.0, 6.0 }
};
```

2-D Table with column-major array layout

```
/* Block parameters (default storage) */
P_rtwdemo_row_lut2d_T rtwdemo_row_lut2d_P = {
  /* Variable: Table_3by2
   * Referenced by: '<Root>/2-D Lookup Table'
   */
  { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 }
};
```

In the generated code, the table data is in row-major order and the interpolation algorithm is optimized for row-major array layout. The row-major algorithm operates on table data that is contiguous in memory. This leads to faster cache access, making these algorithms cache-friendly.

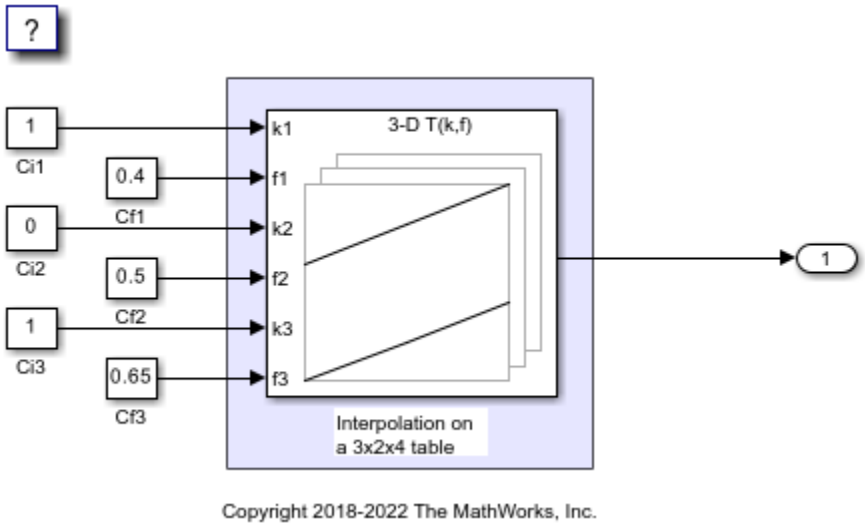
This table summarizes the relationship between array layout and cache-friendly algorithms. It is recommended to use the algorithm that is optimized for the specified array layout to achieve good performance. For example, use row-major interpolation algorithm when the array layout is set to `Row-Major` for code generation.

Array Layout	Algorithm	Cache-Friendly Algorithm
Column-major	Column-major	✓ (Recommended)
Row-major	Row-major	✓ (Recommended)
Row-major	Column-major	⚠ (Not recommended)
Column-major	Row-major	⚠ (Not recommended)

Interpolation on a 3-D Table

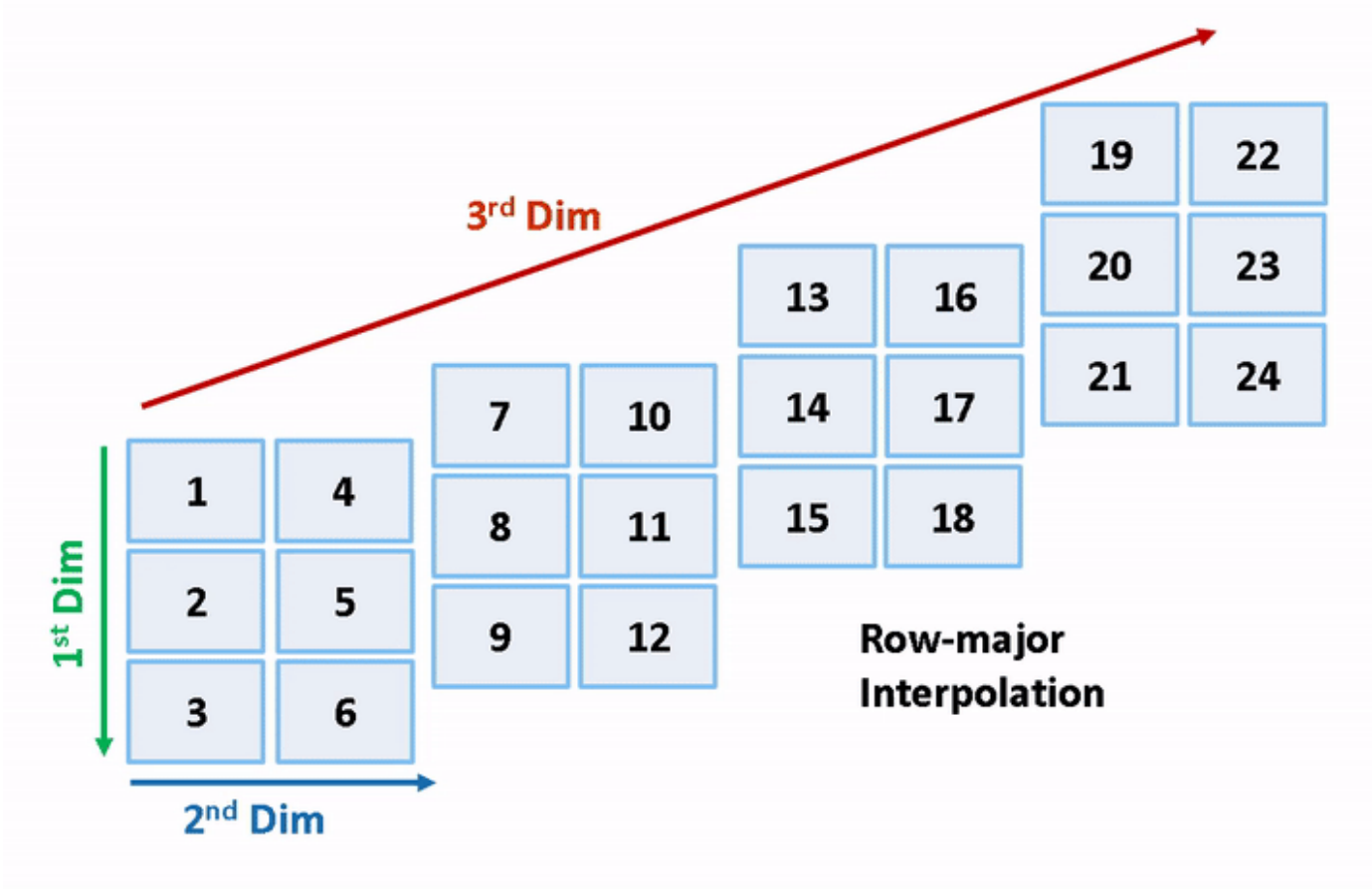
1. Open the example model `RowInterpolationAlgorithm`.

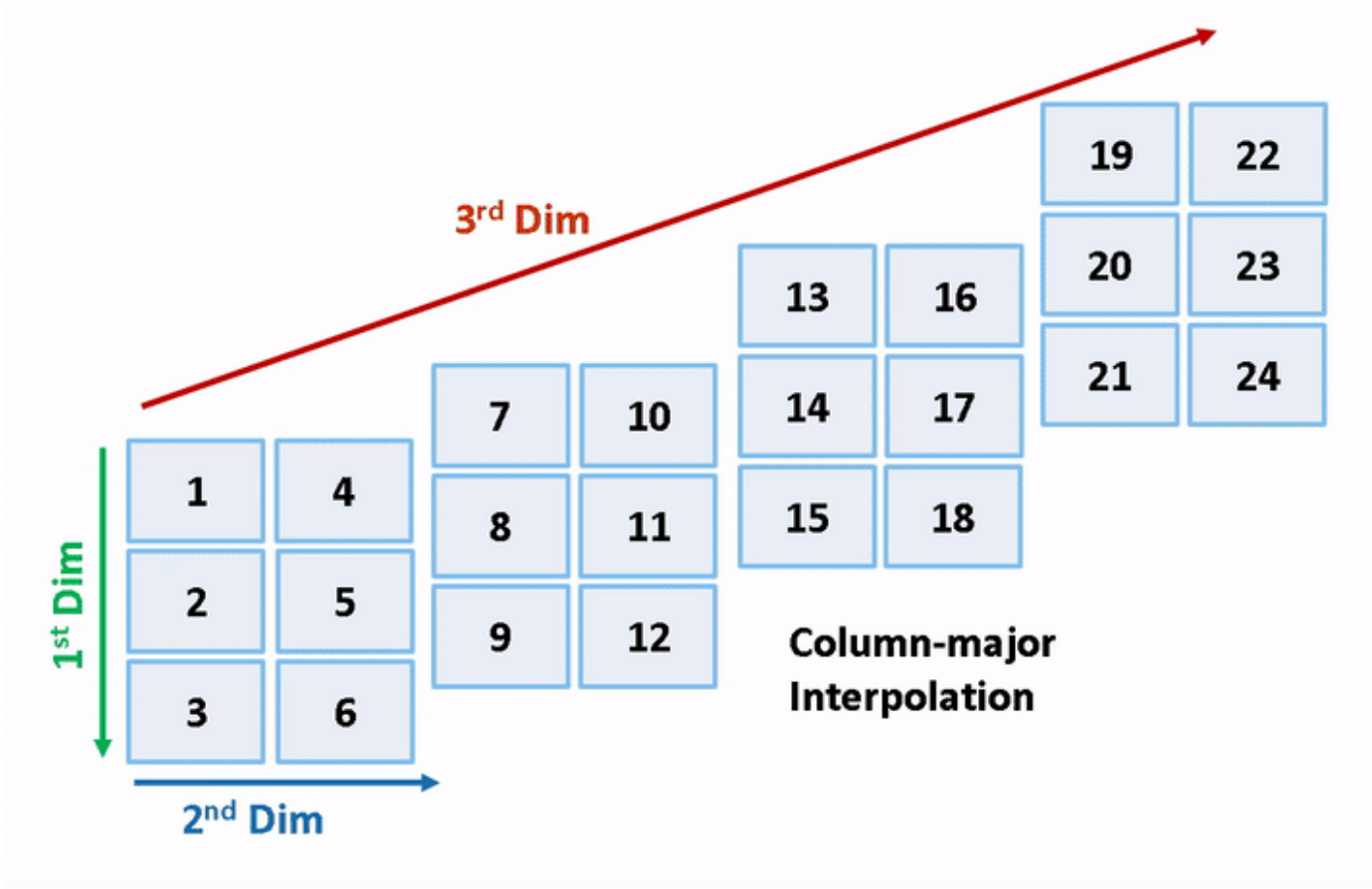
```
open_system('RowInterpolationAlgorithm');
```



2. Generate code for the model by using the preceding procedure. Simulate and generate code from the model by repeating the steps performed on `RowLUT2D` model.

The row-major and column-major interpolations on the 3-D table used in the example model are illustrated here.


3rd Dim
1st Dim
2nd Dim
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Row-major
Interpolation




```
close_system('RowLUT2D',0);
close_system('RowInterpolationAlgorithm', 0);
```

See Also

Related Examples

- “Code Generation of Matrices and Arrays” on page 52-63
- “Interpolation with Subtable Selection Algorithm for Row-Major Array Layout” on page 46-16

Interpolation with Subtable Selection Algorithm for Row-Major Array Layout

This example illustrates the algorithm for interpolating a subtable that the interpolation block selects from a higher dimension table. The interpolation algorithm with subtable selection is optimized for row-major array layout. As a reference, see the interpolation algorithm with subtable selection that is optimized for column-major array layout. The code generated by using row-major interpolation algorithm performs with the best speed and memory usage when operating on table data with row-major array layout. The code generated by using column-major algorithm performs best with column-major array layout.

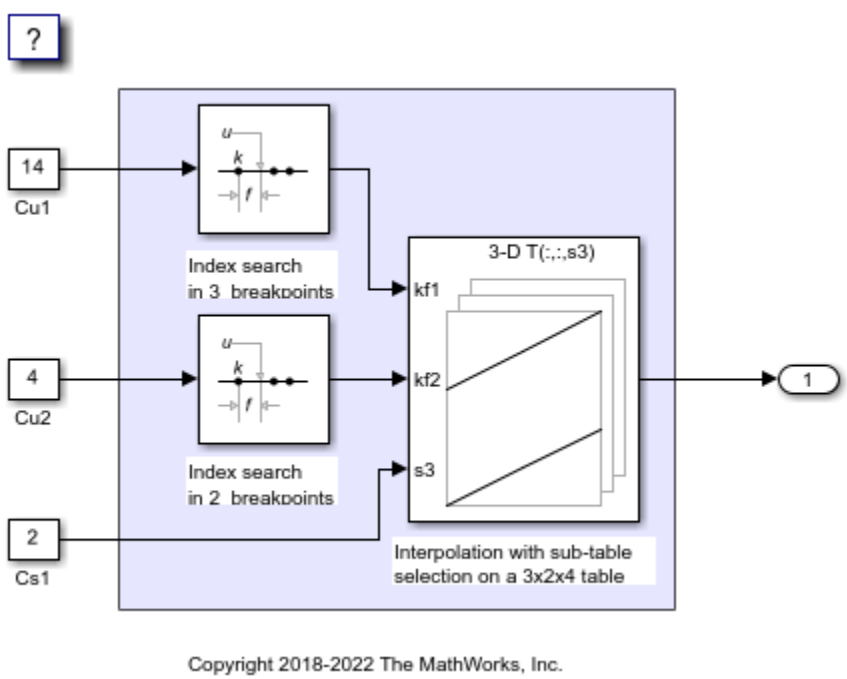
In this example, you:

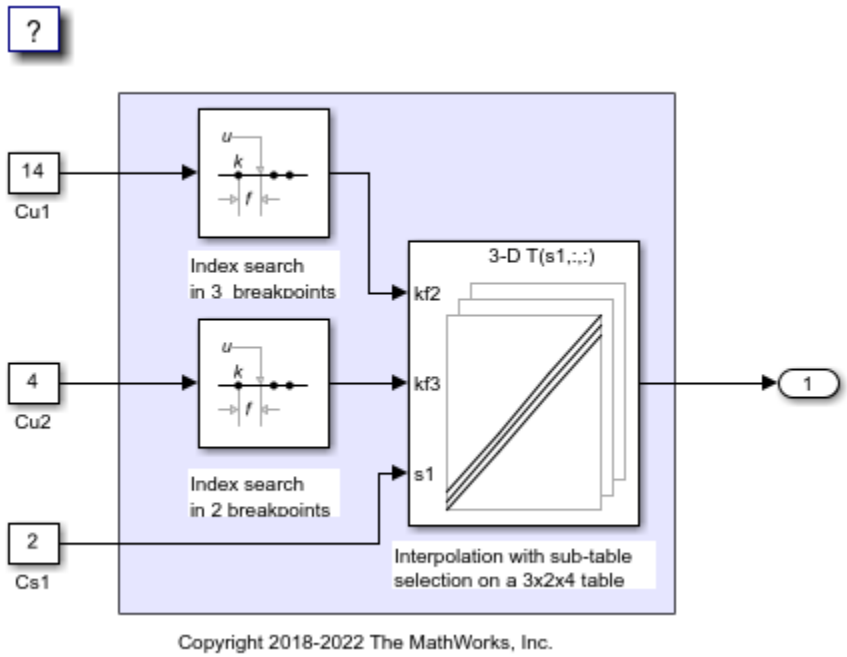
- Interpolate on a selected subtable with column-major and row-major algorithm.
- Preserve block semantics through table permutation.
- Generate code with row-major algorithm and array layout.

Simulate by Using Row-Major Algorithm

1. Open example model `SubtableInterpolationCol` and `SubtableInterpolationRow`.

```
open_system('SubtableInterpolationCol');
open_system('SubtableInterpolationRow');
```





2. By default, Simulink® configures a model with column-major algorithm and column-major array layout. The model `SubtableInterpolationCol` is configured to use a column-major algorithm. Run the model and observe the output stored in workspace variable `yout`.

3. To enable row-major algorithms, open the Configuration Parameters dialog box. On the **Math and Data Types** pane, select the configuration parameter “Use algorithms optimized for row-major array layout” Alternatively, in the MATLAB® Command Window, enter:

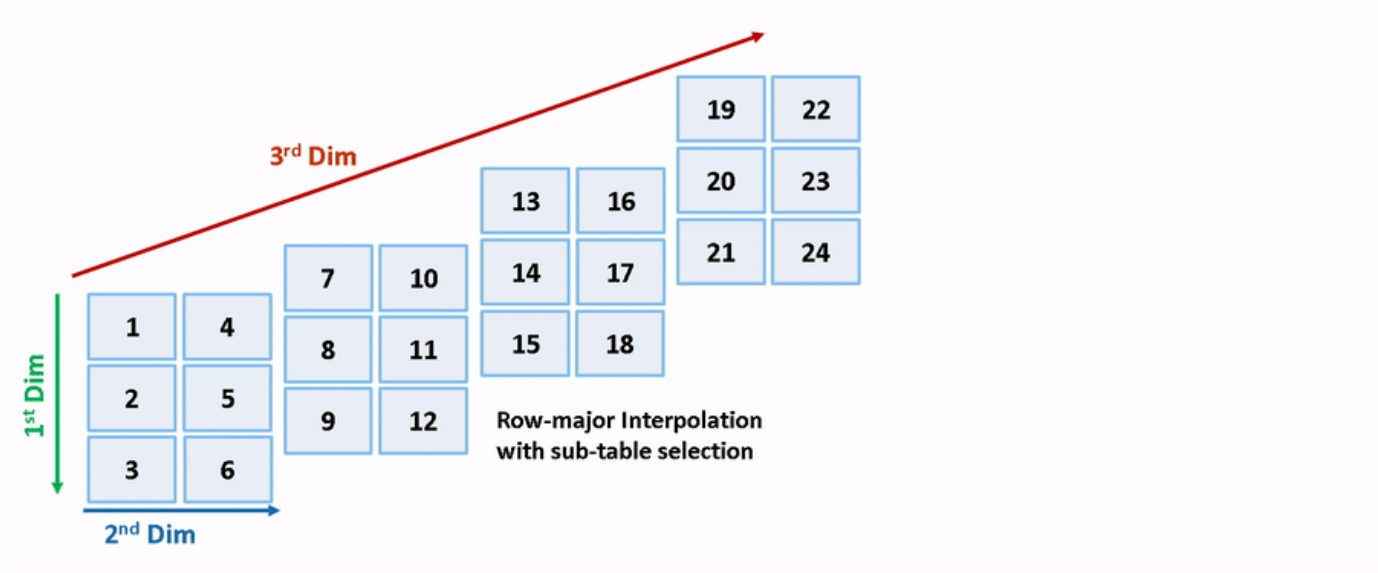
```
set_param('SubtableInterpolationCol','UseRowMajorAlgorithm','on');
```

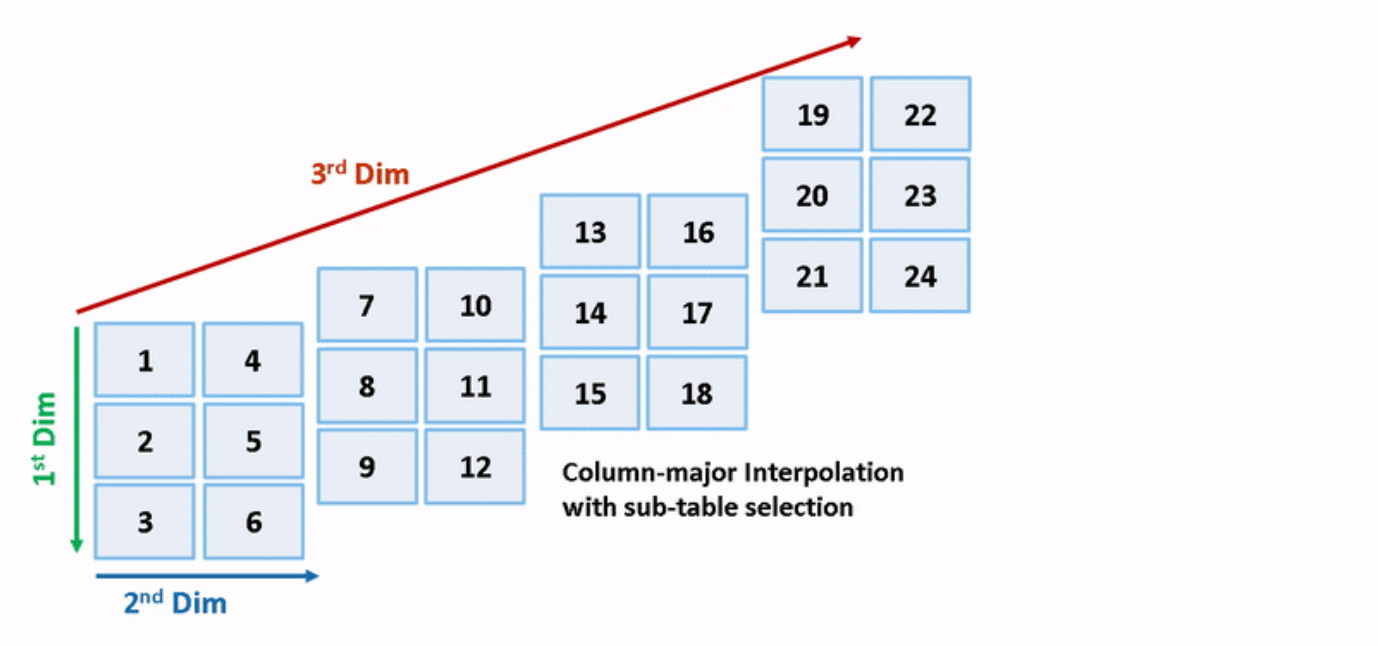
4. Simulate the model and observe the error.

```
The range of values for the first signal in the bus object propagated to the input port kf2
of block 'rtwdemo_col_interpselsubtable/Interp' does not match the size of the corresponding
table dimension of 2. Specify a different table dimension or modify the range for the first
signal in the bus object propagated to the input port kf2.
```

Component: Simulink | Category: Block error

The column-major and row-major algorithms differ in terms of subtable selection and interpolation order. The subtable selection is performed inside the original table. No extra memory is allocated for the subtable. The selected subtable is contiguous in memory. The interpolation order is cache-friendly for column-major algorithms with column-major array layouts and row-major algorithms with row-major array layouts. This illustration compares the row-major and column-major interpolations with subtable selection.





Due to a change in semantics, column-major and row-major interpolations are performed on different subtables or data sets. This interpolation leads to different numeric outputs or an error.

Preserve Semantics Through Table Permutation

With subtable selection, the model semantics change when switching from a column-major algorithm to a row-major algorithm. To preserve the semantics or ensure that the same subtable is selected for interpolation, you need to permute the table data. Otherwise, Simulink might report error if it encounters inconsistent breakpoint and table data between prelookup and interpolation blocks.

1. The block `SubtableInterpolationCol/Interp` has 3-D table data given as **T3d = reshape([1:24], 3,2,4)** and one selection port with input **2** (0-based index). The selected subtable is

T3d(:,:,3) (1-based index in MATLAB) for the column-major algorithm. To preserve the semantics for the row-major algorithm on the same model, that is, select the same subtable with same index and selection port inputs, permute the table as **T3d_p = permute(T3d, [3,1,2])**. The selected subtable is **T3d_p(3,:,:)** (1-based index) for the row-major algorithm.

```
T3d_str = get_param('SubtableInterpolationCol/Interp','Table');
set_param('SubtableInterpolationCol/Interp','Table', ...
    ['permute(',T3d_str,',[3,1,2])']);
```

2. When you import table data from a file, you must permute the table data in the file before importing it. This permutation keeps the table tunable throughout the simulation and code generation workflow.

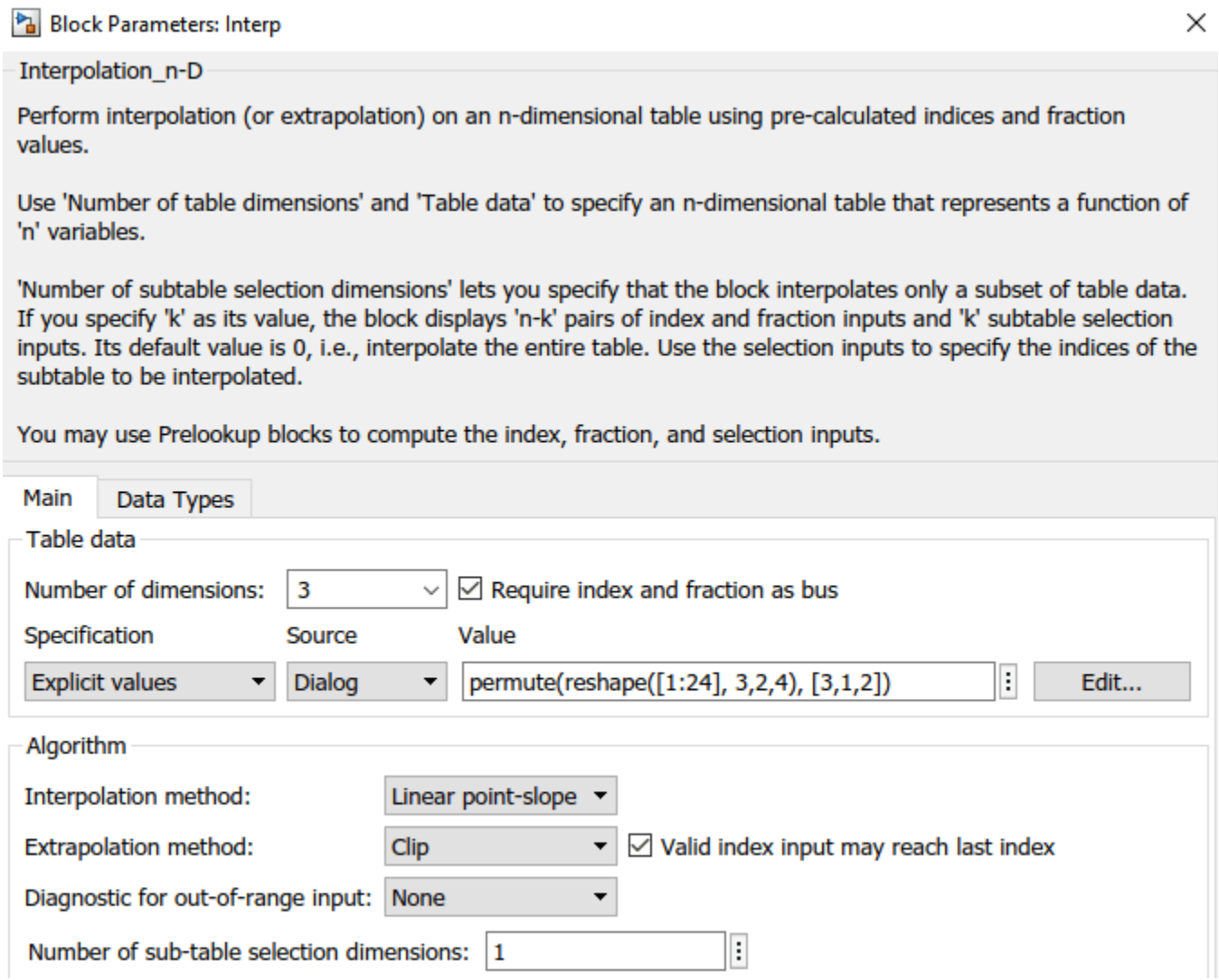
Generate Code by Using Row-Major Algorithm and Array Layout

After permuting the table data, model `SubtableInterpolationCol` is configured for row-major simulation. The model is equivalent to the preconfigured model `SubtableInterpolationRow` that uses a row-major algorithm.

1. To set up the model for row-major code generation, open the Configuration Parameters dialog box. In addition to selecting the **Use algorithms optimized for row-major array layout** configuration parameter, on the **Code Generation > Interface** pane, set the configuration parameter Array layout to `Row-Major` option. This configuration parameter enables the model for row-major code generation. Alternatively, in the MATLAB Command Window, enter:

```
set_param('SubtableInterpolationCol', 'ArrayLayout','Row-major');
```

2. In the dialog box, examine the permuted 3-D table data and the selected 2-D subtable.



3. Change your current folder in MATLAB to a writable folder. On the **C Code** tab, click **Build** to generate C code. In the generated code, observe the 2-D interpolation algorithm optimized for row-major data.

Action	Information	Type of Integration Supported
Reuse shared utility code	You can reuse data and functions across software components by specifying a shared utilities folder for your models. In the Configuration Parameters dialog box, set **Shared code placement** to `Shared location`. If you want software components to share a utilities folder, generate code from a common working folder or specify the same code generation folder through the file generation control parameter, `CodeGenFolder`. For more information, see "Manage Build Process Folders" on page 52-30. Identically-named utility files generated in different releases are functionally equivalent even if file style or comments differ. For cross-release code integration, you can use the `ExistingSharedCode` parameter to specify the reuse of shared utility code from an existing folder. For more information, see: • "Workflow to Reuse Shared Utility Code" on page 52-69 • Existing shared code	Same release, cross-release
Avoid generated header file updates	In general, when you build a model, existing header files (and source files) in the `_sharedutils` or `_shared` folder are not regenerated. However, there are instances when the code generator overwrites existing header files in the folder. The table in "Code Generator Header Files" on page 52-46, which describes header file generation dependencies, provides information about how you can avoid overwriting some header files, for example, `rtwtypes.h` and `multiword_types.h`. For more information, see: • "Build Process Folders" on page 52-31 • "Control Placement of rtwtypes.h for Shared Utility Code" on page 8-54	N/A

Action	Information	Type of Integration Supported
Use cross-release SIL or PIL block	You can use the cross-release SIL or PIL block to integrate generated code from previous releases (R2010a and later) with generated code from the current release. For more information, see “Cross-Release Code Integration” on page 52-71.	Same release, cross-release
Use CRL and `ExistingSharedCode`	You can use the code replacement library (CRL) and `ExistingSharedCode` parameter approaches either separately or jointly: • The CRL approach supports customization of code generation and eliminates the generation of some shared utility files. You must manage the use of utility files that are not replaced by CRL. For more information, see “Develop a Code Replacement Library” on page 72-20. • The **`ExistingSharedCode`** approach, which is based on previously generated utility files, also supports customization and suppresses regeneration of utility files. If customization is not required, the `ExistingSharedCode` approach is simpler to set up and manage. For more information, see “Cross-Release Shared Utility Code Reuse” on page 52-69.	Same release, cross-release

See Also

More About

- “Cross-Release Code Integration” on page 52-71

Integrate Generated Code by Using Cross-Release Workflow

This example shows how to reuse previously generated code by creating a cross-release software-in-the-loop (SIL) block and incorporating the block within an integration model. For more information about the workflow, see “Cross-Release Code Integration” on page 52-71.

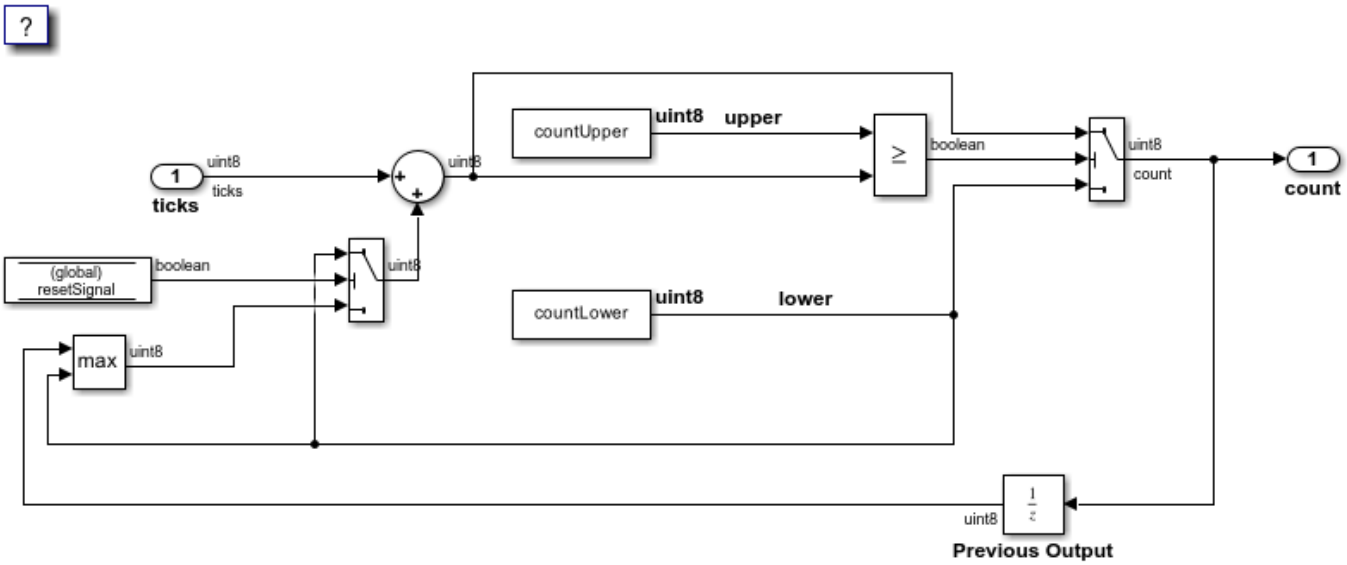
Generate Code from a Model

In the cross-release code integration workflow, you create a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block from code that you previously generated.

This example uses generated code from the current release.

```
model = 'rtwdemo_crossrelease_counter';
close_system(model,0)
load_system(model)
set_param(model, 'SimulationCommand', 'update');
open_system(model)
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_crossrelease_counter
### Successful completion of code generation for: rtwdemo_crossrelease_counter

Build Summary

Top model targets built:

Model                          Action           Rebuild Reason
==============================================================================================
rtwdemo_crossrelease_counter  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.498s
```



Copyright 2017-2020 The MathWorks, Inc.

Identify the build folder that contains the generated code.

```
buildFolder = RTW.getBuildDir(model).BuildDirectory;
```

Identify the folder that contains code for shared utilities associated with the generated code.

```
previousSharedCodeFolder = RTW.getBuildDir(model).SharedUtilsTgtDir;
```

To reuse generated code from a previous release:

1. On the **Simulation** tab, in the **File** section, select **Save > Previous Version**.
2. In the **Save as type field**, specify the release and model type.
3. Click **Save**.
4. Use the previous release to open the saved model, and then generate code.
5. Set the value of `buildFolder` to the location of the code generated in the previous release.
6. Set the value of `previousSharedCodeFolder` to the location of the shared utilities generated in the previous release.

Manage Shared Code

Add the generated shared source code files to a repository folder that is used by the integration model.

```
sharedCodeRepo = 'SharedCodeRepo';
mkdir(sharedCodeRepo);
sharedCodeUpdate(previousSharedCodeFolder, sharedCodeRepo, 'Interactive', false);
```

```
The following files will be copied from slprj\ert\_sharedutils to SharedCodeRepo\R2023a:

    rtwtypes.h

Files copied from slprj\ert\_sharedutils to SharedCodeRepo\R2023a.
```

Import Code

Load the integration model.

```
integrationModel = 'rtwdemo_crossrelease_integration';
close_system(integrationModel, 0);
load_system(integrationModel);
```

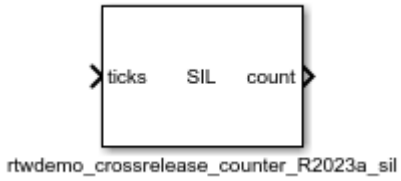
Modify the Simulink configuration set so that it references the existing shared code library.

```
cs = getActiveConfigSet(integrationModel);
set_param(cs, 'ExistingSharedCode', fullfile(pwd, sharedCodeRepo));
```

Create the cross-release SIL block.

```
blockHandle = crossReleaseImport(buildFolder, cs, 'SimulationMode', 'SIL');
```

```
### Starting import process for component: rtwdemo_crossrelease_counter_R2023a
### Starting build process for SIL block: rtwdemo_crossrelease_counter_R2023a
```

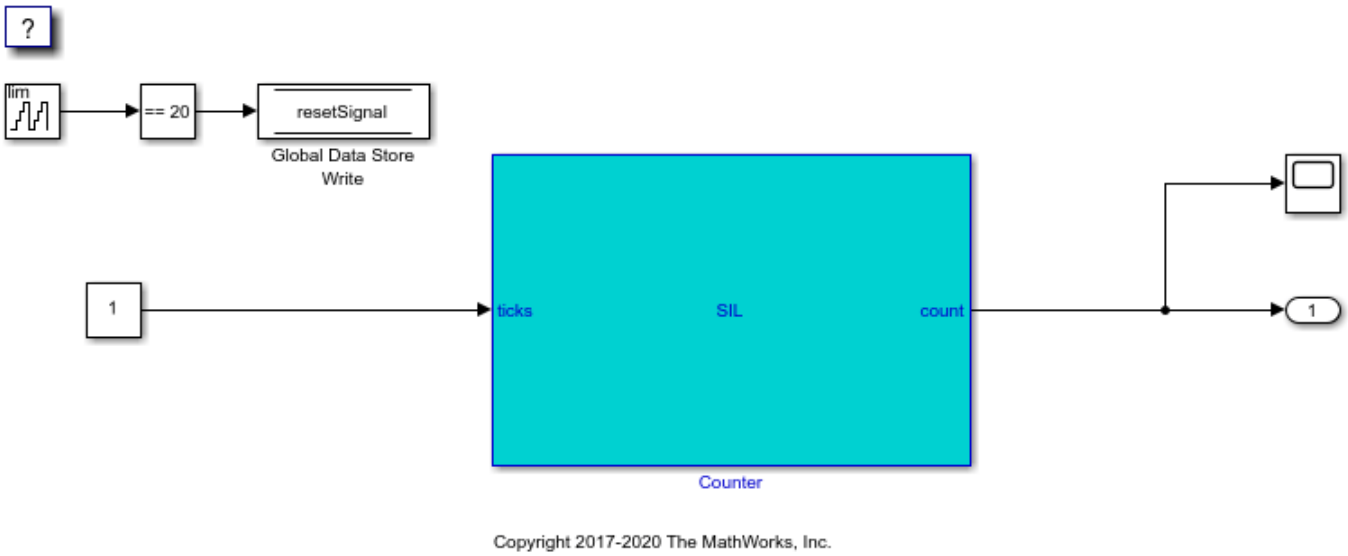


Incorporate Cross-Release SIL Block into Integration Model

To replace a block in the integration model with the cross-release block, use `pil_block_replace`. This function preserves the block size, line connections, and priority.

```
srcBlock = getfullname(blockHandle);
dstBlock = [integrationModel, '/', 'Counter'];
pil_block_replace(srcBlock, dstBlock)
open_system(integrationModel)
```

```
Successfully swapped the following blocks:

untitled/rtwdemo_crossrelease_counter_R2023a_sil
rtwdemo_crossrelease_integration/Counter
```

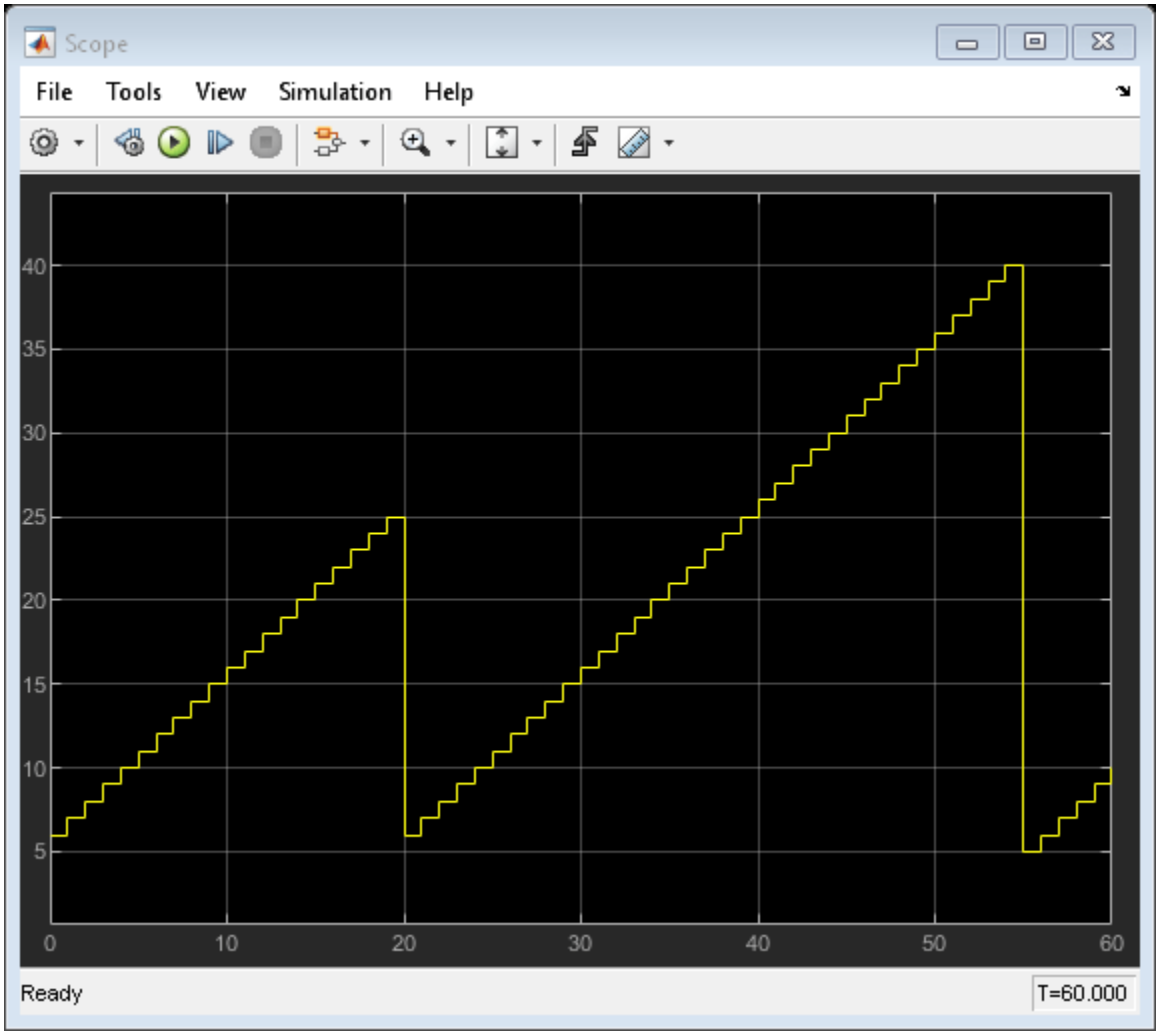


Simulate Integration Model

Run a simulation of the integration model.

```
sim(integrationModel)
```

```
### Preparing to start SIL block simulation: <a href="matlab: targets_hyperlink_manager('run',1)
### Starting SIL simulation for component: rtwdemo_crossrelease_counter_R2023a_sil
### Application stopped
### Stopping SIL simulation for component: rtwdemo_crossrelease_counter_R2023a_sil
```

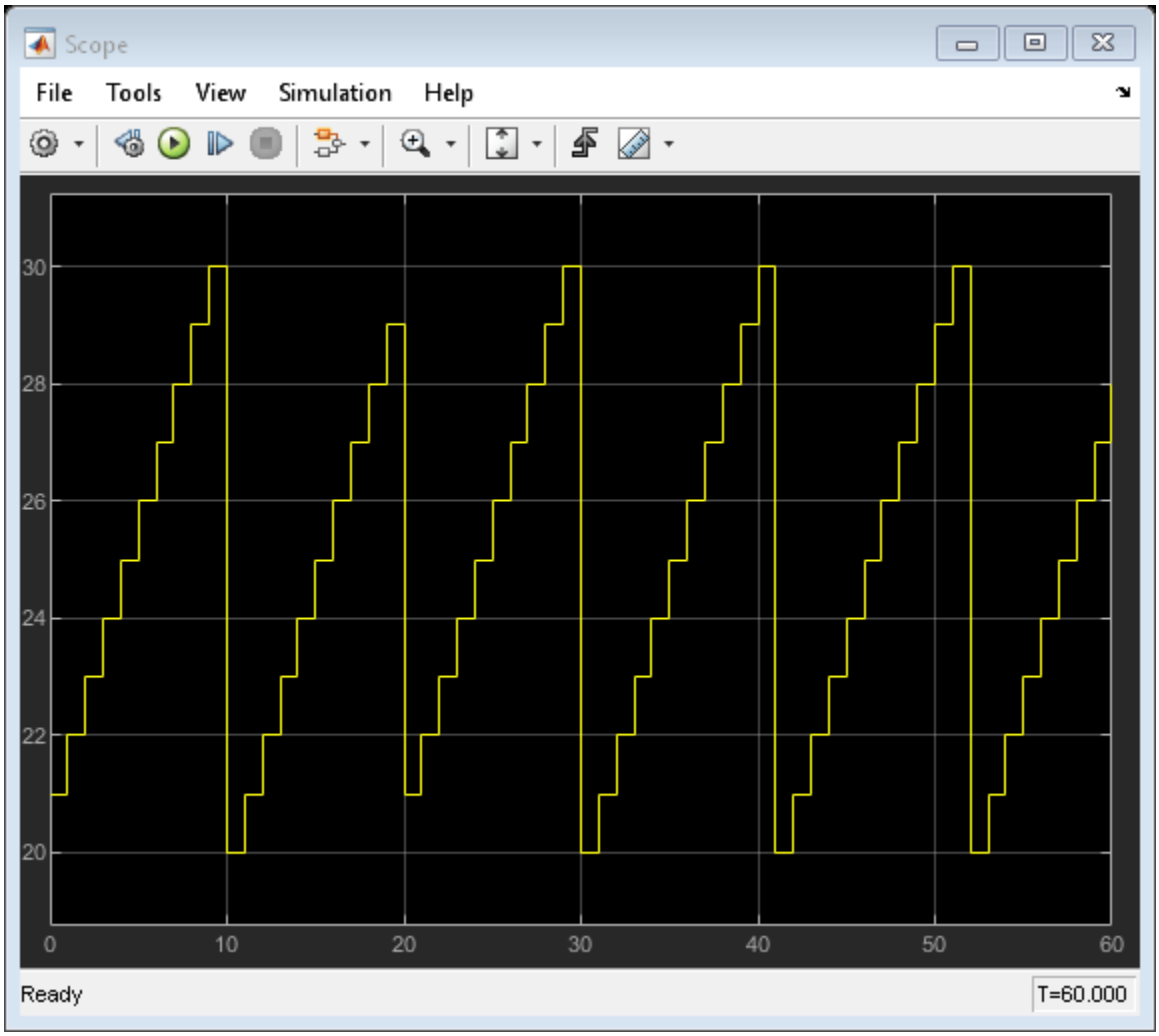


Tune Parameters

The source model for the cross-release SIL block references two tunable parameters, which are controlled by Simulink.Parameter objects in the base workspace. Use these parameters to modify the behavior of the SIL simulation.

```
countUpper.Value = 30;
countLower.Value = 20;
yout_retuned = sim(integrationModel, 'ReturnWorkspaceOutputs', 'on');
```

```
### Preparing to start SIL block simulation: <a href="matlab: targets_hyperlink_manager('run',1)
### Starting SIL simulation for component: rtwdemo_crossrelease_counter_R2023a_sil
### Application stopped
### Stopping SIL simulation for component: rtwdemo_crossrelease_counter_R2023a_sil
```



Configure Storage Class for Signals in Integration Model

Configure:

- Signal names in the integration model to match the names used in the imported code.
- Complementary storage classes.

In this case, the `ticks` input and `count` output are implemented through the `ImportedExtern` storage class in the imported code. If the storage class for the signals connected to the input and output ports of the cross-release block in the integration model is `ExportedGlobal`, the integration model must provide the definitions for the variables.

If the signal names do not match, the integration model generates additional code to copy data between the signal implemented by the integration model and the signal implemented by the imported code.

```
hLines = get_param(dstBlock, 'LineHandles');

% Use the code mappings API to configure the storage class on the signals.
cm = coder.mapping.api.get(integrationModel);
```

```
ticksPortHandle = get(hLines.Inport(1), 'SrcPortHandle');
set(ticksPortHandle, 'Name', 'ticks');
addSignal(cm, ticksPortHandle);
setSignal(cm, ticksPortHandle, 'StorageClass', 'ExportedGlobal');

countPortHandle = get(hLines.Outport(1), 'SrcPortHandle');
set(countPortHandle, 'Name', 'count');
addSignal(cm, countPortHandle);
setSignal(cm, countPortHandle, 'StorageClass', 'ExportedGlobal');
```

Configure Storage Class for Parameters and Data Store Memory in Integration Model

The parameters and data stores are implemented through the `ImportedExtern` storage class in the imported code. If you configure the parameters and data stores to use the `ExportedGlobal` storage class, the integration model must provide the definitions for the variables.

```
resetSignal.CoderInfo.StorageClass = 'ExportedGlobal';
countLower.CoderInfo.StorageClass = 'ExportedGlobal';
countUpper.CoderInfo.StorageClass = 'ExportedGlobal';
```

Generate Code from Integration Model

Once the required shared code is in the shared code repository, remove the previously generated shared utilities folder.

```
if isfolder(RTW.getBuildDir(integrationModel).SharedUtilsTgtDir)
    rmdir(RTW.getBuildDir(integrationModel).SharedUtilsTgtDir, 's');
end
```

Delete the scope and connecting line that do not affect code generation.

```
scopeBlock = [integrationModel, '/', 'Scope'];
hScopeLines = get_param(scopeBlock, 'LineHandles');
hScopeLine = hScopeLines.Inport(1);
assert(strcmp(get(hScopeLine, 'SegmentType'), 'branch'));
delete_line(hScopeLine);
delete_block(scopeBlock);
```

Generate code.

```
slbuild(integrationModel);
```

```
### Starting build procedure for: rtwdemo_crossrelease_integration
### Successful completion of code generation for: rtwdemo_crossrelease_integration

Build Summary

Top model targets built:

Model                              Action           Rebuild Reason
========================================================================================
rtwdemo_crossrelease_integration  Code generated.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.64s
```

Examine Call to Imported Code

To examine the cross-release block code within the integration model code, use the `rtwtrace` utility.

```
rtwtrace(dstBlock);
```

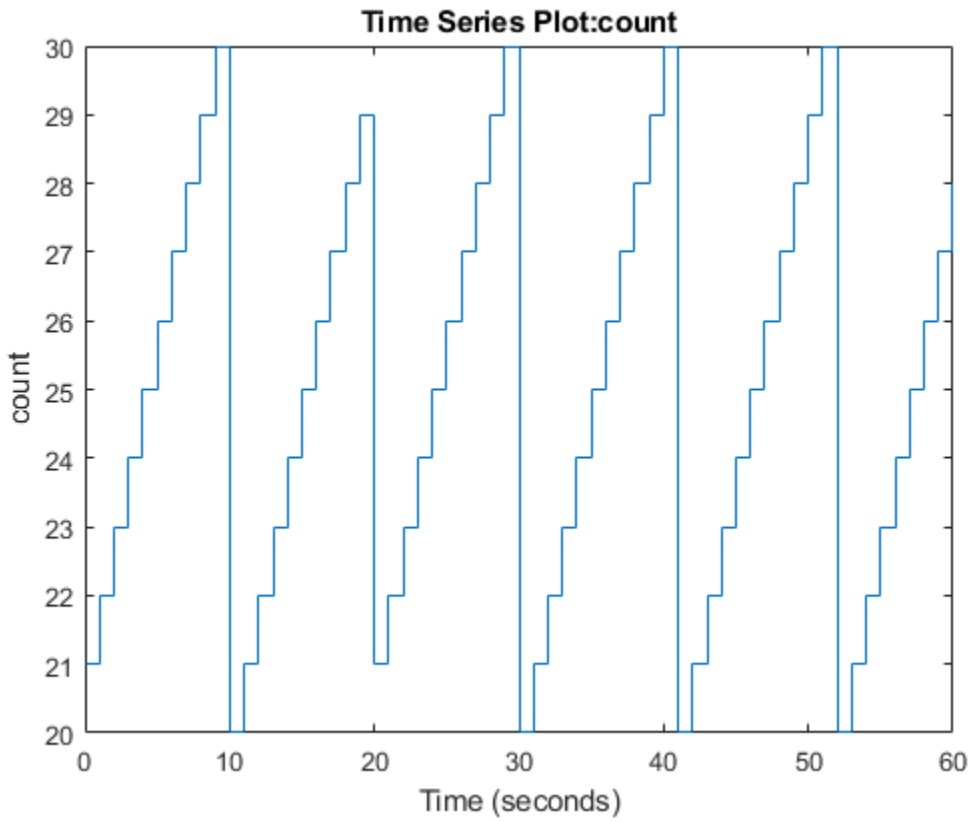
Test Code Generated from Integration Model

Run a top-model software-in-the-loop (SIL) simulation of the integration model. The simulation runs the code generated from the integration model, which calls the imported code.

Log the simulation output in the workspace.

```
set_param(integrationModel, 'SimulationMode', 'software-in-the-loop (sil)');
yout_SIL = sim(integrationModel, 'ReturnWorkspaceOutputs', 'on');
plot(yout_SIL.yout{1}.Values);
```

```
### Starting build procedure for: rtwdemo_crossrelease_integration
### Successful completion of build procedure for: rtwdemo_crossrelease_integration

Build Summary

Top model targets built:

Model                              Action                        Rebuild Reason
========================================================================================
rtwdemo_crossrelease_integration  Code generated and compiled.  S-function rtwdemo_crossrelease_

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.667s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: rtwdemo_crossrelease_integration
### Application stopped
### Stopping SIL simulation for component: rtwdemo_crossrelease_integration
```



Compare Simulation Outputs

Compare the outputs from the simulation where only the imported code ran in SIL mode with the outputs from the simulation where the integration model ran as a top-model in SIL mode.

```
max(abs(yout_SIL.yout{1}.Values.Data - yout_retuned.yout{1}.Values.Data))
```

```
ans =

  uint8

   0
```

Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code

This example shows how you can use a protected model from a previous release (R2018b onwards) to:

- Test numerical equivalence between the protected model and its previously generated code.
- Generate new code in the current release.

The workflow in this example supports only protected models that specify these system target file types:

- ERT
- ERT-based
- AUTOSAR Classic
- GRT (Embedded Coder required)
- GRT-based (Embedded Coder required)

Protected Model

Use the pregenerated protected model `rtwdemo_crossrelease_protected_counter.slxp`, which was created in R2018b.

The protected model allows the user to:

- Open a read-only view of the model.
- Run simulations.
- Use generated code.

The protected model supports the ERT system target file, and the code that it contains is "Readable source code" and supports the "Model reference" code interface.

If you want to re-create the protected model with different settings, you can export the model `rtwdemo_crossrelease_unprotected_counter` from the current release into a format that can be used in MATLAB R2018b. In the current release, run these commands:

```
load_system('rtwdemo_crossrelease_unprotected_counter');
Simulink.exportToVersion('rtwdemo_crossrelease_unprotected_counter', ...
    'rtwdemo_crossrelease_protected_counter.slx', ...
    'R2018b');
close_system('rtwdemo_crossrelease_unprotected_counter', 0);
```

The protected model was created in MATLAB R2018b by running this command:

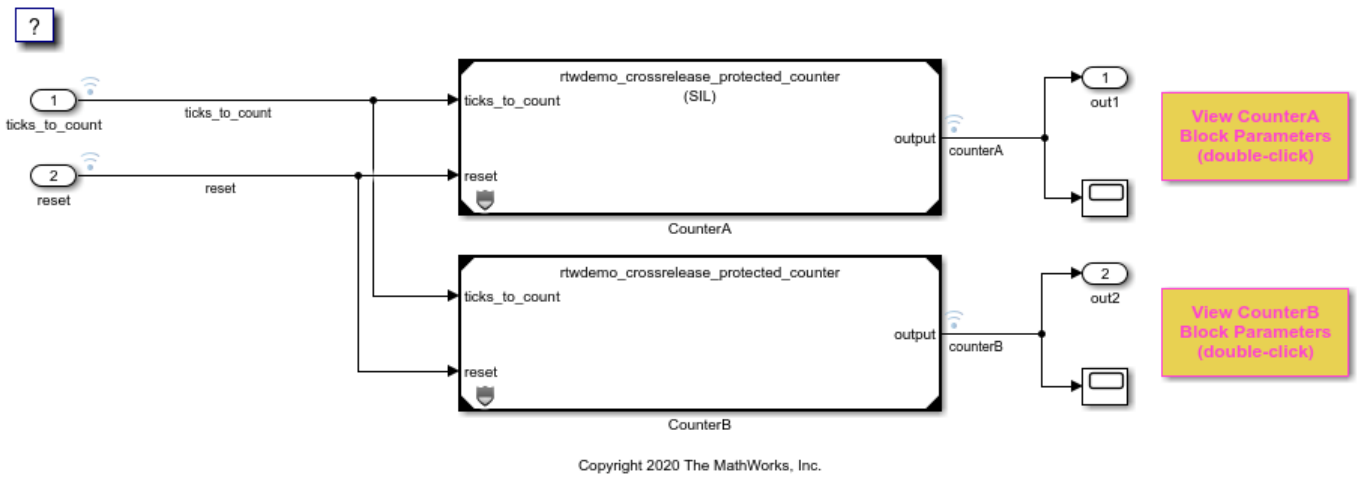
```
Simulink.ModelReference.protect('rtwdemo_crossrelease_protected_counter', ...
    'Mode', 'CodeGeneration', ...
    'OutputFormat', 'MinimalCode', ...
    'ObfuscateCode', false, ...
```

```
    'WebView', true, ...
    'Report', true)
```

Open Top Model

Open `rtwdemo_crossrelease_protected_model`, the top model that references the protected model.

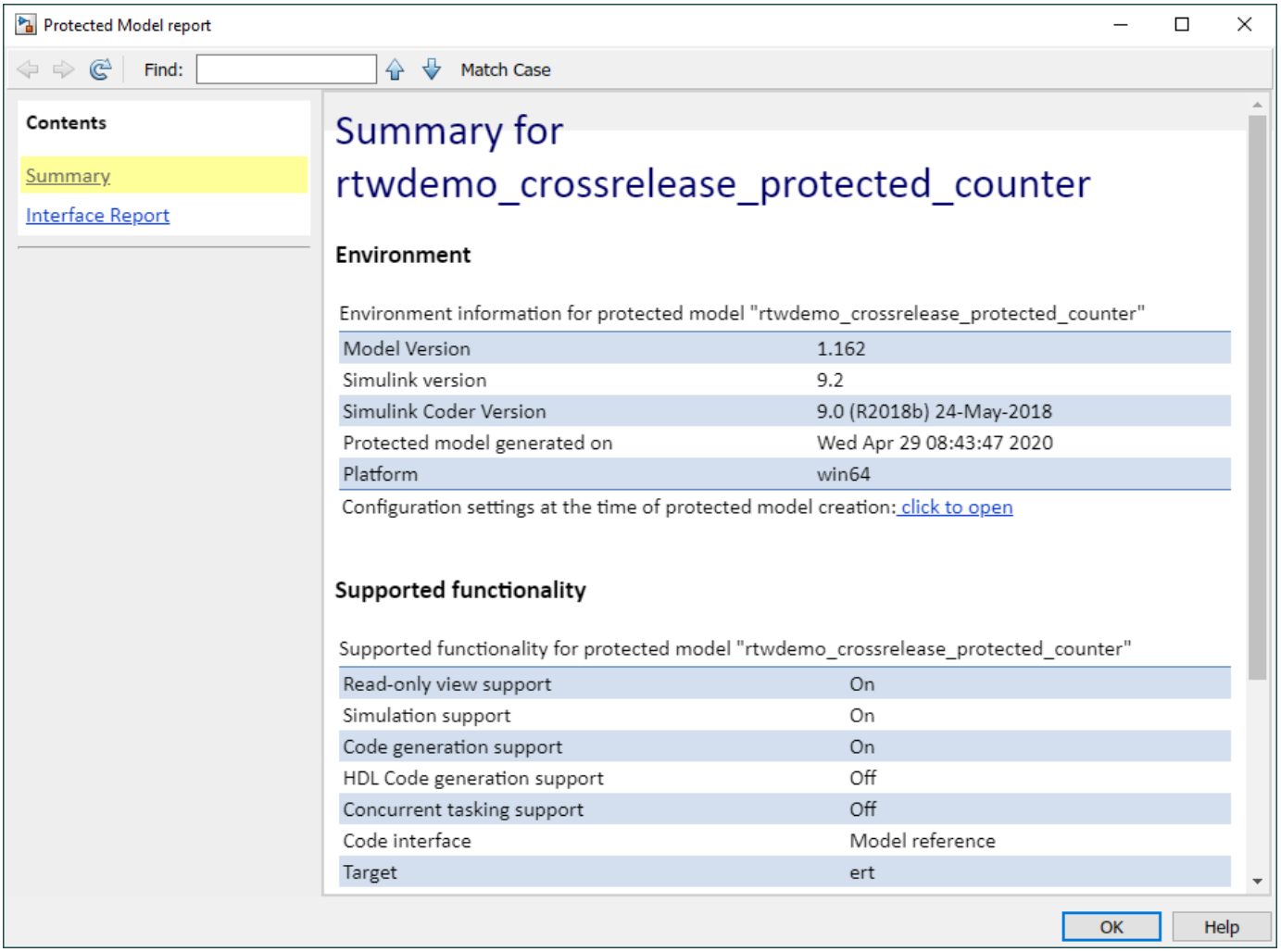
```
model = 'rtwdemo_crossrelease_protected_model';
open_system(model);
```



Open Protected Model Report

To open the protected model report, on one of the Model blocks, right click the protected-model badge icon and then click "Display report". Or, open the report from the command line.

```
protectedModel = 'rtwdemo_crossrelease_protected_counter';
Simulink.ModelReference.ProtectedModel.displayReport(protectedModel);
```

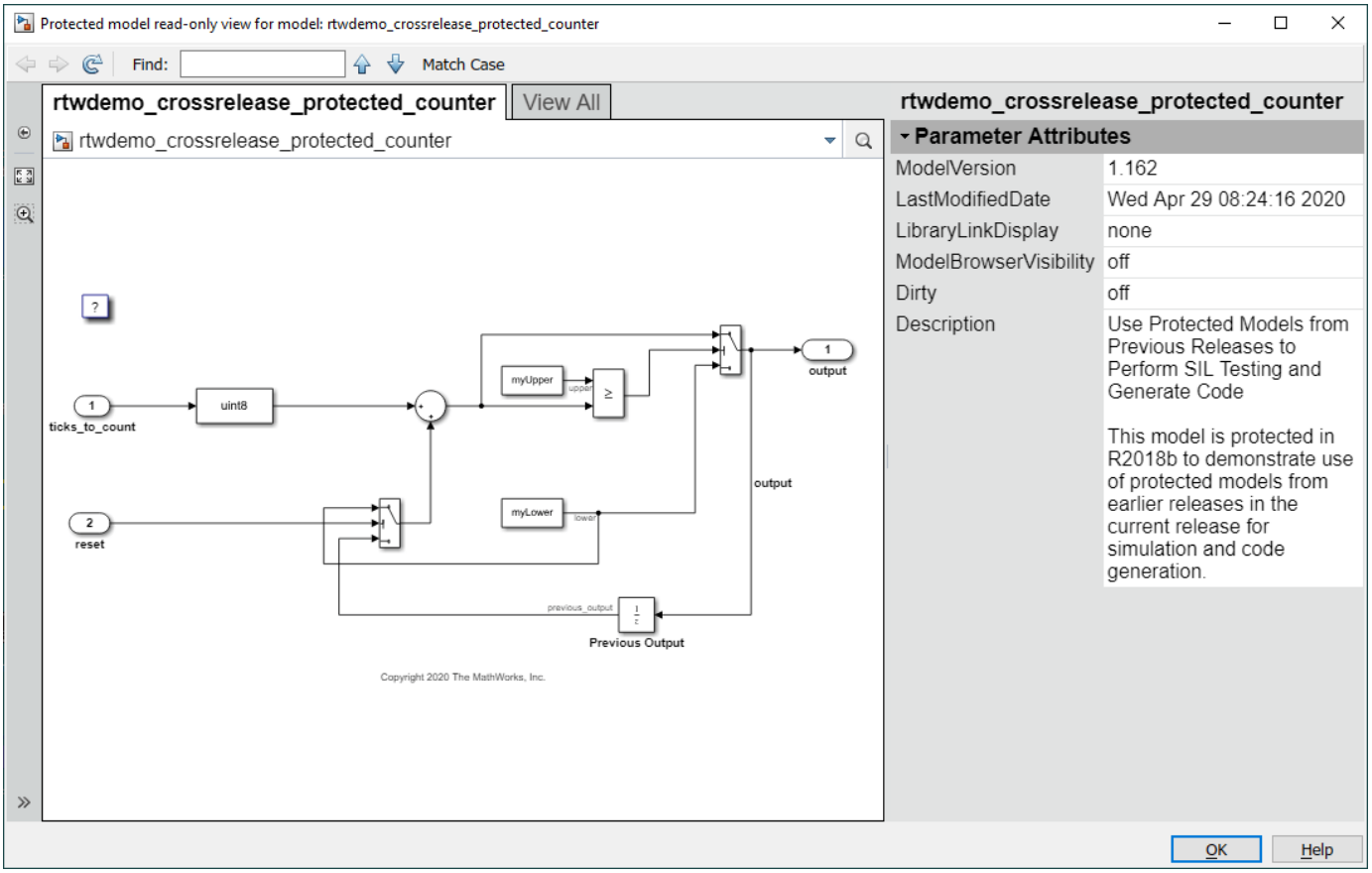


The report shows that the protected model was generated by using Simulink Coder 9.0 (R2018b).

Open Read-Only View

To open a read-only view of the protected model, right click the protected-model badge icon and then click "Show web view". Or, open the read-only view from the command line.

```
Simulink.ModelReference.ProtectedModel.displayWebview(protectedModel);
```



Merge Shared Utility Code

When you integrate code from multiple releases, you can specify the shared utility code that the final build uses.

Specify the shared code folder for the top model.

```
mkdir('sharedCode');
set_param(model, 'ExistingSharedCode', fullfile(pwd, 'sharedCode'));
```

Use `sharedCodeUpdate` to merge protected model shared utility files with the shared code folder content.

```
Simulink.ModelReference.ProtectedModel.CurrentTarget.clear(protectedModel);
sharedCodeUpdate('rtwdemo_crossrelease_protected_counter.slxp', 'sharedCode', 'Interactive', fals


The following files will be copied from rtwdemo_crossrelease_protected_counter.slxp:slprj\ert\_sh

    rtwtypes.h

Files copied from rtwdemo_crossrelease_protected_counter.slxp:slprj\ert\_sharedutils to sharedCo
```

Test the Code in the Protected model

The top model references two instances of the protected model. The **Simulation mode** Model block parameter for "CounterA" and "CounterB" are `Software-in-the-loop (SIL)` and `Accelerator`

respectively. When you run a top-model simulation, "CounterA" runs generated code from the protected model while "CounterB" runs the protected model.

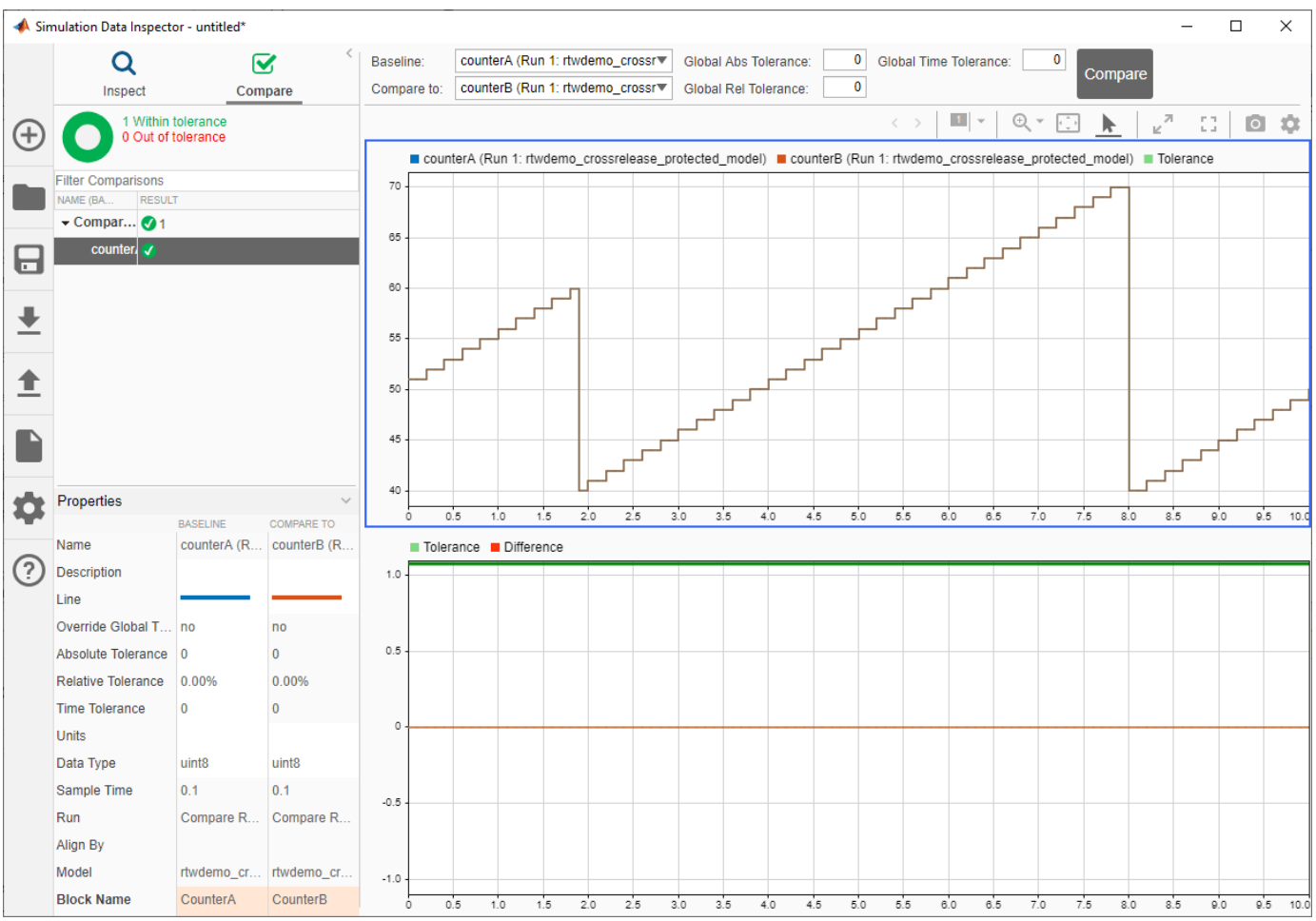
```
simOut = sim(model);
Simulink.sdi.view;
```

```
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Starting SIL simulation for component: rtwdemo_crossrelease_protected_counter
### Application stopped
### Stopping SIL simulation for component: rtwdemo_crossrelease_protected_counter
```

Compare Outputs from Both Model Blocks

To view the simulation output and compare the values of the signals from the Model blocks, use the Simulation Data Inspector.

```
simulationRunObj =  Simulink.sdi.getCurrentSimulationRun(model);
counterASignal = getSignalsByName(simulationRunObj, 'counterA');
counterBSignal = getSignalsByName(simulationRunObj, 'counterB');
Simulink.sdi.compareSignals(counterASignal.ID, counterBSignal.ID);
Simulink.sdi.view(Simulink.sdi.GUITabType.CompareRuns);
```



Generate Code from Top Model

You can generate code from the top model, referencing the code that is contained in the protected model.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_crossrelease_protected_model
### Successful completion of build procedure for: rtwdemo_crossrelease_protected_model

Build Summary

Top model targets built:

Model                                  Action                        Rebuild Reason
==========================================================================================
rtwdemo_crossrelease_protected_model  Code generated and compiled.  Code generation information 

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 38.731s
```

See Also

`sharedCodeUpdate`

More About

- “Reference Protected Models from Third Parties”
- “SIL and PIL Simulations” on page 88-2

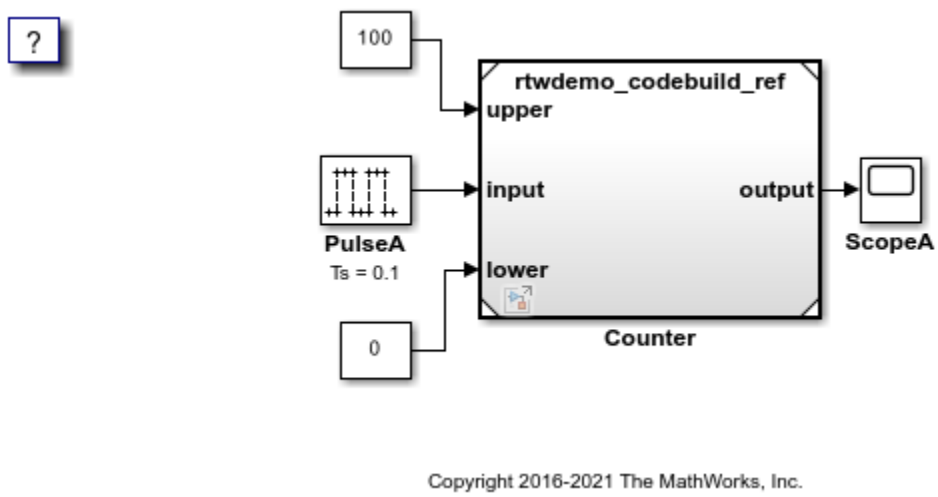
Compile Code in Another Development Environment

This example shows how you can relocate generated code and compile it in another development environment. Use packNGo to create a ZIP file that contains generated source code, build information, and CMake configuration files. In your target development environment, use the codebuild function to build the final library or executable file.

Configure the Model

Open the model.

```
rtwdemo_codebuild
```



Do not automatically compile the code, generate a makefile, or package generated code and artifacts.

```
set_param('rtwdemo_codebuild', 'GenerateMakefile', 'off');
set_param('rtwdemo_codebuild', 'PackageGeneratedCodeAndArtifacts', 'off');
```

Generate Code and CMake Configuration Files

Run code generation command.

```
slbuild('rtwdemo_codebuild');
```

```
### Starting serial model reference code generation build
### Successfully updated the model reference code generation target for: rtwdemo_codebuild_ref
### Starting build procedure for: rtwdemo_codebuild
### Successful completion of code generation for: rtwdemo_codebuild

Build Summary

Code generation targets built:

Model                   Action          Rebuild Reason
================================================================================
rtwdemo_codebuild_ref   Code generated  rtwdemo_codebuild_ref.c does not exist.
```

```
Top model targets built:

Model               Action          Rebuild Reason
========================================================================================
rtwdemo_codebuild  Code generated  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 56.832s
```

Generate the CMakeLists.txt configuration files.

```
buildFolder = RTW.getBuildDir('rtwdemo_codebuild').BuildDirectory;
codebuild(buildFolder, 'BuildMethod', 'cmake');
```

Package Generated Code and CMake Configuration Files

Run packNGo, packaging the files hierarchically.

```
packNGo(buildFolder, 'packType', 'hierarchical', 'nestedZipFiles', false);
```

Switch to Another Development Environment

This is an optional step. For example, if your current development environment is a Linux computer, you can copy the zip file to a Windows computer and then perform the subsequent steps on that computer.

Unzip Source Code and Build Information Files

Unzip the files.

```
rtwdemo_codebuild_files = unzip('rtwdemo_codebuild.zip');
```

Identify folder containing unzipped code for the top component.

```
rtwdemo_codebuild_top_component_folder = fileparts(rtwdemo_codebuild_files{1});
```

Build Executable File Using Default Toolchain

Identify the default toolchain (based on compiler selected by mex -setup).

```
defaultToolchain = coder.make.getDefaultToolchain;
```

Build the executable file.

```
codebuild(rtwdemo_codebuild_top_component_folder, 'BuildMethod', defaultToolchain);
```

Check that executable file is produced.

```
dir(fullfile(rtwdemo_codebuild_top_component_folder, '..', 'rtwdemo_codebuild*'))
```

```
rtwdemo_codebuild.exe
rtwdemo_codebuild_grt_rtw
```

Build Executable File Using Default Template Makefile

Select a template makefile for the current platform.

```
if ispc
    % With ert_vcx64.tmf, you must have Microsoft Visual C++ installed. If
    % it is not installed, specify ert_lcc64.tmf to use lcc-win64 instead.
    templateMakefile = 'ert_vcx64.tmf';
else
    templateMakefile = 'ert_unix.tmf';
end
```

Build the executable file.

```
codebuild(rtwdemo_codebuild_top_component_folder, 'BuildMethod', templateMakefile);
```

Build Shared Library

On Windows, create a definition file that specifies exported symbols.

```
if ispc
    exportsFile = fullfile...
        (rtwdemo_codebuild_top_component_folder, 'rtwdemo_codebuild.def');
    fid = fopen(exportsFile, 'w');
    fwrite(fid, ['EXPORTS' newline]);
    fwrite(fid, ['rtwdemo_codebuild_initialize' newline]);
    fwrite(fid, ['rtwdemo_codebuild_step' newline]);
    fwrite(fid, ['rtwdemo_codebuild_terminate' newline]);
    fclose(fid);
end
```

Build the shared library.

```
codebuild(rtwdemo_codebuild_top_component_folder, 'BuildVariant', 'SHARED_LIBRARY');
```

Build Static Library

```
codebuild(rtwdemo_codebuild_top_component_folder, 'BuildVariant', 'STATIC_LIBRARY');
```

Use CMake Configuration Files

With the CMakeLists.txt files, you can use the third-party tool CMake to generate makefiles or workspaces for a compiler environment of your choice.

```
% Compute the CMake command for the current platform.
if ispc
    cmakeCommand = fullfile(matlabroot, 'bin', computer('arch'), 'cmake', 'bin', 'cmake.exe');
else
    cmakeCommand = fullfile(matlabroot, 'bin', computer('arch'), 'cmake', 'bin', 'cmake');
end
```

Using CMake, build the executable file.

```
cd(rtwdemo_codebuild_top_component_folder)
[status1, cmdout1] = system([cmakeCommand ' .']);
[status2, cmdout2] = system([cmakeCommand ' --build .']);
```

See Also

packNGo | codebuild

More About

- “Relocate or Share Generated Code” on page 59-56
- “Code Compilation Customization”
- “Configure CMake Build Process” on page 59-13
- “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
- “Code Compilation Customization”

Generate Code and Simulate Models in a Project

This example shows how to use the code generation template for a new Project. The code generation project template for Project includes multiple models. The project template also provides utilities (.m scripts) that help you generate controller code and run simulations of the harness model.

Create a Project

Create a new Project from the code generation project template.

To create this project from the Simulink® start page, in the Command Window, type:

```
simulink
```

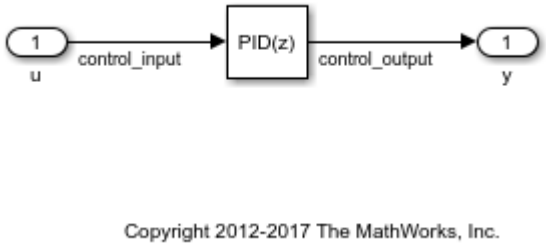
Select the Code Generation template from the start page, and create the `exampleCodeGen` project.

Generate Code

Generate controller code for the `feedback_control.slx` model.

To generate controller code, select the **Project Shortcuts** tab and select the **Generate Controller Code** shortcut.

This shortcut runs the `generate_controller_code.m` script in the `utilities` folder of the project. The script builds the `feedback_control.slx` model in the `controller` folder of the project.

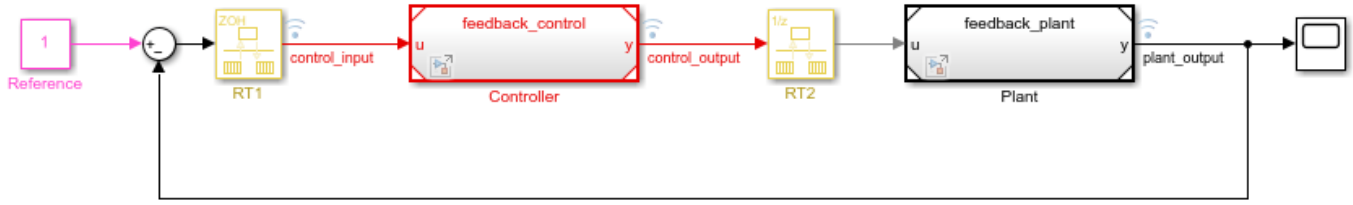


Simulate Models

Simulate the top-level harness `feedback_harness.slx` model.

To open the harness model for simulation, select the **Project Shortcuts** tab and select the **Feedback Harness** shortcut. This shortcut opens the `feedback_harness.slx` model in the `harnesses` folder of the project.

To simulate the model, click **Run**.

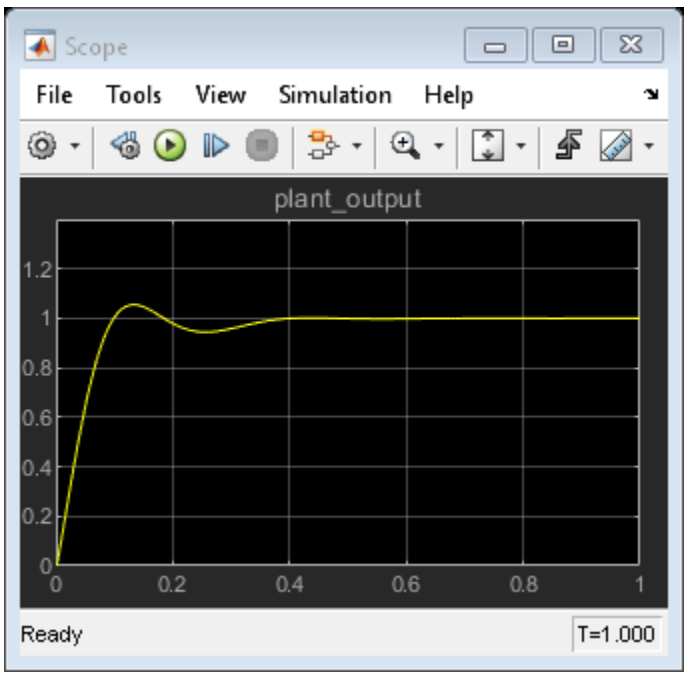


Copyright 2012 The MathWorks, Inc.

Observe Simulation Output

Open the `Scope` block in the model and observe the simulation output.

To open the scope and observe simulation, in the Simulink window, double-click the `Scope` block.

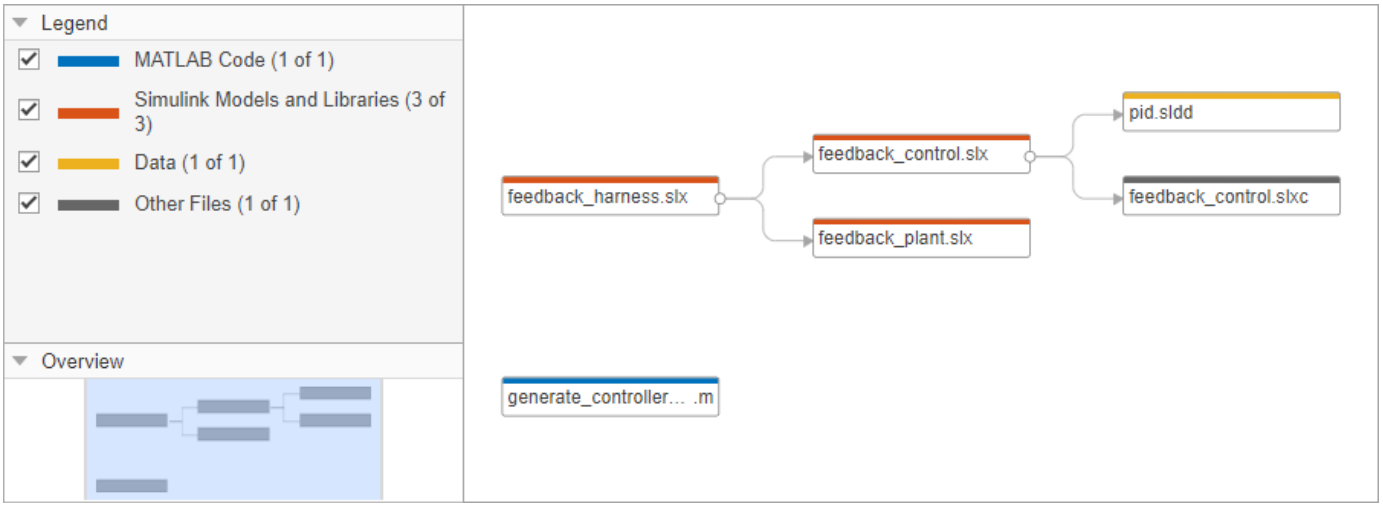


Use the Dependency Analyzer View

To view file dependencies, use the **Dependency Analyzer** view of the project.

From the **Dependency Analyzer** view, you can:

- Observe file dependencies.
- Hover over the dependency arrows to examine the dependency type.
- Double-click on a file or a model to open it.



More Information

- "Work with Project Files"
- `Simulink.createFromTemplate`
- "Automate Startup Tasks"
- "Automate Shutdown Tasks"

Related Examples

- "Create a New Project from a Folder"
- "Create a Project from a Model"

Generate Code and Simulate Models Programmatically with Project

This example shows how to use the Project API to create a new project and automate project tasks for manipulating files. It covers how to programmatically create a blank project, add files and folders, set up the project path, define project shortcuts, generate code, and simulate the harness model.

Project helps you organize large designs by managing and sharing files and settings, finding required files, and interacting with source control.

Manage a Model-Based Design Project

The example models provide a controller that positions a camera fixed to a DC motor that tracks a green ball.

The MBD approach leads to modeling the controller operations:

- Model plant: The plant is a DC motor that rotates to reposition the camera.
- Model controller: The design uses a PID controller with feedback on angular position for positioning the DC motor.
- Model high-level ball tracking scheduler: The scheduler checks for validity of the position of the green ball and the saturation of the reference angle. The scheduler handles the instances of no ball on the screen to track.

The design and Project have the following folders.

Plant -- The plant provides:

- Models with continuous-time dynamic systems that use basic Simulink® blocks.
- Models with linear time invariant (LTI) systems that use the LTI system block.
- Import and export of simulation data to and from the workspace
- Loading of parameters into the base workspace that use data dictionary and model callbacks.

Controller -- The controller provides:

- Models with discrete-time dynamic systems that use basic Simulink blocks.
- Models with discrete-time dynamic systems that use the PID Controller Block.
- Enforcement of signal saturation that use a PWM Conversion subsystem.

Ball Tracking -- The ball tracking provides:

- The basic ball tracking algorithm.
- The terminator and ground blocks as placeholders.
- A hybrid controller that generates motor reference angles using Stateflow®.

Harnesses -- The harnesses provide top-level simulation of the full system.

Create a Blank Project and Add Files

1. Create a blank project. On the **Home** tab, click **New > Project > Blank Project.**

Alternatively, create this project and use currentProject to get a project object to manipulate the project from the Command Window:

```
matlab.project.createProject('modelBasedDesignCodeGenProject');
proj_modelBasedDesignCodeGen = currentProject
```

```
proj_modelBasedDesignCodeGen = 
  Project with properties:

                       Name: "modelBasedDesignCodeGenProject"
    SourceControlIntegration: ""
          RepositoryLocation: ""
       SourceControlMessages: [1×0 string]
                    ReadOnly: 0
                    TopLevel: 1
                Dependencies: [1×1 digraph]
                  Categories: [1×1 matlab.project.Category]
                       Files: [1×0 matlab.project.ProjectFile]
                   Shortcuts: [1×0 matlab.project.Shortcut]
                 ProjectPath: [1×0 matlab.project.PathFolder]
           ProjectReferences: [1×0 matlab.project.ProjectReference]
                StartupFiles: [1×0 string]
               ShutdownFiles: [1×0 string]
         DefinitionFilesType: FixedPathMultiFile
                 Description: ""
                  RootFolder: "C:\Users\ktripp\MATLAB\Projects\modelBasedDesignCodeGenProject1"
         DependencyCacheFile: ""
       SimulinkCodeGenFolder: ""
        ProjectStartupFolder: "C:\Users\ktripp\MATLAB\Projects\modelBasedDesignCodeGenProject1"
         SimulinkCacheFolder: ""
```

2. Copy the example files to folders within the Project.

Alternatively, in the Command Window, type:

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','slproject_codegen_demo'), ...
fullfile(pwd),'f');
```

3. Add existing files to the project:

- Click the **Project Files View** button and select **All files View**.
- Select the folders `ball_tracking`, `controller`, `harnesses`, `plant`, and `utilities`, right-click the icon for one of the folders, and select **Add Folder to Project (Including Child Files)**.

Alternatively, add these files to the project programmatically from the Command Window:

```
addFolderIncludingChildFiles(proj_modelBasedDesignCodeGen,'ball_tracking');
addPath(proj_modelBasedDesignCodeGen,'ball_tracking');
addFolderIncludingChildFiles(proj_modelBasedDesignCodeGen,'controller');
addPath(proj_modelBasedDesignCodeGen,'controller');
addFolderIncludingChildFiles(proj_modelBasedDesignCodeGen,'harnesses');
addPath(proj_modelBasedDesignCodeGen,'harnesses');
addFolderIncludingChildFiles(proj_modelBasedDesignCodeGen,'plant');
addPath(proj_modelBasedDesignCodeGen,'plant');
addFolderIncludingChildFiles(proj_modelBasedDesignCodeGen,'utilities');
addPath(proj_modelBasedDesignCodeGen,'utilities');
```

4. Select the `work` folder, right-click the icon for one of the folders, and select **Project Path > Add to the Project Path (Including Subfolders)**. If you choose to add this folder to the path, generated files (artifacts) from simulation and code generation are available on the project path. If you choose not to include the generated files in the project, the dependency graph does not display these files.

Alternatively, add this folder to the project path from the Command Window:

```
addPath(proj_modelBasedDesignCodeGen,'work');
```

5. To make the project more accessible, define and set properties for project shortcuts. In the **Files** View, select the **Project** tab. Open the `utilities` folder node.

6. Select the files `clean_up_project.m`, `generate_controller_code.m`, and `set_up_project.m`, right-click the icon for one of the files, and select **Create Shortcut**.

7. To set the current working folder, set the simulation cache folder, and run other setup scripts when you open the project:

- Select the `set_up_project.m` script.
- Right-click the file.
- Select **Run at Startup**.

Alternatively, add this setup script to project startup from the Command Window:

```
addStartupFile(proj_modelBasedDesignCodeGen,fullfile(pwd,'/utilities/set_up_project.m'));
```

8. To restore previous settings and run other clean-up scripts when you close the project:

- Select the `clean_up_project.m` script.
- Right-click the file.
- Select **Run at Shutdown**.

Alternatively, add the clean-up script to project shutdown programmatically from the Command Window:

```
addShutdownFile(proj_modelBasedDesignCodeGen,fullfile(pwd,'/utilities/clean_up_project.m'));
```

9. Reopen the project to run the **Startup** shortcut.

Alternatively, reload the project and run the `set_up_project` from the Command Window:

```
reload(proj_modelBasedDesignCodeGen);
run('set_up_project');
```

Generate Code for the Controller from the Dependency Graph

1. On the **Project** tab, click **Dependency Analyzer** to run a dependency analysis on all the files in your project.

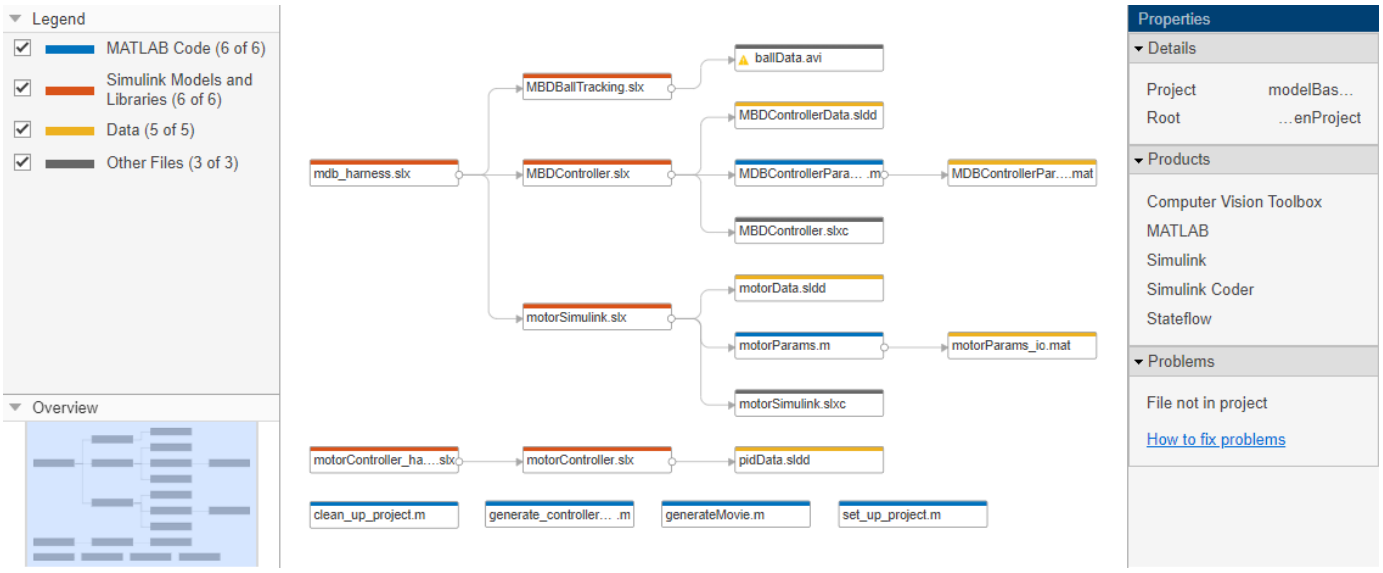
The dependency graph displays your project structure and the files dependencies. It shows how files relate to each other. Each item in the graph represents a file, and each arrow is a dependency.

The contents of the `work` folder do not appear in the dependency graph. This setup simplifies the view to indicate the dependencies in model files and `.m` script file.

a. Examine the **Properties** pane on the right for useful information about your project, such as required products and problem files.

b. Use the **Zoom In** button to zoom into the dependencies map. Use arrow keys or click-drag with the mouse roller wheel to move around in the map.

c. Hover over the dependency arrows to examine the dependency type. For example, the MATLAB® scripts and functions are being run by model callbacks.



2. Select the `MBDController.slx` model on the dependency map. Right-click the model and select **Open**.

3. When the model opens in Simulink, click the **Build** button to generate code for the model. Or, use the `generate_controller_code` shortcut in the project to generate code.

Alternatively, generate code for the model from the Command Window. For the sake of this example, set write permissions to `MBDController.slxc` then generate code for the model:

```
fileattrib('MBDController.slxc', '+w');
evalc('slbuild(''MBDController'')');
```

4. Close the project from the Project window.

Alternatively, close this project and run `clean_up_project` from the Command Window:

```
run('clean_up_project');
close(proj_modelBasedDesignCodeGen);
```

See Also

`currentProject`, `openProject`

More Information

- "Work with Project Files"
- `Simulink.createFromTemplate`

- “Automate Startup Tasks”
- “Automate Shutdown Tasks”

Related Examples

- “Create a New Project from a Folder”
- “Create a Project from a Model”

Source Code Generation in Embedded Coder

- “Generate Code Using Embedded Coder” on page 53-2
- “Generate Code by Using the Quick Start Tool” on page 53-9
- “Manage File Packaging of Generated Code Modules” on page 53-12
- “Tutorial for Generating C Code from a Control Algorithm for an Embedded System” on page 53-17
- “Prepare a Control Algorithm Model for C Code Generation” on page 53-18
- “Configure Data Interface in the Generated Code” on page 53-29
- “Partition Functions in Generated Code” on page 53-39
- “Call External C Code from Model and Generated Code” on page 53-48
- “Build Integrated Code Outside the Simulink Environment” on page 53-53
- “Test Generated Code” on page 53-58
- “Evaluate Performance of the Generated Code” on page 53-64
- “Install and Use Cygwin and Eclipse” on page 53-68

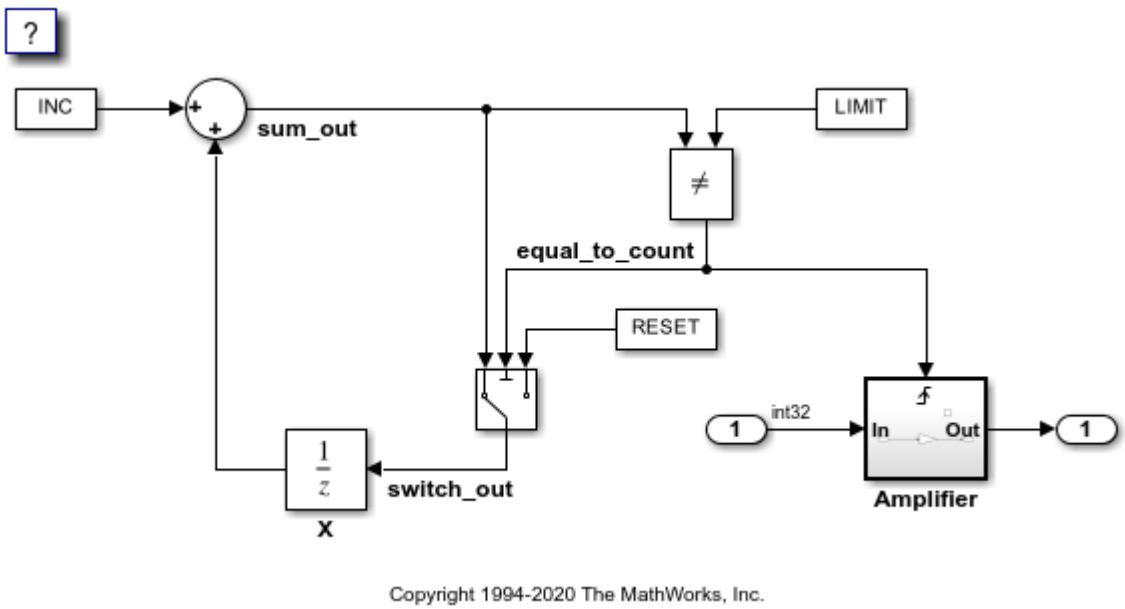
Generate Code Using Embedded Coder

This example shows how to select a system target file for a Simulink® model, configure model parameters, generate C code for embedded systems, and view generated files.

The model represents an 8-bit counter that feeds a triggered subsystem that is parameterized by constant blocks `INC`, `LIMIT`, and `RESET`. `Input` and `Output` represent I/O for the model. The `Amplifier` subsystem amplifies the input signal by gain factor `K`, which updates when signal `equal_to_count` is `true`.

1. Open the model.

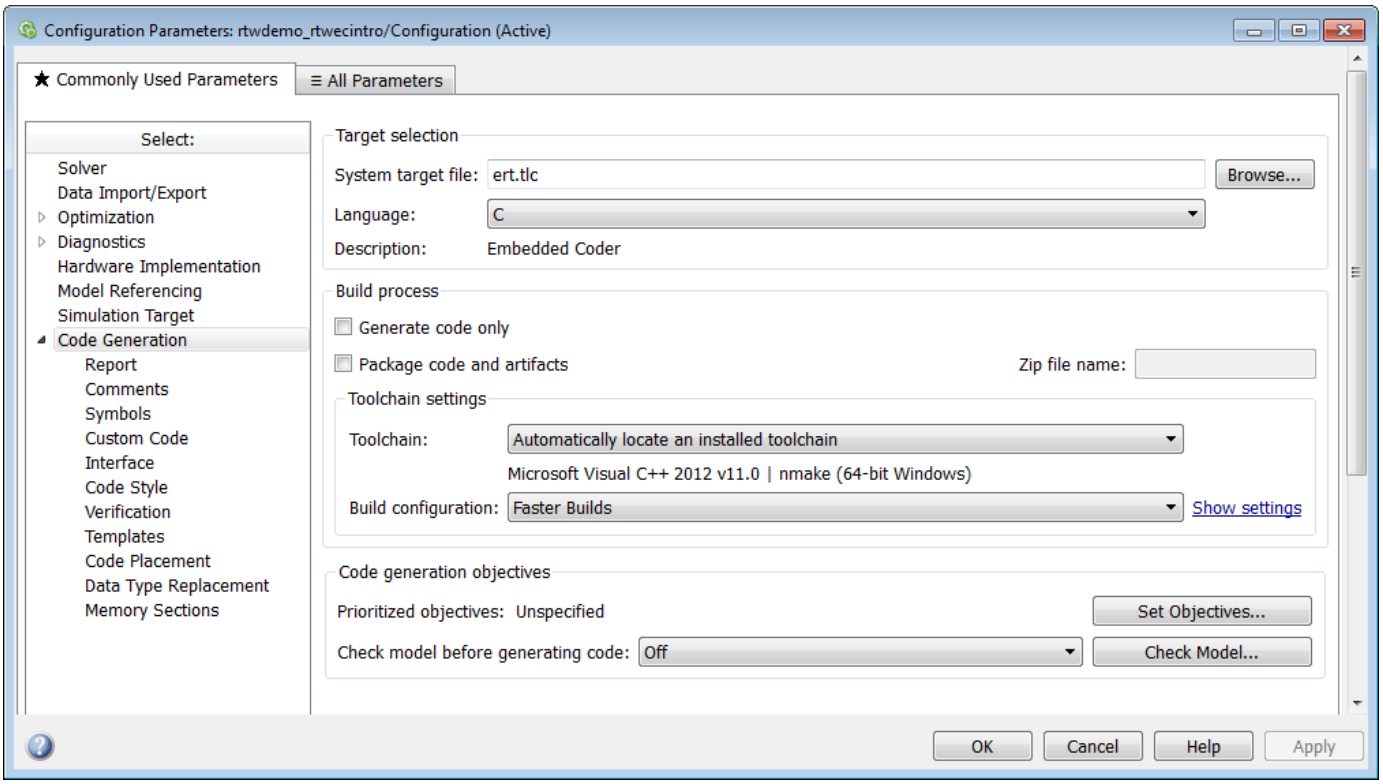
```
model='rtwdemo_rtwecintro';
open_system(model)
```



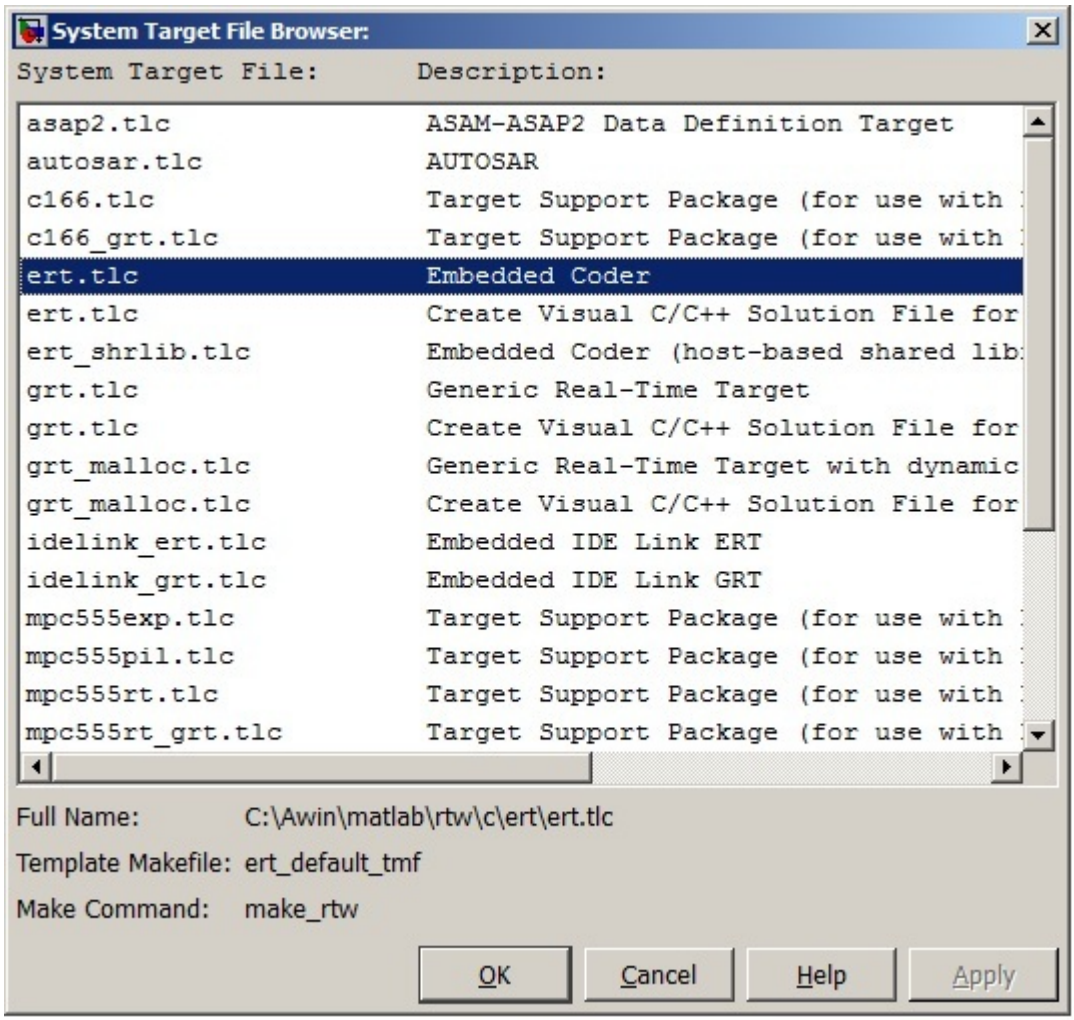
2. Open the Model Configuration Parameters dialog box.

```
cs = getActiveConfigSet(model);
openDialog(cs);
```

3. Select the **Code Generation** tab.



4. To select a system target file, in the **Target Selection** pane, click **Browse** You can generate code for a specific target environment or purpose. Some built-in targeting options are provided by system target files, which control the code generation process for a target environment.

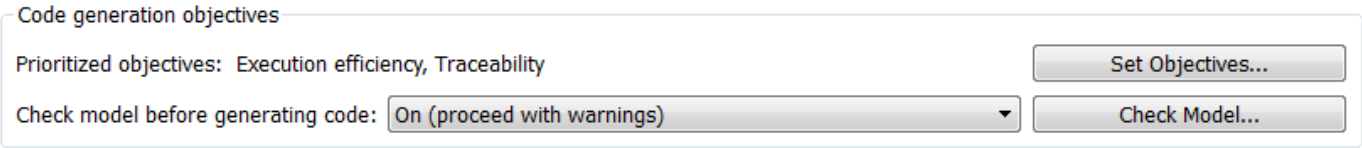


5. Select the **Embedded Real-Time (ERT)** system target file and click **Apply**.

The ERT system target file supports a utility that you can use to specify and prioritize code generation settings based on your application objectives.

6. In the **Code Generation Advisor** pane, click **Set Objectives**.

You can set and prioritize objectives for the generated code. For example, while code traceability might be a very important criterion for your application, you might not want to prioritize it at the cost of code execution efficiency.



7. In the **Set Objectives** pane, select **Execution efficiency** and **Traceability**. Click **OK**.

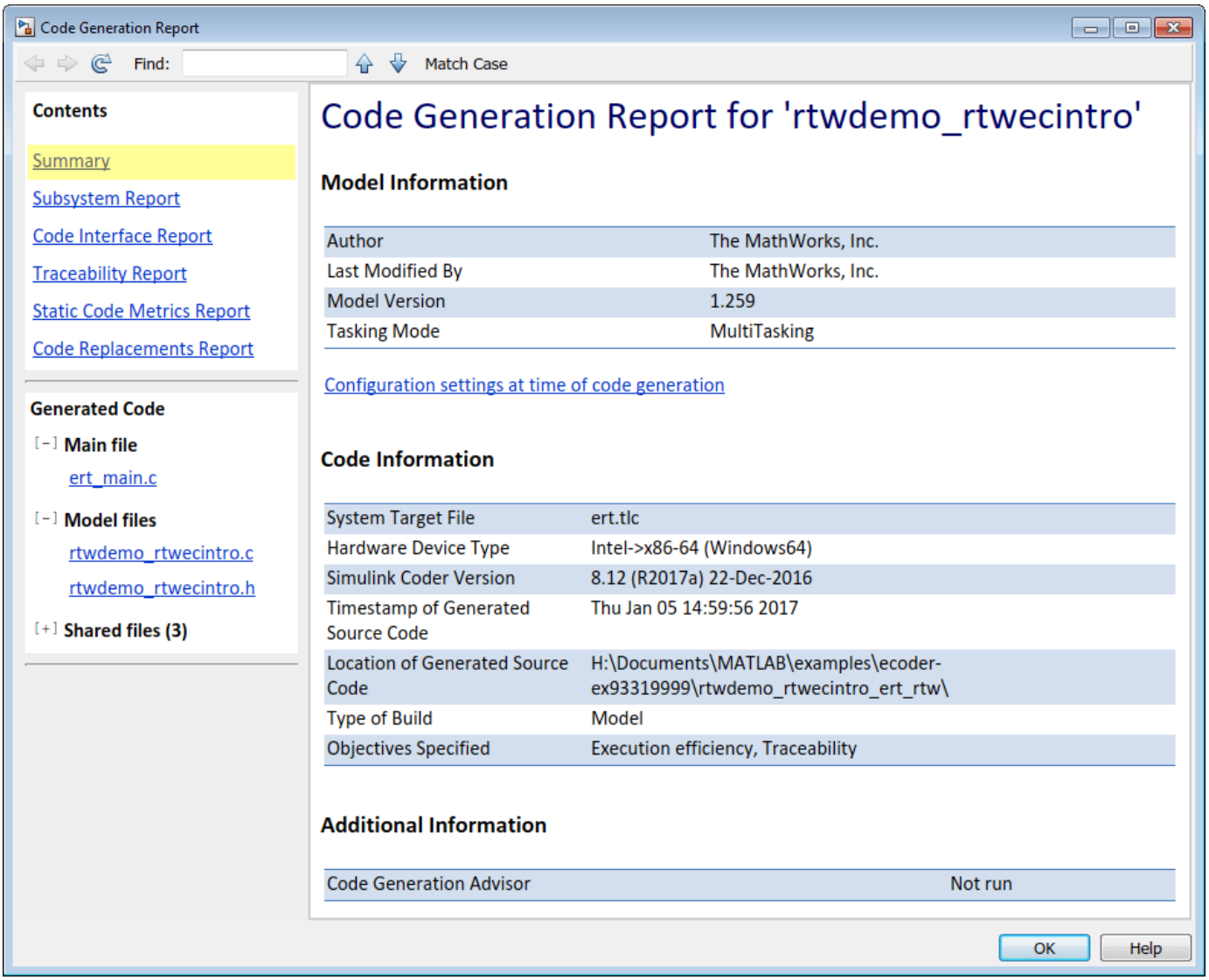
You can select and prioritize a combination of objectives before generating code.

8. In the model window, initiate code generation and the build process for the model by using one of these options:

- Click the Build Model button.
- Press **Ctrl+B**.
- Invoke the `slbuild` command from the MATLAB® command line.

9. View the code generation report.

The report includes `rtwdemo_rtwecintro.c`, associated utility and header files, and traceability and validation reports.



This figure contains a portion of `rtwdemo_rtwecintro.c`

Step function for model: rtwdemo_rtwecintro

File: rtwdemo_rtwecintro.c

```
1   /* Model step function */
2   void rtwdemo_rtwecintro_step(void)
3   {
4     boolean_T rtb_equal_to_count;
5
6     /* Sum: '<Root>/Sum' incorporates:
7      *  Constant: '<Root>/INC'
8      *  UnitDelay: '<Root>/X'
9      */
10    rtDWork.X = (uint8_T)(1U + (uint32_T)rtDWork.X);
11
12    /* RelationalOperator: '<Root>/RelOpt' incorporates:
13     *  Constant: '<Root>/LIMIT'
14     */
15    rtb_equal_to_count = (rtDWork.X != 16);
16
17    /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
18     *  TriggerPort: '<S1>/Trigger'
19     */
20    if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
21    {
22      /* Outport: '<Root>/Output' incorporates:
23       *  Gain: '<S1>/Gain'
24       *  Inport: '<Root>/Input'
25       */
26      rtY.Output = rtU.Input << 1;
27    }
28
29    rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
30      POS_ZCSIG : (int32_T)ZERO_ZCSIG);
31
32    /* End of Outputs for SubSystem: '<Root>/Amplifier' */
33
34    /* Switch: '<Root>/Switch' */
35    if (!rtb_equal_to_count) {
36      /* Update for UnitDelay: '<Root>/X' incorporates:
37       *  Constant: '<Root>/RESET'
38       */
39      rtDWork.X = 0U;
40    }
41
42    /* End of Switch: '<Root>/Switch' */
43  }
```

10. Close the model.

```
bdclose(model)
rtwdemoclean;
```

Related Topics

- “Configure Model, Generate Code, and Simulate”
- “Configure a System Target File”
- “Generate Code and Simulate Models in a Project”
- “Generate Code and Simulate Models Programmatically with Project” on page 52-106

See Also

More About

- “Generate Code by Using the Quick Start Tool” on page 53-9

Generate Code by Using the Quick Start Tool

The Quick Start tool helps you prepare a model for generating readable, efficient code. To start the tool, open the Embedded Coder app. On the **C Code** tab, click **Quick Start**.

After you start the tool, you must answer these questions about the code that you want to generate:

- What is the model or subsystem for code generation?
- What is the type of code output for your generated code?
- Does your application require reentrant, multi-instance code?
- What is the target hardware processor type?
- What is your primary code generation objective?

The tool validates your choices against the model and presents the parameter changes required to generate code. If you choose to generate code, the tool applies the parameter changes to your configuration set and generates the code. After code generation, you can view the code generation report and find information on building, customizing, optimizing, and packaging the code. For more information about the code generation report, see “Reports for Code Generation” on page 54-2.

To further customize your generated code, click **Finish**, and then use the Embedded Coder app with the Code perspective environment. The Code perspective provides the tools to control the names and representation of the model data and functions in the generated code. For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

Quick Start Model Analysis

At each step of the quick start process, the tool validates your model against your selections. The tool checks for model conditions that prevent you from proceeding with code generation. During the analysis step, the tool must also examine your model or subsystem for answers to the following questions. The answers help determine the best configuration for the deployment of your code.

How many sample rates are in your system?

The Quick Start tool evaluates your model to determine the number of periodic sample rates in your system.

Single rate	Your model has only one periodic sample rate. The generated code has a single-entry point function that runs at the time interval of the sample rate.
Multirate	Your model has more than one periodic sample rate. It is possible that the generated code does not execute at the same time intervals. After the analysis step, you can choose to generate a single-entry point function for each of the sample rates or generate a different entry point function for each sample rate. If you choose to generate multitasking code, the code generator produces multiple entry-point functions. These functions run as multiple tasks. Each entry-point function is called at an interval defined by the sample rate that is configured in the model.

If your model contains an asynchronous rate, an additional entry-point function is generated to run at the specific interrupt time.

For more information about sample rates, see “Time-Based Scheduling and Code Generation”.

Does your system contain continuous states?

The Quick Start tool evaluates your model for continuous blocks to determine the correct solver to use.

No	If your system does not contain continuous states, the Quick Start tool configures your model to use a fixed-step discrete solver for code generation if you have not selected one.
Yes	If your system does contain continuous states, the Quick Start tool configures your model to use a fixed-step continuous solver for code generation if you have not selected one. The tool also selects the `SupportContinuous` configuration parameter.

For more information on solvers, see “Solver Selection Criteria”.

Did you configure your system for export function calls?

The Quick Start tool evaluates your model to see if scheduler code must be generated.

No	If you did not configure your system for export function calls, the generated code includes code for the system algorithm and the scheduler code.
Yes	If you configured your system for export function calls, the generated code includes code for the system algorithm. You can manually write the scheduler code or generate it from other models.

For more information, see “Export-Function Models Overview”.

Does your system contain referenced models?

The Quick Start tool evaluates your model to see if it depends on code from other models.

No	If your system does not contain referenced models, the generated code does not depend on code from other models.
Yes	If your system contains referenced models, the generated code for your model depends on other modules generated from referenced models. The code generator can optimize the generated code because it is aware of the relationship between your model and the referenced models. When the Quick Start tool applies configuration parameter changes to your model, it also applies the changes to the referenced models.

For more information, see “Generate Code for Model Reference Hierarchy”.

Configuration Parameter Changes for Models That Have a Configuration Reference

To apply configuration parameter changes to a model that has an active configuration reference, the Quick Start tool:

- Creates a `Simulink.ConfigSet` object `QuickStart_`*`timestamp`* in the workspace or data dictionary that contains the original configuration set. The new object is a copy of the original configuration set that has the parameter changes applied.
- Creates a `Simulink.ConfigSetRef` object that points to the new configuration set object.
- Attaches the new configuration reference to the model and makes it the active configuration.

- If your system contains referenced models that share your original configuration set, the tool attaches the new configuration reference to the referenced models and makes the new configuration reference the active configuration.

To restore the original configuration set, activate the original `Simulink.ConfigSetRef` object.

Note If the Quick Start tool creates the configuration set object in the MATLAB workspace, you must save the object to preserve the configuration set after the MATLAB session ends. For more information, see “Save a Configuration Set”.

Next Steps

After you generate code by using Quick Start tool, possible next steps are:

- “Preserve Variables in Generated Code” on page 36-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” on page 49-2
- “Generated Code Compilation”
- “Manage Configuration Sets for a Model”
- “Build and Run a Program” on page 59-42
- “Relocate or Share Generated Code”

To control the names and representation of the model data and functions in the generated code, use the Code Mappings editor and the Model Data Editor in the Embedded Coder app. For more information, see “C Code Generation Configuration for Model Interface Elements” on page 33-2.

See Also

Related Examples

- “Generate C or C++ Code from Stateflow Blocks” (Stateflow)

Manage File Packaging of Generated Code Modules

The code generator produces code modules. The file packaging configuration controls where the code generator places code into code modules and header files.

To locate and examine the generated code files, use the **Code** view or the HTML code generation report. The code generation report provides hyperlinks in the comments that you can click to view the generated code in the MATLAB Help browser. For more information, see “Traceability in Generated Code” on page 85-3.

In this section...
“Generated Code Modules” on page 53-12 “User-Written Code Modules” on page 53-14 “Customize Generated Code Modules” on page 53-14

Generated Code Modules

The code generator creates a build folder in your working folder to store generated source code. The build folder contains object files, a makefile, and other files created during the code generation process. The default name of the build folder is `model_ert_rtw`.

Code Modules and Header Files Affected by File Packaging summarizes the structure of source code that the code generator produces.

You can customize the generated set of files in several ways:

- File packaging formats: Manage the number of source files generated for your model. Select model configuration parameter **File packaging format**. For more information, see “Customize Generated Code Modules” on page 53-14.
- Nonvirtual subsystem code generation: Instruct the code generator to produce separate functions within separate code files for nonvirtual subsystems. You can control the names of the functions and of the code files. For more information, see “Generate Subsystem Code as Separate Function and Files” on page 4-7.
- Storage classes: Define and use storage classes that partition generated data structures into different files based on file names that you specify. For more information, see “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67.
- Module packaging: Direct the generated code into a required set of `.c` or `.cpp` and `.h` files, and control the internal organization of the generated files. For details, see “Code Interface Configuration”.

Code Modules and Header Files Affected by File Packaging

File	Description
model.c or .cpp	Contains entry points for code implementing the model algorithm (for example, *model*_step, *model*_initialize, and *model*_terminate).
*model*_private.h	Contains local macros and local data that the model and subsystems require. This file is included in the model.c file as a #include statement. You do not need to include *model*_private.h when interfacing handwritten code to the generated code of a model.
model.h	Declares model data structures and a public interface to the model entry-points and data structures. Provides an interface to the real-time model data structure (*model*_M) with accessor macros. The code generator: • Produces a separate header file for each Simulink Function block in a model. • Includes *model*.h in the subsystem .c or .cpp files of a model. If you interface handwritten code to generated code for one or more models, include *model*.h for each of those models.
*model*_data.c or .cpp	Contains (if conditionally generated) the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations for the model structure data types. If the model does not use these data structures and zero representations, *model*_data.c or .cpp is not generated. These structures and zero representations are declared extern in *model*.h.
*model*_types.h	Provides forward declarations for the real-time model data structure and the parameters data structure. Function declarations of reusable functions can require these declarations. Provides type definitions for user-defined types that the model uses.
rtwtypes.h	Defines data types, structures, and macros required by generated code. For more information, see “Control Placement of rtwtypes.h for Shared Utility Code”.
multiword_types.h	Contains type definitions for wide data types and their chunks. File is generated when multiword data types are used or when you select one or more of these model configuration parameters: • **MAT-file logging** • **External mode**
model_reference_types.h	Contains type definitions for timing bridges. File is generated for a referenced model or a model containing Model blocks.
builtin_typeid_types.h	Defines an enumerated type corresponding to built-in data types. File is generated when your model contains a Stateflow chart that uses messages or when you select one or more of these configuration parameters: • **MAT-file logging** • C API model configuration parameter

File	Description
`zero_crossing_types.h`	Contains zero-crossing definitions for models with triggered subsystems where the trigger is `rising`, `falling`, or `either`. File is generated only if required by the model.
`ert_main.c` or `.cpp`	(optional file) If you select model configuration parameter **Generate an example main program**, this file is generated. For more information, see Generate an example main program.
`rtmodel.h`	(optional file) If you clear model configuration parameter **Generate an example main program**, this file is generated. For more information, see Generate an example main program. Contains `#include` directives required by the `rt_main.c` or `rt_cppclass_main.cpp` static main program module. Includes `rtmodel.h` to access model-specific data structures and entry points, because the static main program module is not created at code generation time. For more information, see “Static Main Program Module” on page 71-8.
`model``_capi.c` or `.cpp` *`model`*`_capi.h`	(optional file) Provides data structures that enable a running program to access model signals, states, and parameters without external mode. To learn how to generate and use the *`model`*`_capi.c` or `.cpp` and `.h` files, see “Exchange Data Between Generated and External Code Using C API”.

User-Written Code Modules

Code that you write to interface with generated model code usually includes a customized main module. Base this module on a main program produced by the code generator. This customized main module can also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code. Steps to set up the development environment to support a customized main module include:

1. Establish a working folder for your own code modules.
2. Put your working folder on the MATLAB path.
3. At minimum, inform the build process about the location of your source and object files with model configuration parameter **Additional build information**.
4. Your development process could require generating code for a particular microprocessor or development board and deploying the code on target hardware with a cross-development system. To accomplish these goals, make more extensive modifications to the ERT-based system target file.

For information on how to customize your ERT-based system target file for your production requirements, see “Custom Software for Target Hardware”.

Customize Generated Code Modules

Model configuration parameter **File packaging format** specifies how the code generator packages generated source code into files. You can set the parameter to `Modular`, `Compact (with separate data file)`, and `Compact`. This table describes the generated files and the removed files for each file packaging format.

Generated Files According to File Packaging Format

File Packaging Format	Generated Files	Removed Files
`Modular` (default)	*`model`*`.c` subsystem files (optional) *`model`*`.h` *`model`*`_types.h` *`model`*`_private.h` *`model`*`_data.c` (conditional)	None
`Compact (with separate data file)`	*`model`*`.c` *`model`*`.h` *`model`*`_data.c` (conditional)	*`model`*`_private.h` *`model`*`_types.h` (conditional, see Removed Files According to File Packaging Format)
`Compact`	*`model`*`.c` *`model`*`.h`	*`model`*`_data.c` *`model`*`_private.h` *`model`*`_types.h` (conditional, see Removed Files According to File Packaging Format)

This table describes content placement from the removed files.

Removed Files According to File Packaging Format

Removed File	Generated Content In File
`model``_private.h`	*`model`*`.c` and *`model`*`.h`
`model``_types.h`	*`model`*`.h`
`model``_data.c`	*`model`*`.c`

You can specify a different file packaging format for each referenced model.

Model configuration parameter **Shared code placement** interacts with file packaging operations. If you set **Shared code placement** to `Shared location`, the code generator produces separate files for utility code in a shared location, regardless of the file packaging format. If you set **Shared code placement** to `Auto`, the code generator generates code for utilities according to the file packaging format selection.

- `Modular`: Some shared utility files are in the build folder.
- `Compact (with separate data file)`: Utility code is generated in *`model`*`.c`.
- `Compact`: Utility code is generated in *`model`*`.c`.

File packaging formats `Compact` and `Compact (with separate data file)` generate *`model`*`_types.h` for models containing:

- A Variant Subsystem block. The *model*_types.h file includes preprocessor directives defining the variant objects associated with a variant block.
- Storage classes generating a separate header file.

File packaging formats `Compact` and `Compact (with separate data file)` are not compatible with:

- A model containing a subsystem, which is configured to generate separate source files
- A model containing a noninlined S-function
- A model for which **Shared code placement** is set to `Auto`, which uses data objects for which **Data scope** is set to `Exported`

See Also

More About

- “Manage Build Process Folders” on page 52-30
- “Manage Build Process Files” on page 52-35
- “Manage Build Process File Dependencies” on page 52-43
- “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2

Tutorial for Generating C Code from a Control Algorithm for an Embedded System

This guided tutorial uses a series of examples to show the workflow of generating code from a control algorithm for an embedded system. In each example, you learn how to perform a specific aspect of code generation or integration. The examples are based on a shared model. Each example has a unique version of that model and corresponding data set so that you can use the examples independently. You do not have to run the examples in a specific order. When you transition between examples, you save the current model so that each example captures your modifications to the model and model data.

To recover a model in its original state, delete your local copy of the model and model data. The model data is saved as `PCG_Demo_#_data.mat`.

Use these hyperlinks to navigate to the examples in the tutorial.

1. “Prepare a Control Algorithm Model for C Code Generation” on page 53-18
2. “Configure Data Interface in the Generated Code” on page 53-29
3. “Partition Functions in Generated Code” on page 53-39
4. “Call External C Code from Model and Generated Code” on page 53-48
5. “Build Integrated Code Outside the Simulink Environment” on page 53-53
6. “Test Generated Code” on page 53-58
7. “Evaluate Performance of the Generated Code” on page 53-64
8. “Install and Use Cygwin and Eclipse” on page 53-68

Third-Party Software

Some of the examples use the Eclipse™ IDE and the Cygwin™/gcc compiler. Instructions to install and use Eclipse™ and Cygwin™/gcc are available in “Install and Use Cygwin and Eclipse” on page 53-68.

Prepare a Control Algorithm Model for C Code Generation

Generate code for a control algorithm model, integrate the generated code with an existing system, and validate simulation and executable results.

Understand the Model

This example shows and describes the model from behavioral and structural perspectives. The example also shows how to configure a model for code generation and how to generate code. You learn how to:

- Understand the functional behavior of the example model.
- Understand how to validate the model.
- Become familiar with model-checking tools.
- Become familiar with configuration options that affect code generation.
- Generate code from a model.

Understand the Functional Design of the Model

This example uses a simple model of a throttle controller. The model features redundancy, which safety-critical, drive-by-wire applications commonly use. The model highlights a standard model structure and a set of basic blocks in algorithm design.

In the current configuration, the code that the model generates is not configured for a production target system. In this example, you change the target configuration and observe the resulting changes to the format of the generated code.

Inspect the Top-Level Model

Open the example model.

The top-level model consists of:

- Four subsystems: `PI_ctrl_1`, `PI_ctrl_2`, `Define_Throt_Param`, and `Pos_Command_Arbitration`.
- Top-level inputs: `pos_rqst`, `fbk_1`, and `fbk_2`.
- Top-level outputs: `pos_cmd_one`, `pos_cmd_two`, and `ThrotComm`.
- Signal routing.
- No transformative blocks. Transformative blocks change the value of a signal, for example, Sum and Integrator blocks.

The layout shows a basic model architectural style.

- Separation of calculations from signal routing (lines and buses)
- Partitioning into subsystems

This style suits many types of models.

Inspect Subsystems

Two subsystems represent PI controllers: `PI_ctrl_1` and `PI_ctrl_2`. The subsystems have identical content and, for now, use identical data. Later, you use the subsystems to learn how the code generator can create reusable functions.

The PI controllers come from a *library*, which is a group of related blocks or models that you intend to reuse. Libraries provide one of two methods for including and reusing models. You see the second method, model referencing, later in this series.

When you use a library block in a model, you cannot edit the instance of the block in the model. Instead, edit the block definition in the library. Then, instances of the block in different models remain consistent.

Open the subsystem `PI_ctrl_1`.

The Stateflow® chart `Pos_Command_Arbitration` performs basic error checking on the two command signals. If the command signals are too far apart, the chart sets the output to a `fail_safe` position.

Open `Pos_Command_Arbitration`.

Inspect Model Configuration Parameter Settings for Code Generation

To prepare a model for code generation, open the **Embedded Coder** app. Then, set code generation model configuration parameters. The parameters determine the method that the code generator uses to generate the code and the resulting code format.

Code Generation Objectives

You can manually configure the model configuration parameters. Alternatively, you can choose from predefined objectives to automatically configure the model configuration parameters.

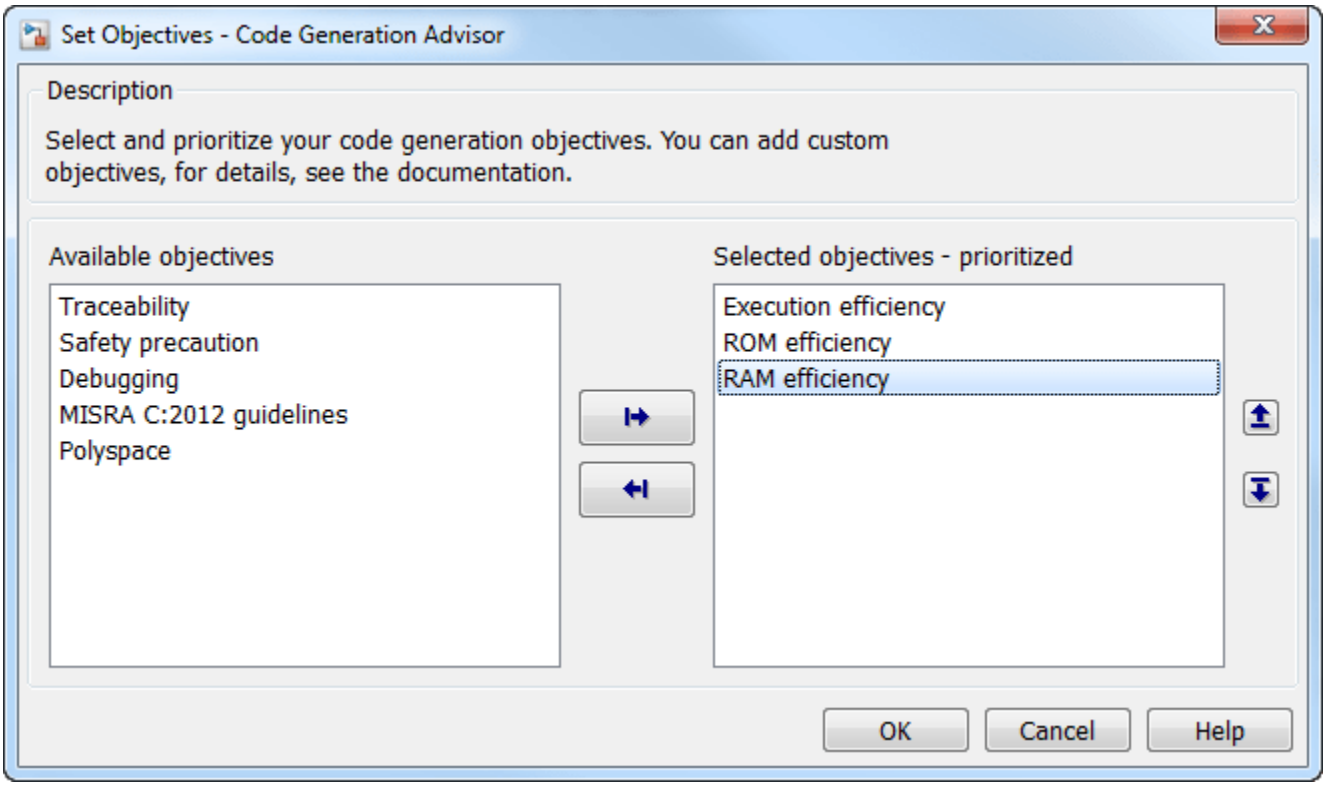
You can choose from these high-level code generation objectives:

- Execution efficiency
- ROM efficiency
- RAM efficiency
- Traceability
- Safety precaution
- Debugging

Each objective checks the current model configuration parameters against the recommended values of the objectives. Each objective also includes a set of Code Generation Advisor checks. You can use these additional checks to verify that the model configuration parameters are set to create code that meets the objectives.

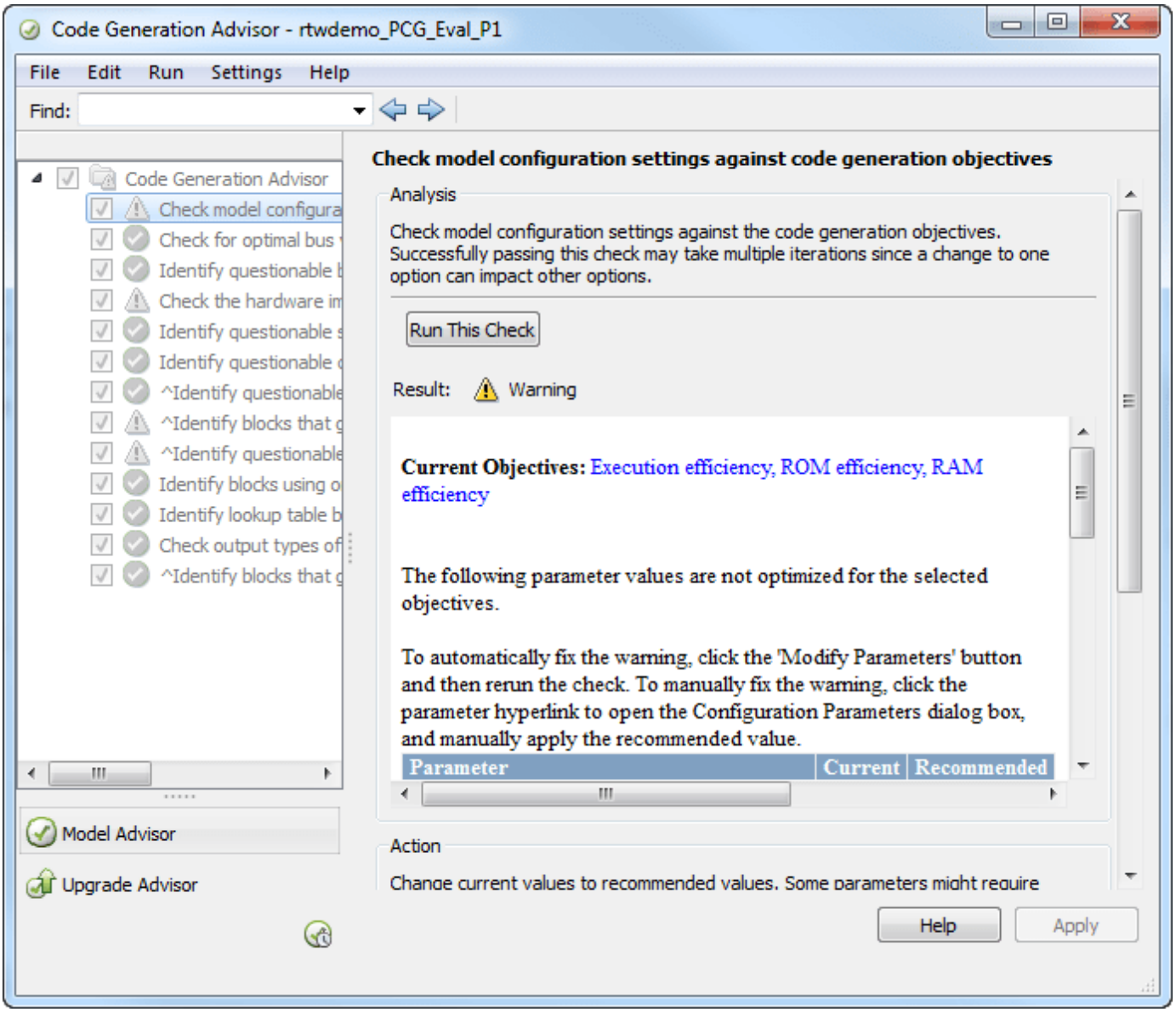
Some of the recommendations that the objectives make conflict with the Code Generation Advisor checks. The order in which you select objectives determines the result. Simulink® resolves conflicts by satisfying objectives that have a higher priority.

The figure shows how to set the priority to `Execution efficiency` > `ROM efficiency` > `RAM efficiency`. To open the dialog box, in the Configuration Parameters dialog box, select the **Code Generation** pane. Then, click **Set objectives**.



You can run the Code Generation Advisor to check the model based on the specified objectives. To open the Code Generation Advisor, in the **C Code** tab, click **C/C++ Code Advisor**.

The Code Generation Advisor creates a list of checks based on the objectives that you select. The first check reviews current values of the model configuration parameters and suggests alternative values based on the objectives. The check provides an automated method for setting the parameters to recommended values.



Manual Configuration Options

In the Model Configuration Parameters dialog box, these panes are relevant to code generation:

- Solver
- Hardware Implementation
- Code Generation

Solver

Open the **Solver** pane.

- To generate code for a model, you must set model configuration parameter **Type** to `Fixed-step`.
- Parameter **Fixed-step size** sets the base rate of the system. Set the parameter to the lowest common multiple of rates in the system.

- Parameter **Solver** controls the integration algorithms that the code generator uses.
- To generate an entry-point function for each rate in the system, select parameter **Treat each discrete rate as a separate task**.

Hardware Implementation

Open the **Hardware Implementation** pane.

Use hardware implementation parameters to specify a hardware board. Simulink® adjusts other settings on the pane, including hidden microprocessor device details, based on your board selection. To view or adjust the hidden parameter settings, such as the word size and byte ordering, click **Device details**.

Code Generation

Open the **Code Generation** pane.

Use the **Code Generation** pane to specify the system target file and optimizations. This example uses the Embedded Coder® system target file (`ert.tlc`). You can extend that system target file to create a customized configuration. Some of the basic parameters on the **Code Generation** pane and its subpanes include:

System target file

- ert.tlc - "Base" Embedded Coder®
- grt.tlc - "Base" Generic Real-Time Target
- Hardware-specific targets

Make file

Code optimizations

- Remove unused branches from the code and control the creation of temporary variables.
- Control which signals have explicit initialization code.
- Enable and disable use of overflow and division-by-zero protection code.

Code formatting options

- Use of parentheses
- Header file information
- Variable naming conventions

Inclusion of custom code

- C files
- H files
- Object files
- Folder paths

Generation of ASAP2 files

Save Model Configuration Parameters

You can save the values of model configuration parameters as a MATLAB® function. At the command prompt, enter:

```
hCs = getActiveConfigSet('rtwdemo_PCG_Eval_P1');
hCs.saveAs('ConfiguredData');
```

The MATLAB® function saves a textual representation of the configuration parameter object. You can use the generated file for archiving or comparing different versions of the files by using traditional diff tools. You can also visually inspect the contents of the file.

You can run the function to set the configuration parameters of other models.

```
hCs2 = ConfiguredData;
attachConfigSet('myModel', hCs2, true);
setActiveConfigSet('myModel', hCs2.Name);
```

Understand the Simulation Testing Environment

You test the throttle controller model in a separate model called a *test harness*. A test harness is a model that evaluates the control algorithm. Using a test harness:

- Separates the test data from the control algorithm
- Separates the plant or feedback model from the control algorithm
- Provides a reusable environment for multiple versions of the control algorithm

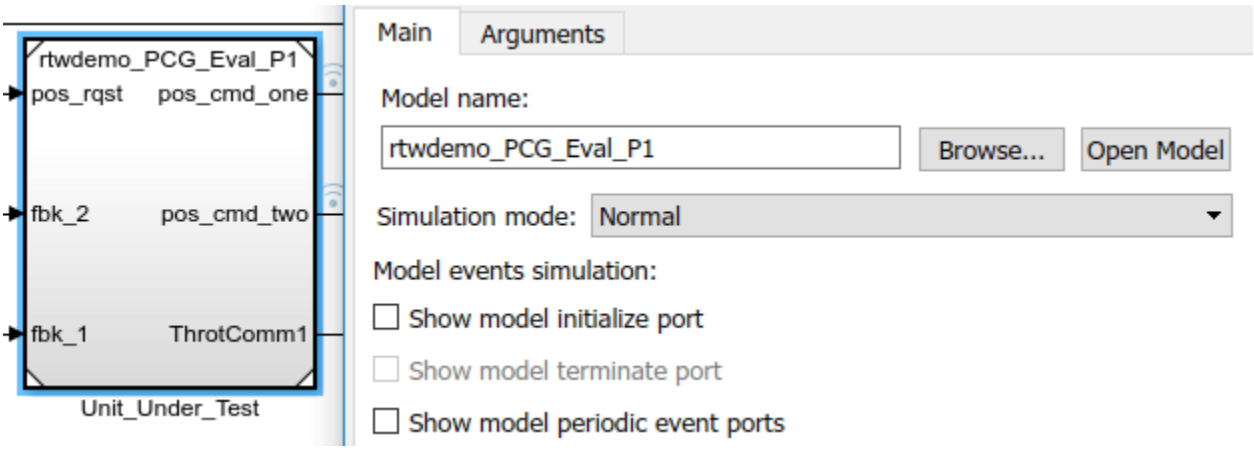
Open the test harness.

A typical simulation testing environment consists of these parts:

- Unit under test
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

Highlight the unit under test.

The control algorithm is the *unit under test*. The test harness model references the control algorithm through a Model block. With Model blocks, you can reuse components. The Model block refers to the control algorithm by name (`rtwdemo_PCG_Eval_P1`).



The Model block enables inclusion (referencing) of a model in another model as a *compiled function*. By default, Simulink® compiles the referenced model when you change it. Compiled functions have these advantages over libraries:

- Large models simulate faster.
- You can simulate compiled functions directly.
- The simulation requires less memory. When you add multiple instances of the model (multiple Model blocks), only one copy of the compiled model exists in memory.

Highlight the test vector source.

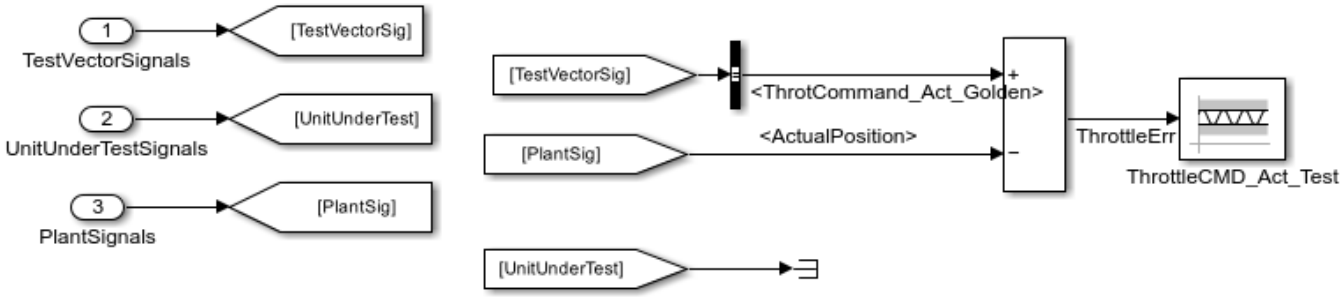
The model uses a Signal Editor block as the test vector source. The block has data that drives the simulation (`pos_rqst`) and the expected results that the Verification subsystem uses. This example uses only one set of test data, though in a typical application you create a test suite that fully exercises the system.

Highlight Verification.

The test harness compares the simulation results against *golden data*, which is a set of test results that indicate the desired behavior for the model. In this model, the V&V Assertion block compares the simulated throttle value position from the plant against the golden value that the test harness provides. If the difference between the two signals is greater than 5%, the test fails and the Assertion block stops the simulation.

Alternatively, you can evaluate the simulation data after the simulation completes execution. You can use either MATLAB® scripts or third-party tools to perform the evaluation. Post-execution evaluation provides greater flexibility in the analysis of the data, though you must wait until execution finishes. Combining the two methods can yield a highly flexible and efficient test environment.

```
load_system('rtwdemo_PCGEvalHarness')
open_system('rtwdemo_PCGEvalHarness/Verification')
```



Highlight the plant/feedback system.

This example models the throttle dynamics by breaking a transfer function down into a canonical form. You can create plant models to model a specific level of fidelity. Many applications use a different plant model at each stage of testing.

Highlight scaled input and output.

The subsystems that scale input and output perform these primary functions:

- Select signals to route to the unit under test and to the plant.
- Rescale signals between engineering units and units that the unit under test requires.
- Handle rate transitions between the plant and the unit under test.

Run Simulation Tests

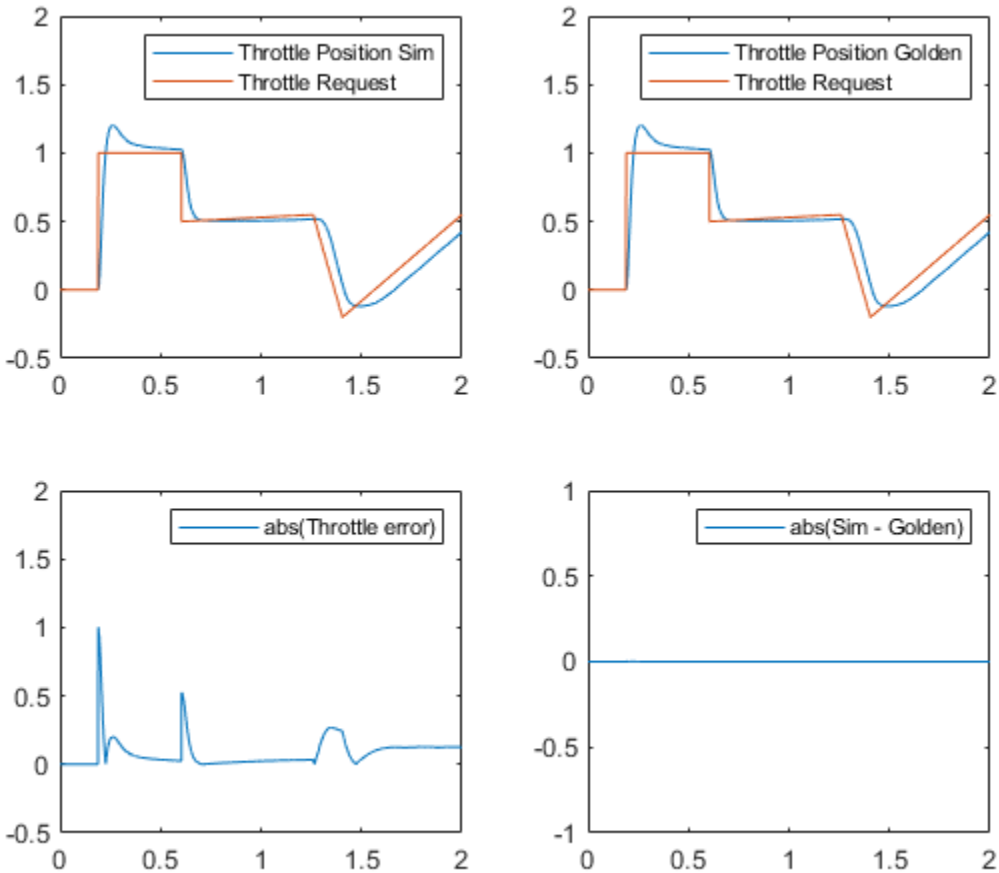
To run the test harness model simulation, click **Start** or click this hyperlink.

Run the test harness.

The first time the test harness runs, Simulink® must compile the referenced model. You can monitor the compilation progress in the Command Window.

When the model simulation is complete, Simulink® displays the results in a plot figure.

```
sim('rtwdemo_PCGEvalHarness')
```



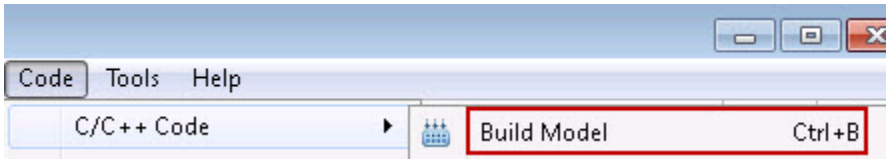
The lower-right plot shows the difference between the expected (golden) throttle position and the throttle position that the plant calculated.

Generate Code

To generate code, open the **Embedded Coder** app. Then, using one of these techniques:

- In the model, press **Ctrl+B**.
- In the **C Code** tab, select **Build** or **Generate Code**.
- Click this hyperlink:

Generate code for the model.



The code generator produces several files. The resulting code, though computationally efficient, is not yet organized for integration into the production environment.

Examine the Generated Code

The code generator creates multiple files that you can view from the **Code** view or code generation report. In addition to the standard C definition and header files, the code generator creates a set of HTML files. The HTML files provide hyperlinks between the code and the model.

In the Model Explorer, open the HTML Code Browser.

In the generated code, observe that:

- The controller code exists in one function called `ModelName_step`, which is in the file `rtwdemo_PCG_Eval_P1.c`.
- The code generator folds the operations of multiple blocks into one statement of code.
- The function `ModelName_initialize` initializes variables.
- Data structures define model data (for example, `rtwdemo_PCG_Eval_P1_U.pos_rqst`).

```
/* Model step function */
void rtwdemo_PCG_Eval_P1_step(void)
{
  real_T rtb_Sum3;
  real_T rtb_IntegralGainShape;
  real_T Discrete_Time_Integrator1;
  real_T Discrete_Time_Integrator1_i;

  /* Sum: '<S2>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_1'
   *  Inport: '<Root>/pos_rqst'
   */
  rtb_IntegralGainShape = rtwdemo_PCG_Eval_P1_U.pos_rqst -
    rtwdemo_PCG_Eval_P1_U.fbk_1;
```

- `rtwdemo_PCG_Eval_P1.c`: C file that defines step and initialization functions
- `ert_main.c`: Example Main file that includes a simple scheduler
- `rtwdemo_PCG_Eval_P1.h`: H file that contains type definitions of the Simulink® Coder™ data structures
- `PCG_Eval_p1_private.h`: File that declares data that only the generated code uses
- `rtwdemo_PCG_Eval_P1_types.h`: H file that declares the real-time model data structure

For the next example in this series, see “Configure Data Interface in the Generated Code” on page 53-29.

See Also

More About

- “Support Model Referencing” on page 94-61

- “Select and Configure C or C++ Compiler” on page 59-3
- “Recommended Settings Summary for Model Configuration Parameters”
- “Signal Groups”
- “Simulink Check”

Configure Data Interface in the Generated Code

Specify signals, states, and parameters for inclusion in generated code.

Learn how to control these attributes of data in the generated code:

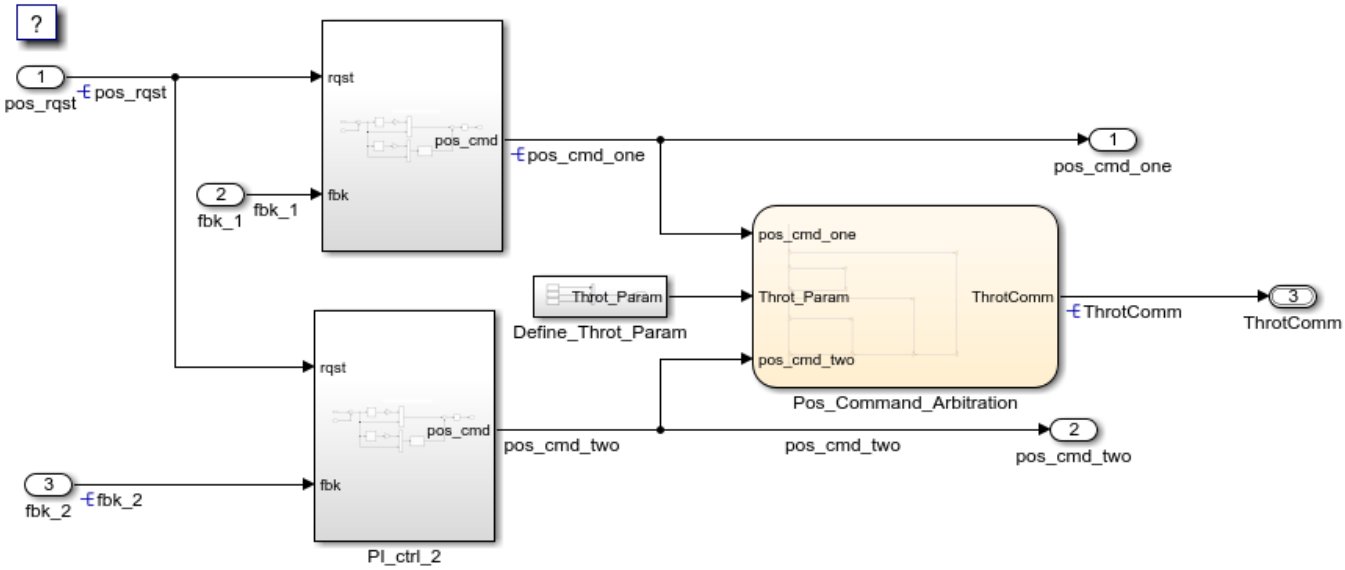
- Name
- Data type
- Data storage class

For information about the example model, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Open Example Model

Open the example model, `rtwdemo_PCG_Eval_P2`.

```
open_system('rtwdemo_PCG_Eval_P2')
```



Data Declaration

Most programming languages require that you declare data and functions before using them. The declaration specifies:

- Scope: The region of the program that has access to the data
- Duration: The period during which the data exist in memory
- Data type: The amount of memory allocated for the data
- Initialization: A value, a pointer to memory, or NULL

The combination of scope and duration is the storage class. If you do not provide an initial value, most compilers assign a zero value or a null pointer.

Supported data types include:

- `double`: Double-precision floating point
- `single`: Single-precision floating point
- `int8`: Signed 8-bit integer
- `uint8`: Unsigned 8-bit integer
- `int16`: Signed 16-bit integer
- `uint16`: Unsigned 16-bit integer
- `int32`: Signed 32-bit integer
- `uint32`: Unsigned 32-bit integer
- Fixed Point: 8-, 16-, 32-bit word lengths

With storage classes you can:

- Generate exported files, with custom names, that contain global variable declarations and definitions.
- Import custom header files that contain global variable declarations.
- Generate `const` or `volatile` type qualifiers in declarations.
- Represent a parameter as a macro (`#define` or compiler option).
- Package signals or parameters into flat structures or bit fields.

Control Data in Simulink and Stateflow

This example uses data objects to specify code generation settings for data. Alternatively, you can store the settings in the model by using dialog boxes. Both methods allow full control over the data type and storage class. You can use both methods in a single model.

This example focuses on these types of data objects:

- Signal
- Parameter
- Bus

The code generator uses the objects from the MATLAB® base workspace or a Simulink® data dictionary. You can create and inspect the objects with commands at the command prompt or by using the Model Data Editor or the Model Explorer with the Code Mappings Editor.

As an example, inspect the definition of the `Simulink.Signal` object `pos_cmd_one`, which the model created in the base workspace:

1. Open the **Embedded Coder** app.
2. On the **Modeling** tab, click **Model Data Editor**.
3. In the Model Data Editor, inspect the **Signals** tab.
4. Click the **Show/refresh additional information** button.
5. Next to the **Filter contents** box, toggle the **Filter using selection** button.

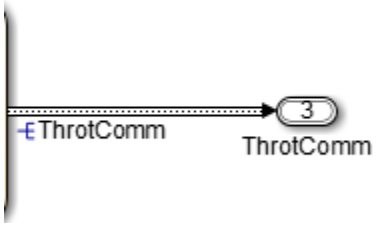
6. In the model, select the `pos_cmd_one` signal line. Now, the data table contains two rows: One that corresponds to the signal line in the model and one that corresponds to the signal object in the base workspace. For the row that corresponds to the signal line, the **Resolve** check box is selected, which means the signal line acquires settings from the signal object.
7. Optionally, to inspect other properties of the signal object, adjust the setting for the **Change view** drop-down list.
8. In the model, select the `pos_cmd_one` signal line and pause on the ellipsis that appears above or below the signal line to open the action bar. Click the Add Signal button.
9. Open the **Code Mappings Editor**, click the **Signals** tab. Inspect the `pos_cmd_one` signal storage class.

Return to the Model Data Editor, toggle the **Filter using selection button**. Now, the data table shows these signal lines and signal objects:

- `pos_cmd_one`
- `pos_rqst`
- `P_InErrMap`
- `ThrotComm`

`ThrotComm` is a bus signal, which is an instantiation of the `Simulink.Bus` object `ThrottleCommands`. If a bus signal is nonvirtual, the signal appears as a structure in the C code. Bus objects do not appear in the Model Data Editor. However, you can inspect the object in the Model Explorer (**View > Model Explorer**).

As in C, you can use a bus definition (`ThrottleCommands`) to create multiple instances of the structure. In a model diagram, a bus signal appears as a wide line with central dashes.



A data object has properties that you configure for simulation and for code generation, including:

- `DataType` (numeric data type for storage in the generated code)
- `StorageClass` (storage class for code generation)
- `Value` (parameter value)
- `InitialValue` (initial value for signal)
- `Alias` (alternative name for the data, which the code generator uses)
- `Dimensions` (size and number of dimensions of parameter or signal value)
- `Complexity` (numeric complexity)
- `Unit` (physical units such as cm)
- `Min` (minimum value)
- `Max` (maximum value)

Use the property `Description` to specify custom documentation for data objects.

Add New Data Objects

You can create data objects for named signals, states, and parameters. To associate a data object with a construct, the construct must have a name.

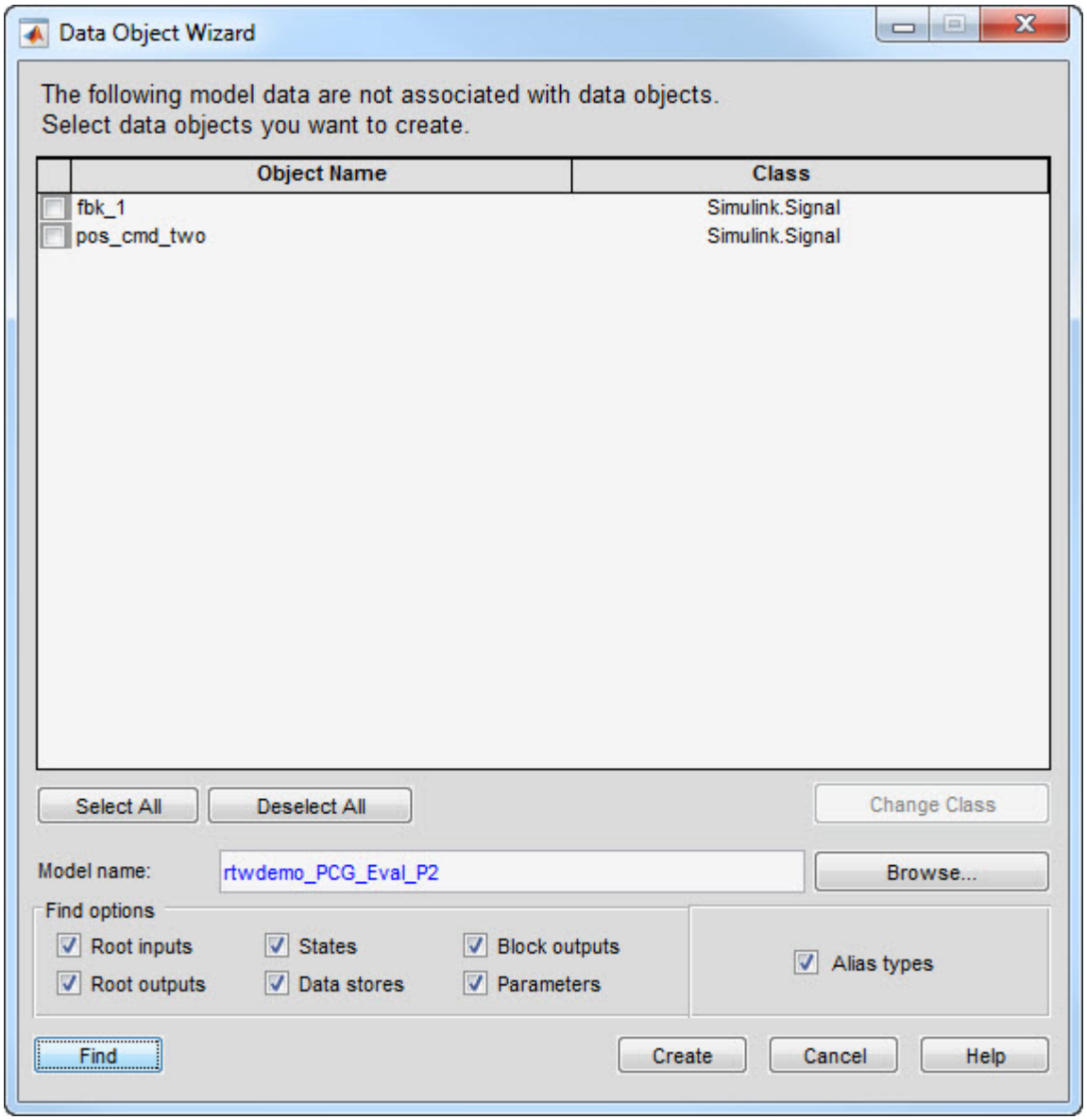
The Data Object Wizard tool finds constructs for which you can create data objects, and then creates the objects for you. The example model includes two signals that are not associated with data objects: `fbk_1` and `pos_cmd_two`.

To find the signals and create data objects for them:

1. Open the Data Object Wizard.

```
dataobjectwizard('rtwdemo_PCG_Eval_P2')
```

2. Click **Find** to find candidate constructs.



3. Click **Select All** to select all candidates.

4. Click **Create** to create the data objects.

The Data Object Wizard performs the equivalent of the following commands:

```
fbk_1 = Simulink.Signal;
fbk_1.Dimensions = 1;
fbk_1.DataType = 'double';
outport = get_param('rtwdemo_PCG_Eval_P2/fbk_1','PortHandles');
outport = outport.Outport;
set_param(outport,'MustResolveToSignalObject','on')

pos_cmd_two = Simulink.Signal;
pos_cmd_two.Dimensions = 1;
pos_cmd_two.DataType = 'double';
```

```
outport = get_param('rtwdemo_PCG_Eval_P2/PI_ctrl_2','PortHandles');
outport = outport.Outport;
set_param(outport,'MustResolveToSignalObject','on')
```

Configure Data Objects

Set the data type and storage class for each data object. To configure data object properties, inspect the appropriate tab in the Model Data Editor. Use the **Change view** drop-down list to show design properties (such as data type) and code generation settings (such as storage class).

In the Model Data Editor, inspect the **Signals** tab and click the **Show/refresh additional information** button to display information for the signal objects that the Data Object Wizard created. Then, configure these properties for the objects:

- `fbk_1`: Data type `double`, storage class `ImportedExtern`
- `pos_cmd_two`: Data type `double`, storage class `ExportedGlobal`

Alternatively, you can use these commands at the command prompt to configure the objects:

```
fbk_1.DataType = 'double';
fbk_1.StorageClass = 'ImportedExtern';

pos_cmd_two.DataType = 'double';
pos_cmd_two.StorageClass = 'ExportedGlobal';
```

Control File Placement of Parameter Data

With Embedded Coder®, you can control the file placement of definitions for parameters and constants. The example model writes the parameter definitions to the file `eval_data.c`.

To change the placement of parameter and constant definitions, set the data placement options for the model configuration. In the Configuration Parameters dialog box, configure the options on the **Code Generation > Code Placement** pane.

In the example model, inspect the **Code Placement** pane in the Configuration Parameters dialog box. The model places data definitions in the file `eval_data.c` and declarations in the file `eval_data.h`.

Declare Block Parameters Tunable or Inlined

You can control the default tunability of block parameters in the generated code by setting model configuration parameter **Default parameter behavior**.

For the example model, the default parameter behavior is set to `Inlined`. By default, block parameters appear in the code as numeric literal values, not as variables stored in memory. You can use `Simulink.Parameter` objects to override inlining and preserve tunability for individual parameters.

Enable Signal Data Objects in Generated Code

Make sure that the signal data objects (`Simulink.Signal`) appear in the generated code. For individual signal lines in the model, select the option **Signal name must resolve to Simulink signal object** to explicitly resolve the signal name to a `Simulink.Signal` object in a workspace or data dictionary. In the Model Data Editor, set **Change view** to `Code` and select the check box in the **Resolve** column.

Alternatively, you can select the option for all signals in a model. At the command prompt, use the `disableimplicitsignalresolution` function.

View Data Objects in Generated Code

Generate code from the example model.

```
slbuild('rtwdemo_PCG_Eval_P2');
```

```
### Starting build procedure for: rtwdemo_PCG_Eval_P2
### Successful completion of code generation for: rtwdemo_PCG_Eval_P2

Build Summary

Top model targets built:

Model                Action           Rebuild Reason
========================================================================================
rtwdemo_PCG_Eval_P2  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.884s
```

Create a code generation report to more easily view the generated files. In the Simulink Editor, select model configuration parameter **Open Model Report**.

You can access these generated files:

- `rtwdemo_PCG_Eval_P2.c`: Defines the step and initialization functions. Uses the defined data objects.
- `eval_data.c`: Assigns initial values to the defined parameters. The model configuration specifically sets the file name.
- `eval_data.h`: Provides `extern` declarations of parameter data. The model configuration specifically sets the file name.
- `ert_main.c`: Defines scheduling functions.
- `rtwdemo_PCG_Eval_P2.h`: Contains the type definitions of the default structures that store signal, state, and parameter data. Due to the data object settings, some data appear in `eval_data.c` instead.
- `PCG_Eval_p2_private.h`: Declares private (local) data for the generated functions.
- `rtwdemo_PCG_Eval_P2_types.h`: Declares the real-time model data structure.
- `rtwtypes.h`: Provides mapping to the data types that Simulink® Coder™ defines (`typedef`). Used for integration with external systems.

For example, view the file `eval_data.c`, which allocates `const` memory for global variables that correspond to the `Simulink.Parameter` objects in the base workspace.

```
cfile = fullfile('rtwdemo_PCG_Eval_P2_ert_rtw','eval_data.c');
rtwdemodbtype(cfile,'/* Exported data definition */','* [EOF]',1,1)
```

```
/* Exported data definition */

/* Const memory section */
/* Definition for custom storage class: Const */
```

```
const real_T I_Gain = -0.03;
const real_T I_InErrMap[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5,
  1.0 } ;

const real_T I_OutMap[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 } ;

const real_T P_Gain = 0.74;
const real_T P_InErrMap[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 } ;

const real_T P_OutMap[7] = { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0 } ;

/*
 * File trailer for generated code.
 *
 * [EOF]
```

View the code algorithm in the model `step` function in the file `rtwdemo_PCG_Eval_P2.c`. The algorithm uses the data object names directly.

```
cfile = fullfile('rtwdemo_PCG_Eval_P2_ert_rtw','rtwdemo_PCG_Eval_P2.c');
rtwdemodbtype(cfile,'/* Model step function */',...
    '/* Sum: ''<S2>/Sum3'' incorporates:',1,0);
```

```
/* Model step function */
void rtwdemo_PCG_Eval_P2_step(void)
{
  real_T Discrete_Time_Integrator1;
  real_T Discrete_Time_Integrator1_i;
  real_T rtb_IntegralGainShape;

  /* Sum: '<S2>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_1'
   *  Inport: '<Root>/pos_rqst'
   */
  rtb_IntegralGainShape = *pos_rqst - fbk_1;

  /* DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' */
  if (rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_SYSTE != 0) {
    /* DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' */
    Discrete_Time_Integrator1 =
      rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSTAT;
  } else {
    /* DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' incorporates:
     *  Gain: '<S2>/Int Gain1'
     *  Lookup_n-D: '<S2>/Integral  Gain Shape'
     *  Lookup_n-D: '<S3>/Integral  Gain Shape'
     *  Product: '<S2>/Product3'
     */
    Discrete_Time_Integrator1 = I_Gain * look1_binlx(rtb_IntegralGainShape,
      &I_InErrMap[0], &I_OutMap[0], 8U) * rtb_IntegralGainShape * 0.001 +
      rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSTAT;
  }

  /* End of DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' */
```

Without storage classes and other code generation settings that the data objects specify, the generated code:

- Inlines invariant signals and block parameters when you select these optimizations in the model configuration parameters
- Places model input and output signals, block states, and tunable parameters in global data structures
- Does not create global variables that correspond to MATLAB variables

In contrast, the example code shows that most of the default data structures have been replaced with user-defined data objects. For example, these signals and parameters appear as global variables in the code:

- `pos_rqst`
- `fbk_1`
- `I_Gain`
- `I_OutMap`
- `I_InErrMap`

However, the local variable `rtb_IntegralGainShape` and the state variable `rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSTAT` still use code generator data structures. Data objects do not exist for these entities.

Store Model Data in Data Dictionary

When you end your MATLAB session, variables that you create in the base workspace do not persist. To permanently store data objects and bus objects, consider linking the model to a data dictionary.

1. In the example model, on the **Modeling** tab, under **Design**, click **Link to Data Dictionary**.
2. In the Model Properties dialog box, click **New**.
3. In the Create a New Data Dictionary dialog box, set **File name** to `rtwdemo_PCG_Eval_dict` and click **Save**.
4. In the Model Properties dialog box, click **Apply**.
5. Click **Migrate data**.
6. Click **Migrate** in response to message about copying referenced variables.
7. Click **OK**.

Alternatively, to manually migrate the objects into a data dictionary, you can use programmatic commands:

```
% Create a list of the variables and objects that the target
% model uses.
usedVars = {Simulink.findVars('rtwdemo_PCG_Eval_P2').Name};
% Create a new data dictionary in your current folder. Link the model to
% this new dictionary.
dictObj = Simulink.data.dictionary.create('rtwdemo_PCG_Eval_dict.sldd');
set_param('rtwdemo_PCG_Eval_P2','DataDictionary','rtwdemo_PCG_Eval_dict.sldd')
% Import only the target variables from the base workspace to the data
% dictionary.
importFromBaseWorkspace(dictObj,'clearWorkspaceVars',true,'varList',usedVars);
saveChanges(dictObj);
```

The data dictionary permanently stores the objects that the model uses. To view the contents of the dictionary, click the model data badge in the lower-left corner of the model, then click the **Data Dictionary** link. In the Model Explorer **Model Hierarchy** pane, select the **Design Data** node.

The separation of data from the model provides these benefits:

One model, multiple data sets

- Use of different data types to change the targeted hardware (for example, for floating-point and fixed-point targets)
- Use of different parameter values to change the behavior of the control algorithm (for example, for reusable components with different calibration values)

Multiple models, one data set

- Sharing of data between Simulink® models in a system
- Sharing of data between projects (for example, transmission, engine, and wheel controllers might all use the same CAN message data set)

See Also

More About

- “About Data Types in Simulink”
- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2
- “What Is a Data Dictionary?”

Partition Functions in Generated Code

This example shows how to associate subsystems in a model with function names and files.

Learn how to:

- Specify function and file names in the generated code.
- Identify the parts of the generated code that are required for integration.
- Generate code for atomic subsystems.
- Identify data that are required to execute a generated function.

For information about the example model and other examples in this series, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Atomic and Virtual Subsystems

The example models in “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29 use *virtual subsystems*. Virtual subsystems visually organize blocks but do not affect the model functionality. *Atomic subsystems* evaluate the blocks included in a model as a unit. With atomic subsystems, you can specify additional function partitioning information. In a model, atomic subsystems appear with a bold border.

View Changes in Model Architecture

Open the example model rtwdemo_PCG_Eval_P3.

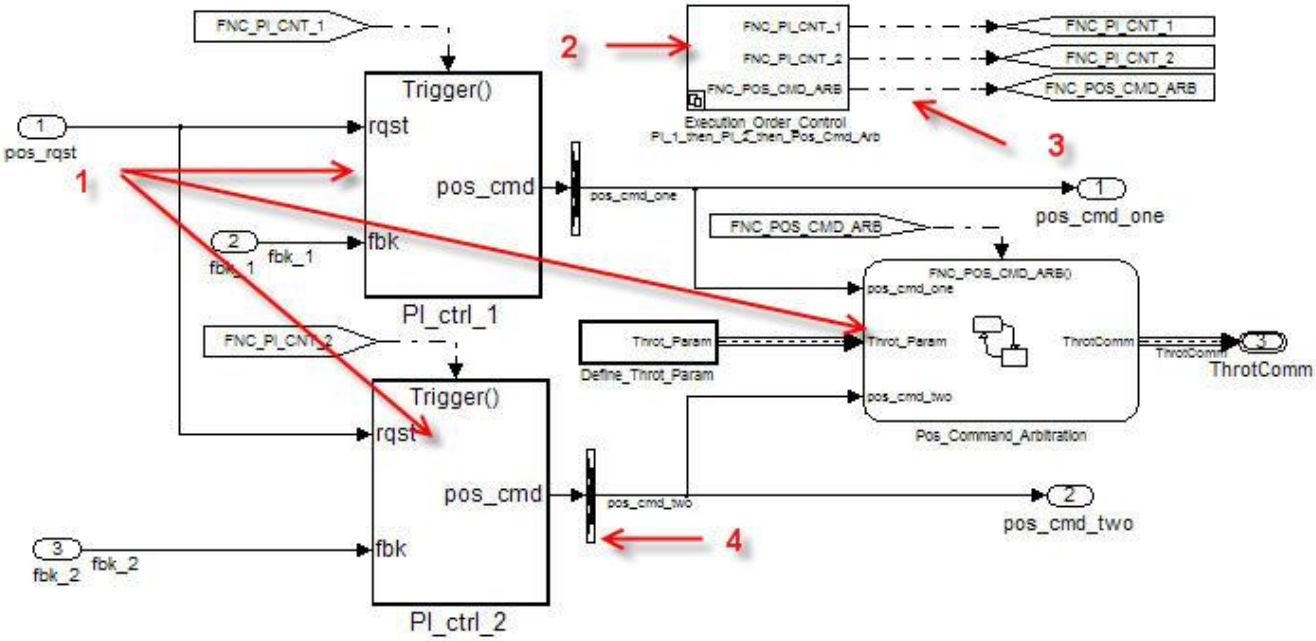
Save a copy of the model to a writable folder.

This example shows how to replace the virtual subsystems with *function-call subsystems*. Function-call subsystems:

- Are atomic subsystems
- Enable you to control subsystem execution order
- Execute when a *function call signal* triggers

By controlling the execution order of the subsystems, you can match the model with an existing system that has a specific execution order.

The figure identifies the function call subsystems (1) `PI_ctrl_1`, `PI_ctrl_2`, and `Pos_Command_Arbitration`.



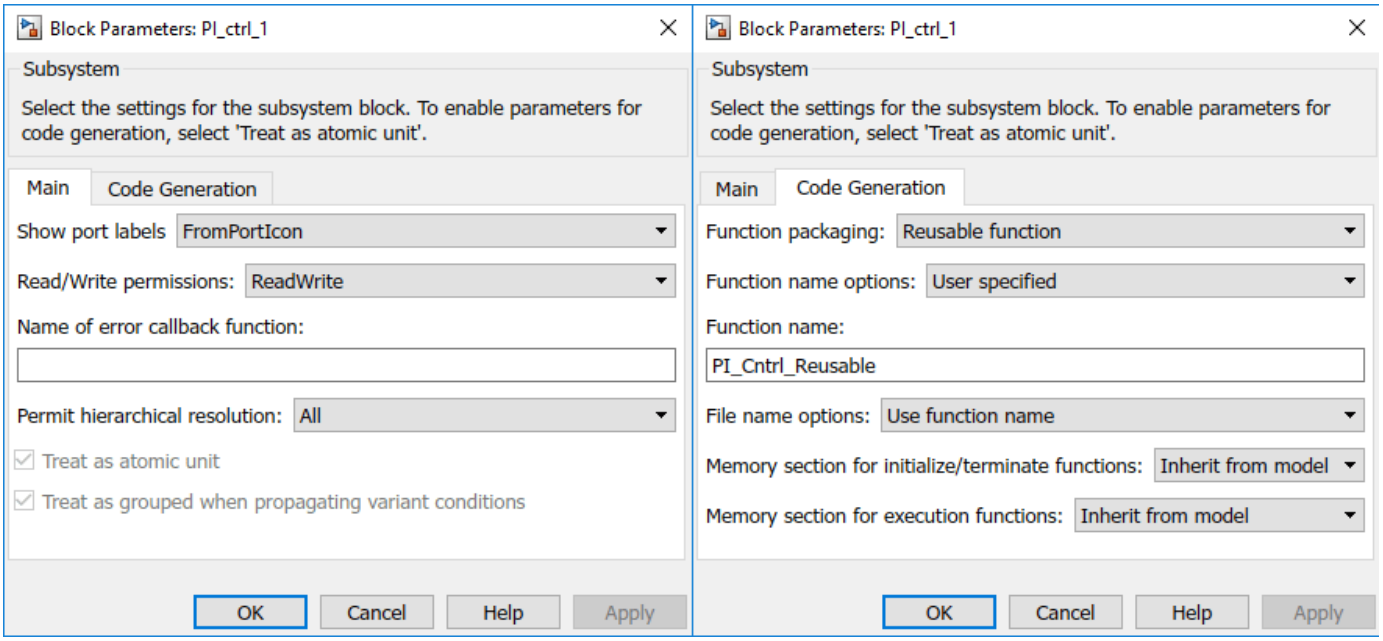
This version of the model contains the new subsystem `Execution_Order_Control` (2), which contains a Stateflow® chart that models the calling functionality of a scheduler. The subsystem controls the execution order of the function call subsystems through function call signals (3). Later in this example, you examine how changing the execution order can change the simulation results.

This version of the model contains new Signal Conversion blocks (4) at the outputs of the PI controllers. With these additional blocks in place, the code generator can generate a single reentrant function for the PI controllers.

Control Function Location and File Placement in the Generated Code

In “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29, the code generator creates a single *`model`*`_step` function that contains the control algorithm code. However, many applications require a greater level of control over the file placement of functions. By modifying the parameters of atomic subsystems, you can specify multiple functions within a single model.

The figure shows the subsystem parameters for `PI_ctrl_1`.



Treat as atomic unit

- Enables other submenus. For atomic subsystems, this parameter is automatically selected and disabled.

Sample time

- Specifies a sample time for execution. Not available for function-call subsystems.

Function packaging options

- `Auto` -- Determines how the subsystem appears in the generated code. This value is the default.
- `Inline` -- Places the subsystem code inline with the rest of the model code.
- `Function` -- Generates the code for the subsystem as a function.
- `Reusable function` -- Generates a reusable (reentrant) function from the subsystem. The function passes all input and output data through formal parameters. The function does not directly access global variables.

Function name options

- Selecting `Function` or `Reusable function` for **Function packaging** enables function name options.
- `Auto` -- Determines the function.
- `Use subsystem name` -- Bases the function on the subsystem name.
- `User specified` -- Applies the specified file name.

File name options

- Selecting `Function` or `Reusable function` for **Function packaging** enables file name options.
- `Auto` -- Places the function definition in the module generated for the parent system, or, if the model root is the parent, in `model.c`.

- `Use subsystem name` -- Generates a separate file. The name of the file is the name of the subsystem or library block.
- `Use function name` -- Generates a separate file. The name of the file is the name that you specify with **Function name options**.
- `User specified` -- Applies the specified, unique file name.

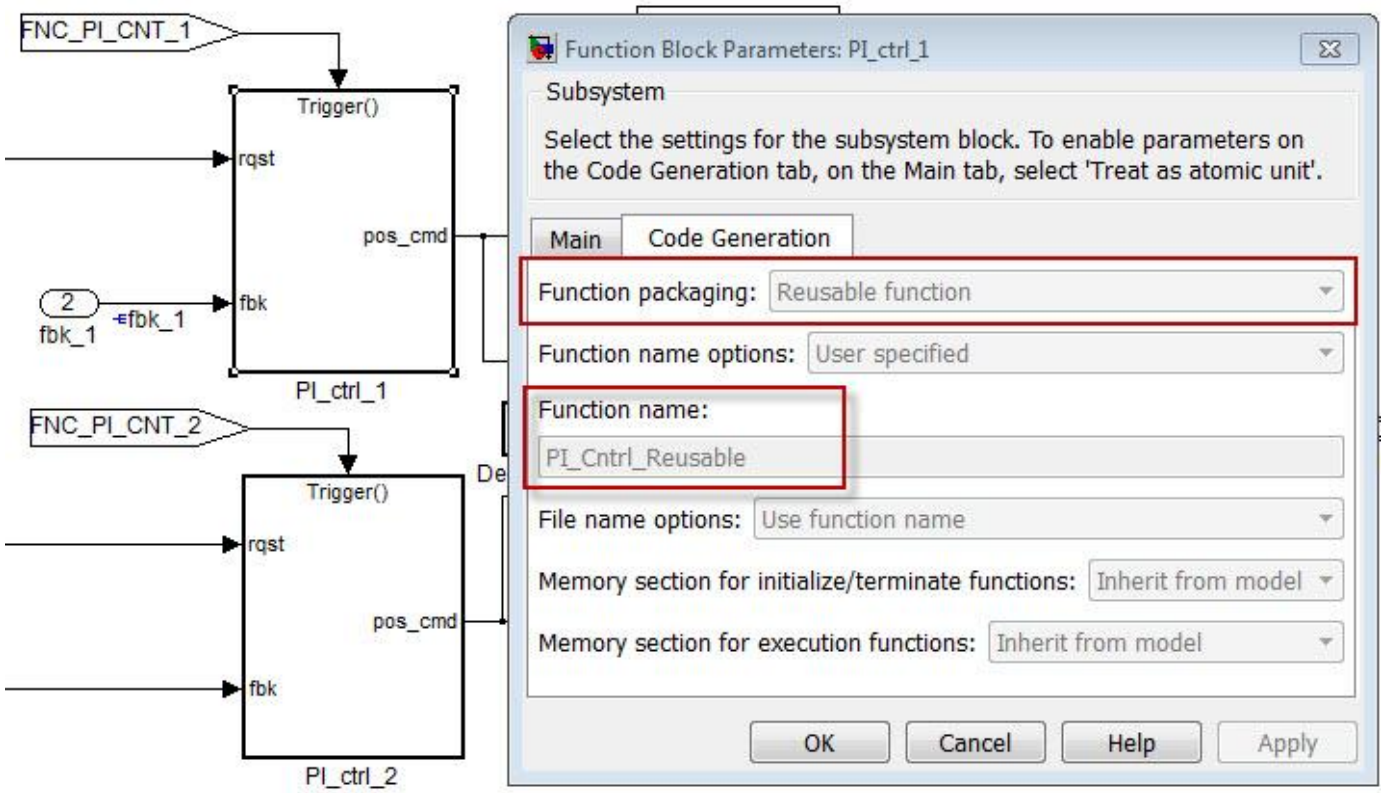
Function with separate data

- Enabled when you set **Function packaging** to `Function`. When selected, the code generator separates the internal data of the subsystem (for example, signals) from the data of the parent model. The subsystem owns this separate data.

Generate Reentrant Code

Embedded Coder® supports *reentrant code*. Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Reentrant code is used in operating systems and other system software that uses multithreading to handle concurrent events. Reentrant code does not maintain state data, so there are no persistent variables in the function. Calling programs maintain state variables and must pass the state data into the function. Multiple users or processes can share one copy of a reentrant function.

To generate reentrant code, you must first specify the subsystem as reusable by configuring the subsystem parameter **Function packaging**.



In some cases, the configuration of the model prevents reusable code. The table lists common issues.

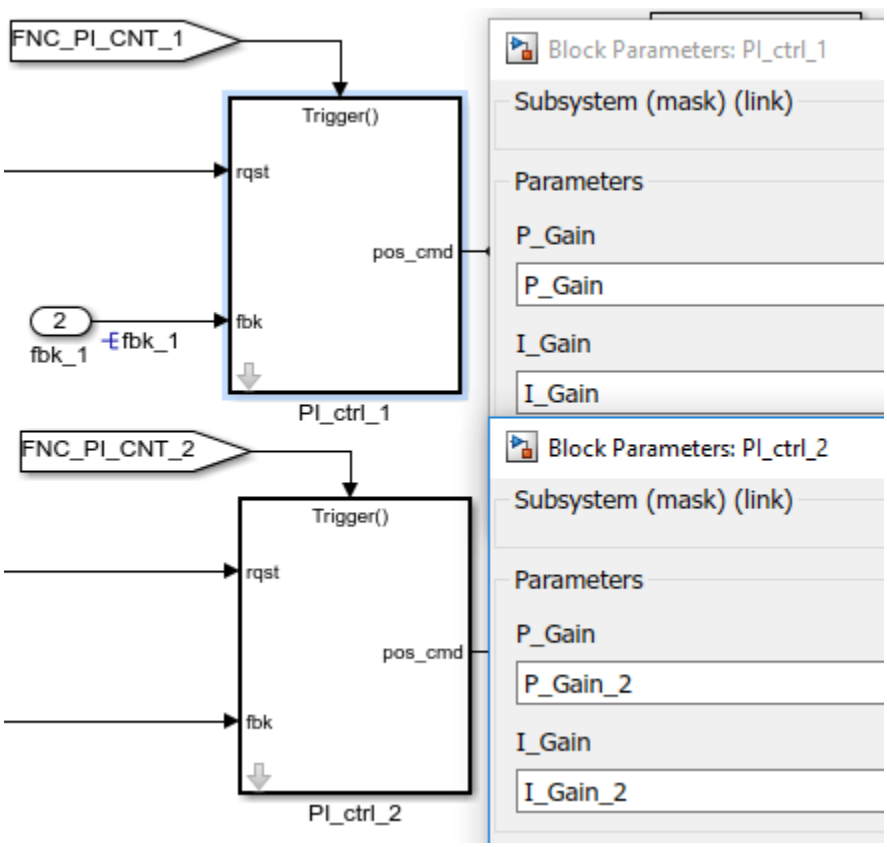
Cause	Solution

`Subsystem output feeds global signal data`	`Add a Signal Conversion block between the subsystem and the global signal.`
`Generated function receives data (formal parameters) through pointers`	`Select Configuration Parameters > Model Referencing > Pass fixed-size scalar root inputs by value for code generation.`
`Subsystem uses global signal data in internal algorithm`	`Use a port to pass the global data in and out of the subsystem.`

Use a Mask to Pass Parameter Values into Library Subsystem

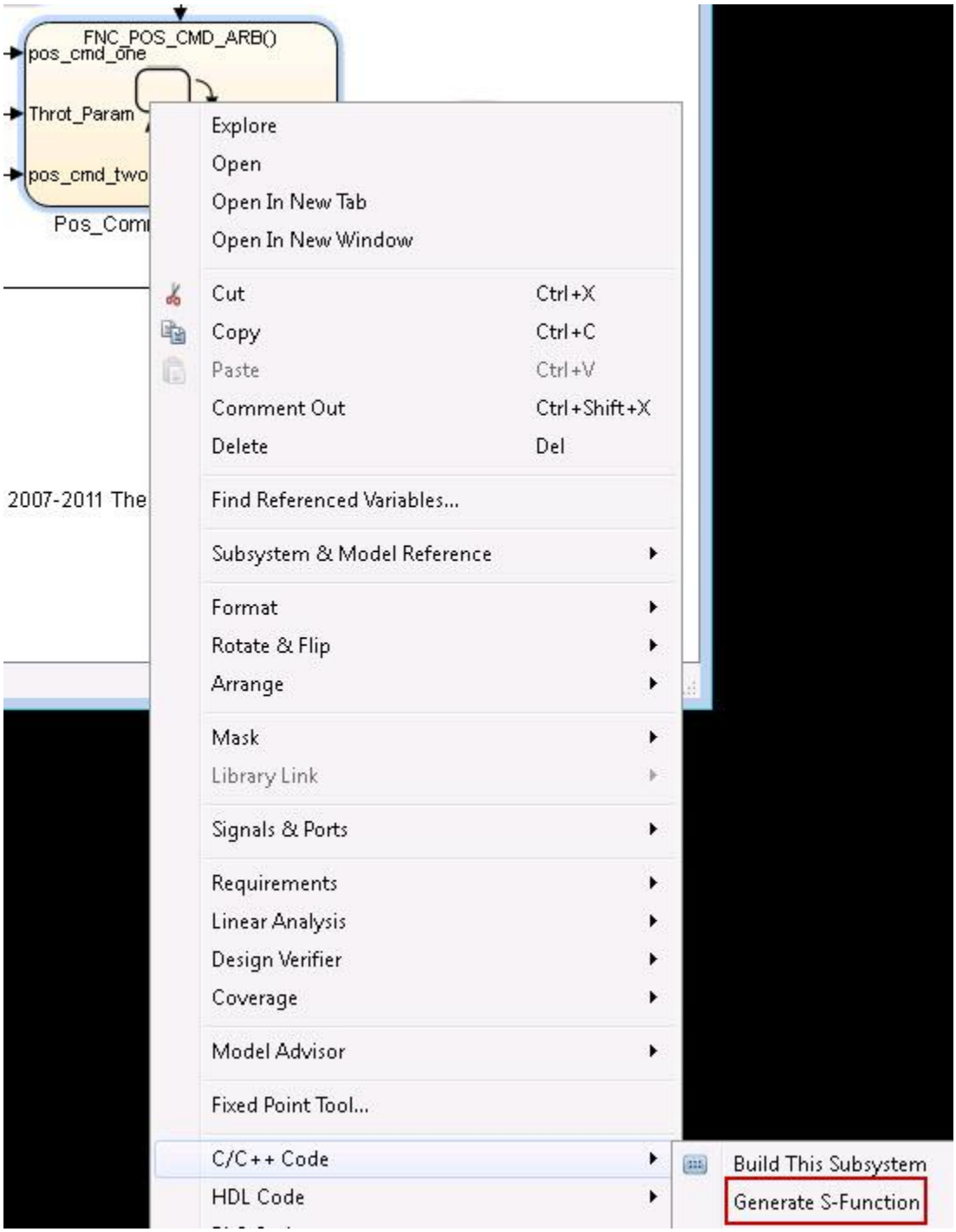
To define algorithmic parameter data (such as a gain or coefficient) outside the scope of a reusable library block or subsystem, you can apply a *mask* to the block or subsystem and create a mask parameter. You can then specify a different parameter value for each instance of the block or subsystem. Each mask parameter appears in the generated code as a formal parameter of the reentrant function.

In this version of the model, the subsystems `PI_ctrl_1` and `PI_ctrl_2` are masked. In each mask, the values of the `P` and `I` gains are set by data objects such as `I_Gain_2` and `P_Gain_2`.



Generate Code for Atomic Subsystem

In “Prepare a Control Algorithm Model for C Code Generation” on page 53-18 and “Configure Data Interface in the Generated Code” on page 53-29, you generate code at the root level of the model. Alternatively, you can build a specific subsystem.



To initiate a subsystem build, use the context menu. You can choose from these options:

1. **Build This Subsystem**: Treats the subsystem as a separate mode and creates the full set of source C files and header files. This option does not support function-call subsystems.
2. **Generate S-Function**: Generates C code for the subsystem and creates an S-Function wrapper. You can then simulate the code in the original model. This option does not support function-call subsystems.

3. **Export Functions**: Generates C code without the scheduling code that comes with the **Build This Subsystem** option. Use this option to build subsystems that use triggers, such as function-call subsystems.

Alternatively, open the **Embedded Coder** app, select the subsystem and on the **C Code** tab, click **Build**.

Examine Generated Code

This example compares the files that are generated for the full system build with the files that are generated for exported functions. You also examine how the masked data appears in the code.

Run the build script for the three options. Then, examine the generated files by clicking the hyperlinks.

1. Generate code from the entire model.
2. Export the function `PI_ctrl_1`.
3. Export the function `Pos_Command_Arbitration`.

rtwdemo_PCG_Eval_P3.c

- Full Build: Yes, Step function
- PI_ctrl_1: No
- Pos_Command_Arbitration: No

PI_ctrl_1.c

- Full Build: No
- PI_ctrl_1: Yes, Trigger function
- Pos_Command_Arbitration: No

Pos_Command_Arbitration.c

- Full Build: No
- PI_ctrl_1: No
- Pos_Command_Arbitration: Yes, Init and Function

PI_Ctrl_Reusable.c

- Full Build: Yes
- PI_ctrl_1: Yes
- Pos_Command_Arbitration: No

ert_main.c

- Full Build: Yes
- PI_ctrl_1: Yes
- Pos_Command_Arbitration: Yes

eval_data.c

- Full Build: Yes(1)
- PI_ctrl_1: Yes(1)
- Pos_Command_Arbitration: No, Eval data not used in diagram

(1) `eval_data.c` has different content in the full and export function builds. The full build includes all of the parameters that the model uses. The export function contains only the variables that the subsystem uses.

Masked Data in the Generated Code

In the file `rtwdemo_PCG_Eval_P3.c`, the call sites of the reentrant function use the data objects `P_Gain`, `I_Gain`, `P_Gain_2`, and `I_Gain_2` as arguments.

```
PI_Cntrl_Reusable(*pos_rqst, fbk_1, &rtwdemo_PCG_Eval_P3_B->PI_ctrl_1,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl_1, I_Gain, P_Gain);
PI_Cntrl_Reusable(*pos_rqst, fbk_2, &rtwdemo_PCG_Eval_P3_B->PI_ctrl_2,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl_2, I_Gain_2, P_Gain_2);
```

Effect of Execution Order on Simulation Results

By default, Simulink® executes the subsystems in this order:

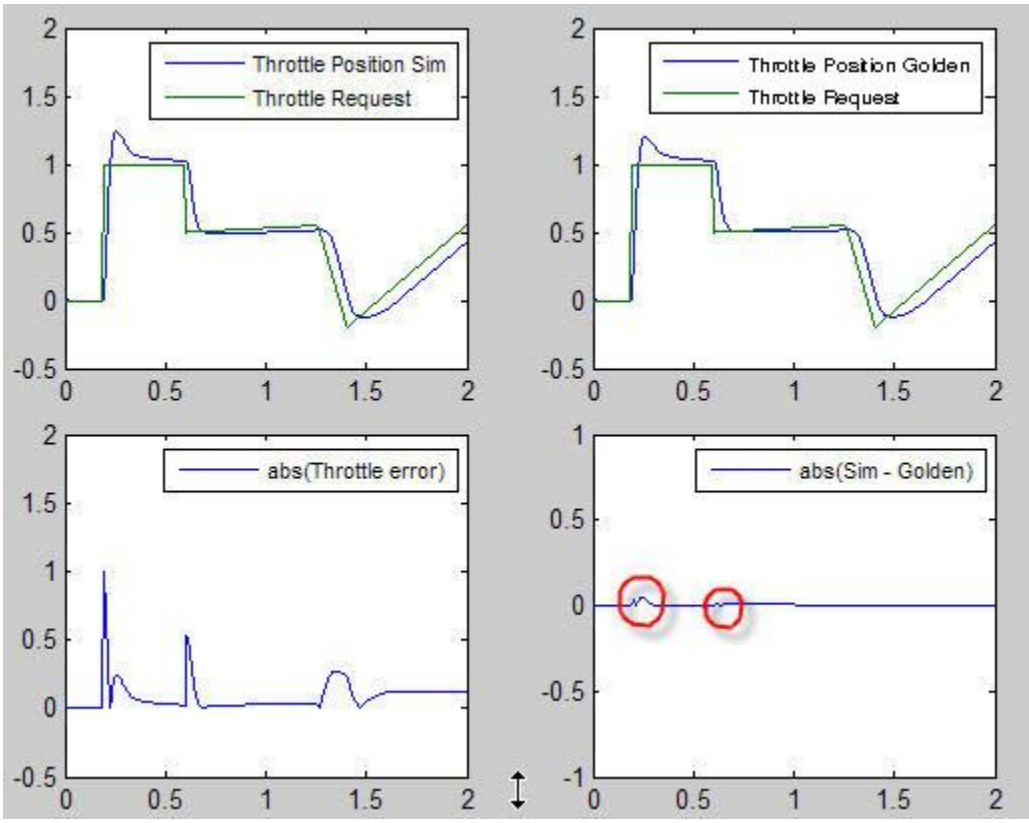
1. `PI_ctrl_1`
2. `PI_ctrl_2`
3. `Pos_Command_Arbitration`

For this example, you can specify one of two alternative orders of execution. You can then use the test harness to observe the effect of the execution order on the simulation results. The subsystem `Execution_Order_Control` has two configurations that control the execution order. To choose a configuration, use the subsystem context menu.

Change the execution order and observe the results.

1. Set the execution order to `PI_ctrl_1`, `PI_ctrl_2`, `Pos_Command_Arbitration`.
2. Open the test harness.
3. Run the test harness.
4. Change the execution order to `Pos_Command_Arbitration`, `PI_ctrl_1`, `PI_ctrl_2`.
5. Run the test harness.

The simulation results (throttle position over time) vary slightly depending on the order of execution. You can see the difference most clearly when the throttle request changes.



For the next example in this series, see “Call External C Code from Model and Generated Code” on page 53-48.

See Also

More About

- “Generate Code and Executables for Individual Subsystems” on page 4-2
- “S-Functions and Code Generation” on page 13-2
- “Generate Component Source Code for Export to External Code Base” on page 58-40
- “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114
- “Masking Fundamentals”

Call External C Code from Model and Generated Code

Call existing, external functions from a simulation or from the generated code by using the Legacy Code Tool.

Learn how to:

- Evaluate a C function as part of a Simulink® model simulation.
- Call a C function from the code that you generate from a model.

For information about the example model, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Replacement Process

Open the example model, `rtwdemo_PCG_Eval_P4`.

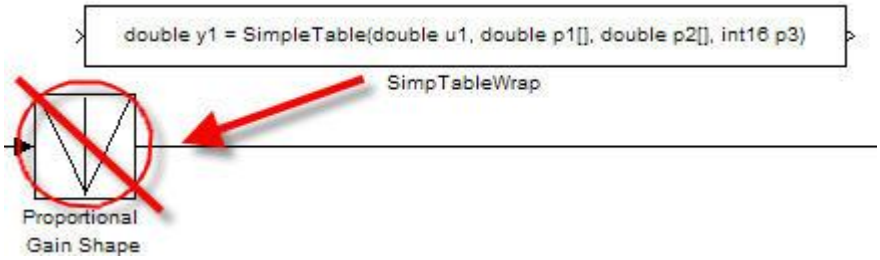
For many Model-Based Design applications, in addition to Simulink® models, a design includes a set of existing C functions that have been tested and validated. You can integrate these functions into a Simulink® model and generate code that uses the functions.

In this example, you create a custom Simulink® block that calls an existing C function. You then include the block in a model and test the overall system through model simulation in Simulink®.

In the example model, you can replace the Lookup blocks (lookup tables) in the PI controllers with calls to an existing C function. The function is defined in files `SimpleTable.c` and `SimpleTable.h`.

View `SimpleTable.c`.

View `SimpleTable.h`.



Create Block That Calls C Function

To specify a call to an existing C function, use an S-Function block. You can automate the creation of the S-Function block by using the Legacy Code Tool. In the tool, you first specify an interface for your existing C function. The tool then uses that interface to create an S-Function block.

Use the Legacy Code Tool to create an S-Function block for the existing C function in `SimpleTable.c`.

1. Create a structure to contain the definition of the function interface.

```
def = legacy_code('initialize')
```

You can use the structure `def` to define the function interface to the existing C code.

2. Populate the fields of the structure.

```
% Defines the function interface, inputs, return values and parameters
def.OutputFcnSpec = ['double y1 = SimpleTable(double u1,',...
                     'double p1[], double p2[], int16 p3)'];

% the C and H files where the external code is defined
def.HeaderFiles   = {'SimpleTable.h'};
def.SourceFiles   = {'SimpleTable.c'};
% The name of the created S-Function
def.SFunctionName = 'SimpTableWrap';
```

3. Create the S-function.

```
legacy_code('sfcn_cmex_generate',def)
```

4. Compile the S-function.

```
legacy_code('compile',def)
```

5. Create the S-Function block.

```
legacy_code('slblock_generate',def)
```

The generated S-Function block calls the C function in `SimpleTable.c`. You can now use this S-Function block in models.

6. Create the TLC file.

```
legacy_code('sfcn_tlc_generate',def)
```

This command creates a TLC file, which is the component of an S-Function that specifies how to generate code for the block.

Validate External Code Through Simulation

When you integrate existing C code in a Simulink® model, validate the generated S-Function block.

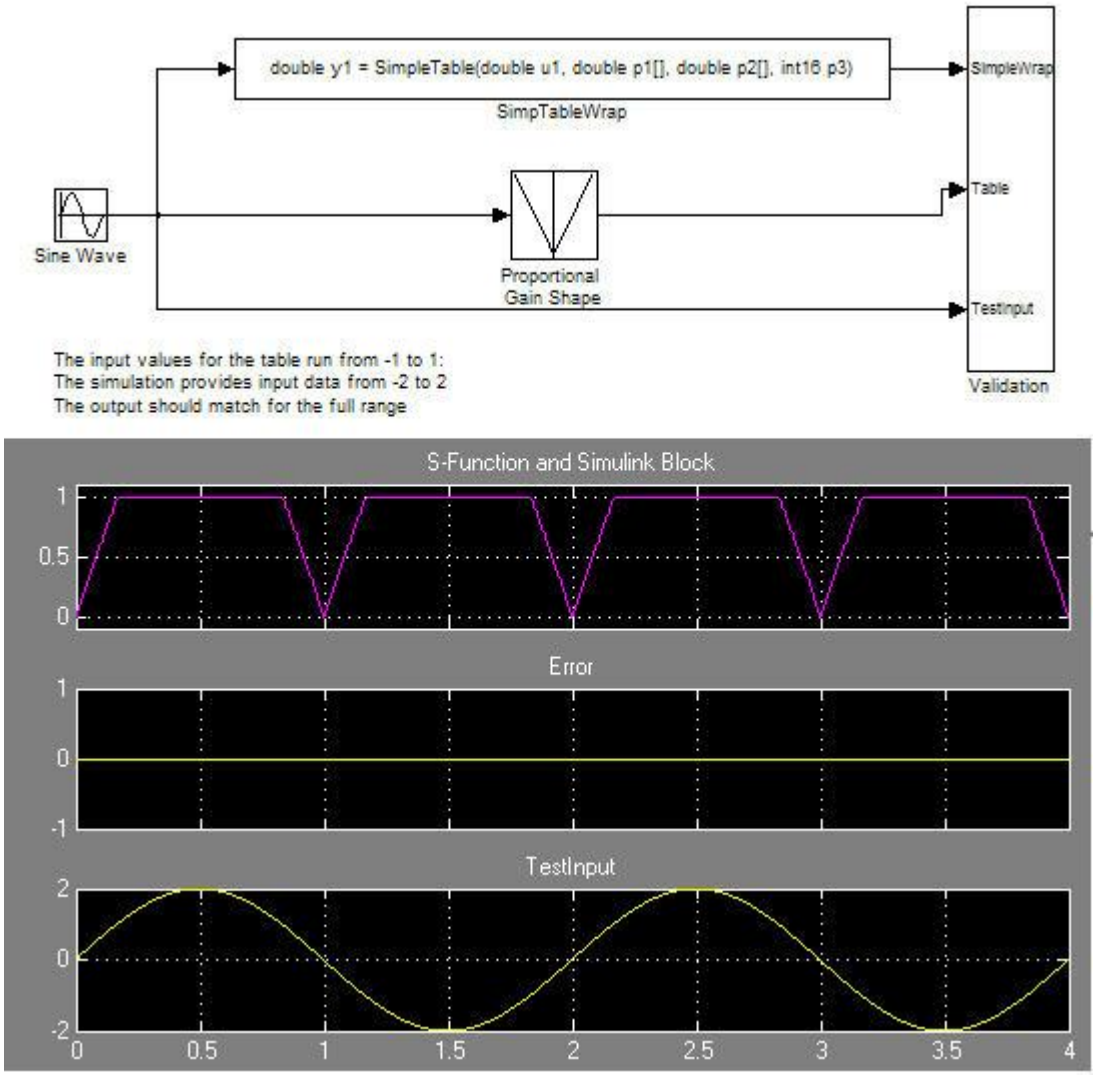
To validate the replacement of the Lookup blocks, compare the simulation results produced by the Lookup blocks with the results produced by the new S-Function block.

1. Open the validation model.

- The Sine Wave block produces output values from [-2 : 2].
- The input range of the lookup table is from [-1 : 1].
- The lookup table outputs the absolute value of the input.
- The lookup table clips the output at the input limits.

2. Run the validation model.

The figure shows the validation results. The existing C code and the Simulink® table block produce the same output values.



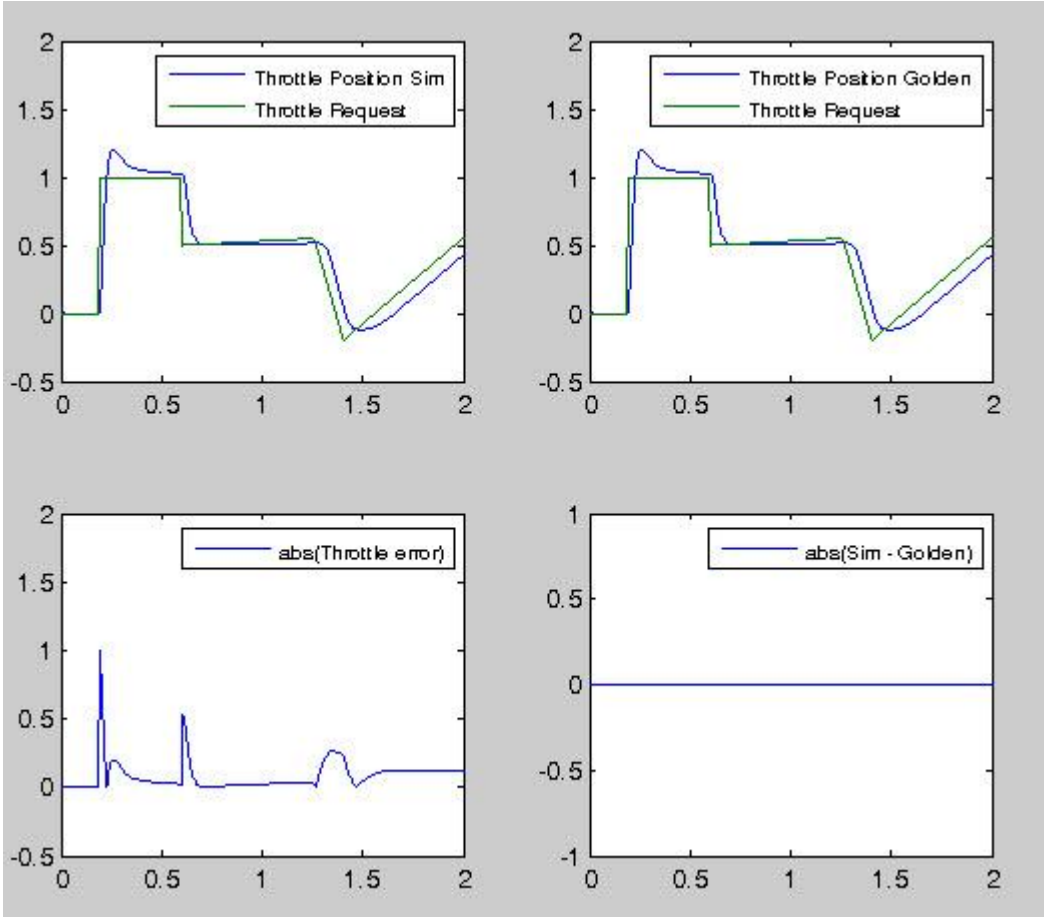
Validate C Code as Part of Simulink® Model

After validating the existing C function code as a standalone component, validate the S-function in the model. To complete the validation, use a test harness model.

1. Open the test harness.

2. Run the test harness.

The simulation results match the golden values.



Call C Function from the Generated Code

The code generator uses the TLC file to process the S-Function block like any other block. The code generator can implement *expression folding* with the S-Function block, an operation that combines multiple computations into a single output calculation.

1. Build the full model.

2. Examine the generated code in `PI_Control_Reusable.c`.

The generated code now calls the `SimpleTable` C function.

The figures show the generated code before and after the C code integration. Before the integration, the code calls a generated lookup routine. After the integration, the generated code calls the C function `SimpleTable`.

```
Discrete_Time_Integrator1 = rtp_Masked_I_Gain * look1_binlx(rtb_Sum2,
  (&(I_InErrMap[0])), (&(I_OutMap[0])), 8U) * rtb_Sum2 * 0.001 +
  localDW->Discrete_Time_Integrator1_DSTAT;
```

```
/* DiscreteIntegrator: '<S3>/Discrete_Time_Integrator1' incorporates:
 *  Gain: '<S3>/Int_Gain1'
 *  Product: '<S3>/Product3'
 *  S-Function (SimpTableWrap): '<S3>/SimpTableWrap1'
 */
Discrete_Time_Integrator1 = I_Gain * SimpleTable(rtb_Sum2, (real_T*)
  (&(I_InErrMap[0])), (real_T*)(&(I_OutMap[0])), 7) * rtb_Sum2 * 0.001 +
  localDW->Discrete_Time_Integrator1_DSTAT;
```

See Also

More About

- “Integrate C Functions Using Legacy Code Tool”
- “S-Functions and Code Generation” on page 13-2

Build Integrated Code Outside the Simulink Environment

Identify required files and interfaces for calling generated code in an external build process.

Learn how to:

- Collect files required for building integrated code outside of Simulink®.
- Interface with external variables and functions.

For information about the example model and related examples, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Collect and Build Required Data and Files

The code that Embedded Coder® generates requires support files that MathWorks® provides. To relocate the generated code to another development environment, such as a dedicated build system, you must relocate these support files. You can package these files in a zip file by using the `packNGo` utility. This utility finds and packages the files that you need to build an executable image. The utility uses tools for customizing the build process after code generation, which include a `buildinfo_data` structure, and a `packNGo` function. These files include external files that you identify in the **Code Generation > Custom Code** pane in the Model Configuration Parameters dialog box. The utility saves the `buildinfo` MAT-file in the *`model`*`_ert_rtw` folder.

Open the example model, `rtwdemo_PCG_Eval_P5`.

This model is configured to run `packNGo` after code generation.

Generate code from the entire model.

To generate the zip file manually:

1. Load the file `buildInfo.mat` (located in the `rtwdemo_PCG_Eval_P5_ert_rtw` subfolder).
2. At the command prompt, enter the command `packNGo(buildInfo)`.

The number of files in the zip file depends on the version of Embedded Coder and on the configuration of the model that you use. The compiler might require a subset of the files in the zip file. The compiled executable size (RAM/ROM) depends on the linking process. The linker likely includes only the object files that are necessary.

Integrating the Generated Code into an Existing System

This example shows how to integrate the generated code into an existing code base. The example uses the Eclipse™ IDE and the Cygwin™/gcc compiler. The required integration tasks are common to integration environments.

Overview of Integration Environment

A full embedded controls system consists of multiple hardware and software components. Control algorithms are just one type of component. Other components can be:

- An operating system (OS)
- A scheduling layer

- Physical hardware I/O
- Low-level hardware device drivers

Typically, you do not use the generated code in these components. Instead, the generated code includes interfaces that connect with these components. MathWorks provides hardware interface block libraries for many common embedded controllers. For examples, see the Embedded Targets block library.

This example provides files to show how you can build a full system. The main file is `example_main.c`, which contains a simple main function that performs only basic actions to exercise the code.

View `example_main.c`.

```
int_T main(void)
{
  /* Initialize model */
  rt_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
  rtwdemo_PCG__Define_Throt_Param(); /* SubSystem: '<Root>/Define_Throt_Param' */
  defineImportData();                /* Defines the memory and values of inputs */

  do /* This is the "Schedule" loop.
      Functions would be called based on a scheduling algorithm */
  {
    /* HARDWARE I/O */

    /* Call control algorithms */
    PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
    PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
    pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
    pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

    rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                    &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

The file:

- Defines function interfaces (function prototypes).
- Includes files that declare external data.
- Defines `extern` data.
- Initializes data.
- Calls simulated hardware.
- Calls algorithmic functions.

The order of function execution matches the order of subsystem execution in the test harness model and in `rtwdemo_PCG_Eval_P5.h`. If you change the order of execution in `example_main.c`, results that the executable image produces differ from simulation results.

Match System Interfaces

Integration requires matching the *Data* and *Function* interfaces of the generated code and the existing system code. In this example, the `example_main.c` file imports and exports the data through `#include` statements and `extern` declarations. The file also calls the functions from the generated code.

Connect Input Data

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The generated code accesses the two feedback signals through direct reference to imported global variables (storage class `ImportedExtern`). The code accesses the position signal through an imported pointer (storage class `ImportedExternPointer`).

The handwritten file `defineImportedData.c` defines the variables and the pointer. The generated code does not define the variables and the pointer because the handwritten code defines them. Instead, the generated code declares the imported data (`extern`) in the file `rtwdemo_PCG_Eval_P5_Private.h`. In a real system, the data typically comes from other software components or from hardware devices.

View `defineImportedData.c`.

```
/* Define imported data */
#include "rtwtypes.h"
#include "defineImportedData.h"

real_T fbk_1;
real_T fbk_2;
real_T dummy_pos_value = 10.0;
real_T *pos_rqst;
void defineImportData(void)
{
    pos_rqst = &dummy_pos_value;
}
```

View `rtwdemo_PCG_Eval_P5_Private.h`.

```
/* Imported (extern) block signals */
extern real_T fbk_1;                    /* '<Root>/fbk_1' */
extern real_T fbk_2;                    /* '<Root>/fbk_2' */

/* Imported (extern) pointer block signals */
extern real_T *pos_rqst;                /* '<Root>/pos_rqst' */
```

Connect Output Data

In this example, you do not access the output data of the system. The example “Test Generated Code” on page 53-58 shows how you can save the output data to a standard log file. You can access the output data by referring to the file `rtwdemo_PCG_Eval_P5.h`.

View `rtwdemo_PCG_Eval_P5.h`.

Access Additional Data

The generated code contains several structures that store commonly used data including:

- Block state values (integrator, transfer functions)
- Local parameters
- Time

The table lists the common data structures. Depending on the configuration of the model, a combination of these structures appear in the generated code. The data is declared in the file `rtwdemo_PCG_Eval_P5.h`, but in this example, you do not access this data.

```
Data Type      Data Name     Data Purpose

Constants      |model_cP|    Constant parameters
Constants      |model_cB|    Constant block I/O
Output         |model_U|     Root and atomic subsystem input
Output         |model_Y|     Root and atomic subsystem output
Internal data  |model_B|     Value of block output
Internal data  |model_D|     State information vectors
Internal data  |model_M|     Time and other system level data
Internal data  |model_Zero|  Zero-crossings
Parameters     |model_P|     Parameters
```

Match Function Call Interfaces

By default, functions that the code generator generates have a `void Func(void)` interface. If you configure the model or atomic subsystem to generate reentrant code, the code generator creates a more complex function prototype. In this example, the `example_main` function calls the generated functions with valid input arguments.

```
PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

Calls to the function `PI_Cntrl_Reusable` use a mixture of separate, unstructured global variables and Simulink Coder™ data structures. The handwritten code defines these variables. The structure types are defined in `rtwdemo_PCG_Eval_P5.h`.

Build Project in Eclipse Environment

This example uses the Eclipse IDE and the Cygwin GCC debugger to build the embedded system. The example provides installation files for both programs. Software components and versions numbers are:

- Eclipse SDK 3.2
- Eclipse CDT 3.3

- Cygwin/GCC 3.4.4-1
- Cygwin/GDB 20060706-2

To install and use Eclipse and GCC, see “Install and Use Cygwin and Eclipse” on page 53-68.

You can install the files for this example by clicking this hyperlink:

Set up the build folder.

Alternatively, to install the files manually:

1. Create a build folder (`Eclipse_Build_P5`).
2. Unzip the file `rtwdemo_PCG_Eval_P5.zip` into the build folder.
3. Delete the files `rtwdemo_PCG_Eval_P5.c`, `ert_main.c` and `rt_logging.c`, which are replaced by `example_main.c`.

You can use the Eclipse debugger to step through and evaluate the execution behavior of the generated C code. See the example “Install and Use Cygwin and Eclipse” on page 53-68.

To exercise the model with input data, see “Test Generated Code” on page 53-58.

Related Topics

- “Generate Component Source Code for Export to External Code Base” on page 58-40
- “Generate Shared Library for Export to External Code Base” on page 58-54

See Also

More About

- “Code Compilation Customization”
- “Customize Post-Code-Generation Build Processing” on page 93-7
- “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17
- “Relocate or Share Generated Code” on page 59-56

Test Generated Code

Validate generated code by using system-level S-functions or by running the code in an external environment.

Learn about:

- Different methods for testing the generated code
- How to test generated code in Simulink®
- How to test generated code outside of Simulink

For information about the example model and other examples in this series, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Validation Methods

Simulink supports multiple system testing methods for validating the behavior of generated code.

Windows® run-time executable: Generate a Microsoft® Windows executable and run the executable from the command prompt.

- Benefit: Easy to create and enables use of a C debugger to evaluate code
- Drawback: Partial emulation of target hardware

Software-in-the-loop (SIL) testing: Use an S-function to include the generated code in the Simulink model.

- Benefit: Easy to create, enables reuse of the Simulink test environment, and enables use of a C debugger to evaluate code
- Drawback: Partial emulation of target hardware

Processor in the loop (PIL) testing: Run a non-real-time cosimulation. Simulink executes a portion of the model (typically a plant model) while the target processor runs a different portion of the model (typically a controller). After you download code to the target processor, the processor-in-the-loop technology handles the communication of signals between Simulink and the target during the cosimulation.

- Benefit: Enables reuse of the Simulink test environment, enables use of a C debugger with the simulation, and involves the target processor
- Drawback: Requires additional steps to set up the test environment and the processor does not run in real time

On-target rapid prototyping: Run the generated code on the target processor as part of the full system.

- Benefit: You can determine actual hardware constraints and test the component within the full system. The processor runs in real time
- Drawback: Requires hardware and requires additional steps to set up the test environment

External Mode: Run the generated code on the target processor as part of the full system.

- Benefit: You can determine actual hardware constraints and test the component within the full system
- Drawback: Requires hardware and requires additional steps to set up the test environment

Reuse Test Data Outside of Simulink

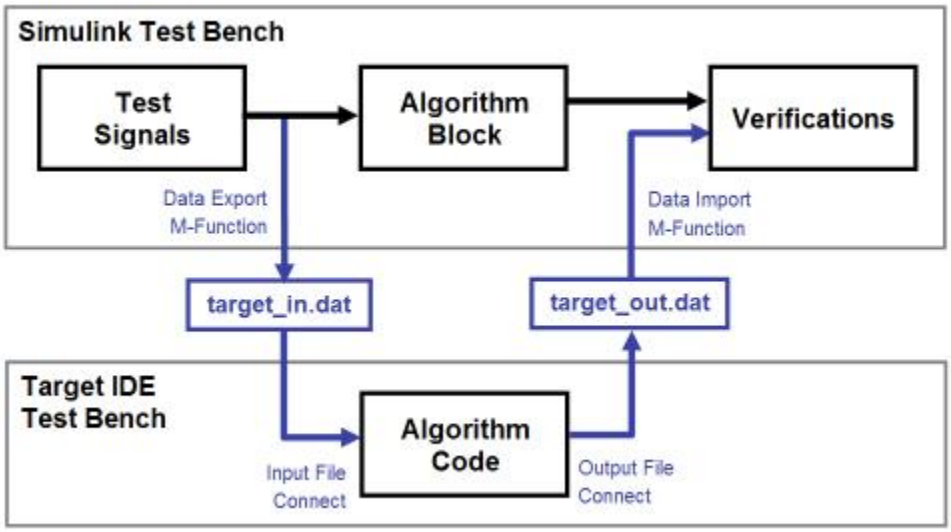
In other examples in this series, you test the system through simulation in Simulink. The test input data come from a Signal Editor block in the test harness model.

To test the system outside of Simulink, you can reuse the test input data.

1. Save the Simulink test data into a file.
2. Format the data in a way that the system code can access.
3. Read the data file as part of the system code procedures.

To use Simulink to perform verification, you can save the test output data from the external environment in a format that MATLAB® can read.

In this example, the file `hardwareInputs.c` contains the data from the Signal Editor block in the test harness model.



Test via Software-in-the-Loop (Model Block SIL)

Create and Configure Model Block for SIL

Simulink can generate code from a Model block, wrap the code into an S-Function, and add the S-Function back into the model for software-in-the-loop testing.

Open the example model, `rtwdemo_PCG_Eval_P6`.

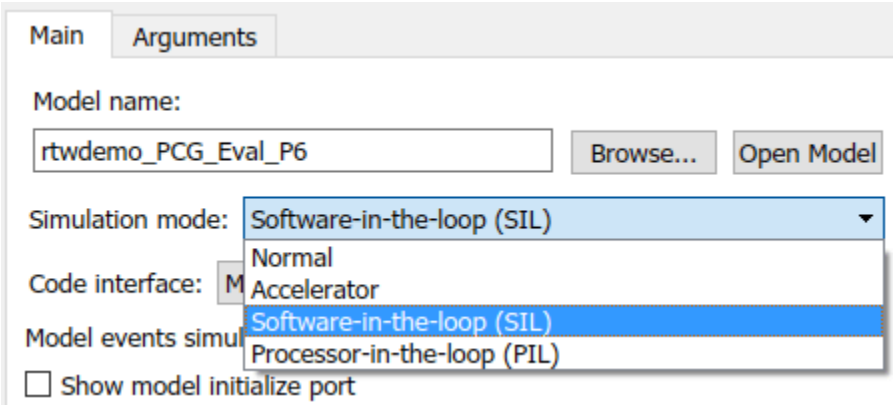
Open the test harness model.

The test harness uses a Model block to access the example model. In this example, you run a software-in-the-loop test on the example model.

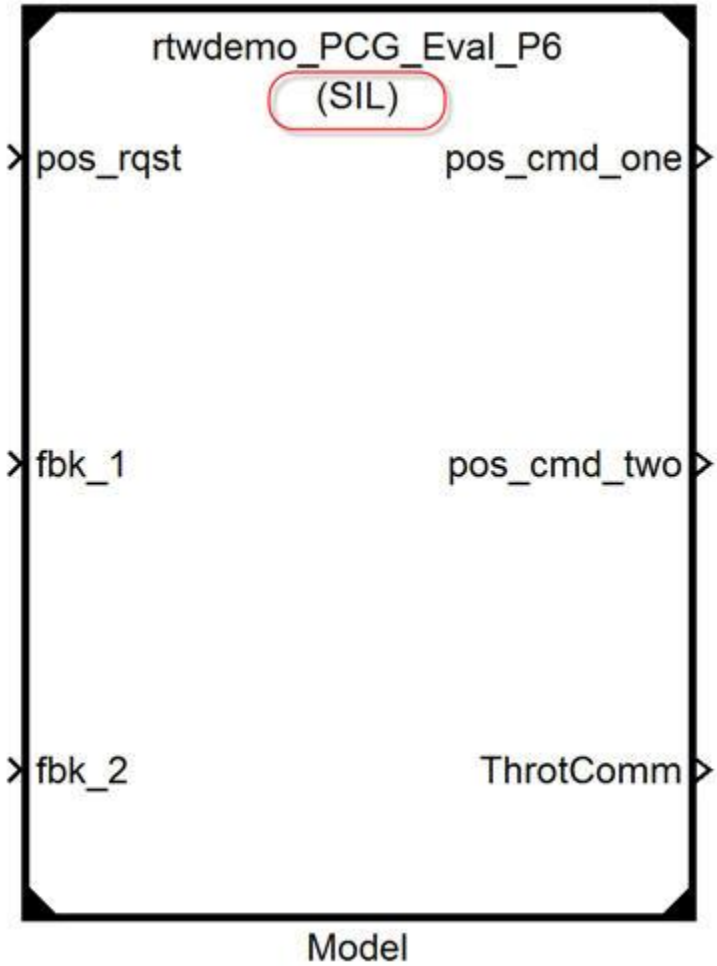
1. Right-click the Model block and select **Block Parameters (ModelReference)**.

2 Set **Model name** to the name of the model that you want to test.

3 Set **Simulation mode** to `Software-in-the-loop (SIL)`.



The Model block now shows a `(SIL)` tag.



Configure Models for SIL

Configure the example model, `rtwdemo_PCG_Eval_P6`, and the test harness model, `rtwdemo_PCGEvalHarnessHTGTSIL`, for SIL simulation by adjusting the hardware implementation parameters.

Open the example model.

For both models, in the Configuration Parameters dialog box, select the **Support long long** check box.

Save a copy of the example model, `rtwdemo_PCG_Eval_P6`, in your current folder.

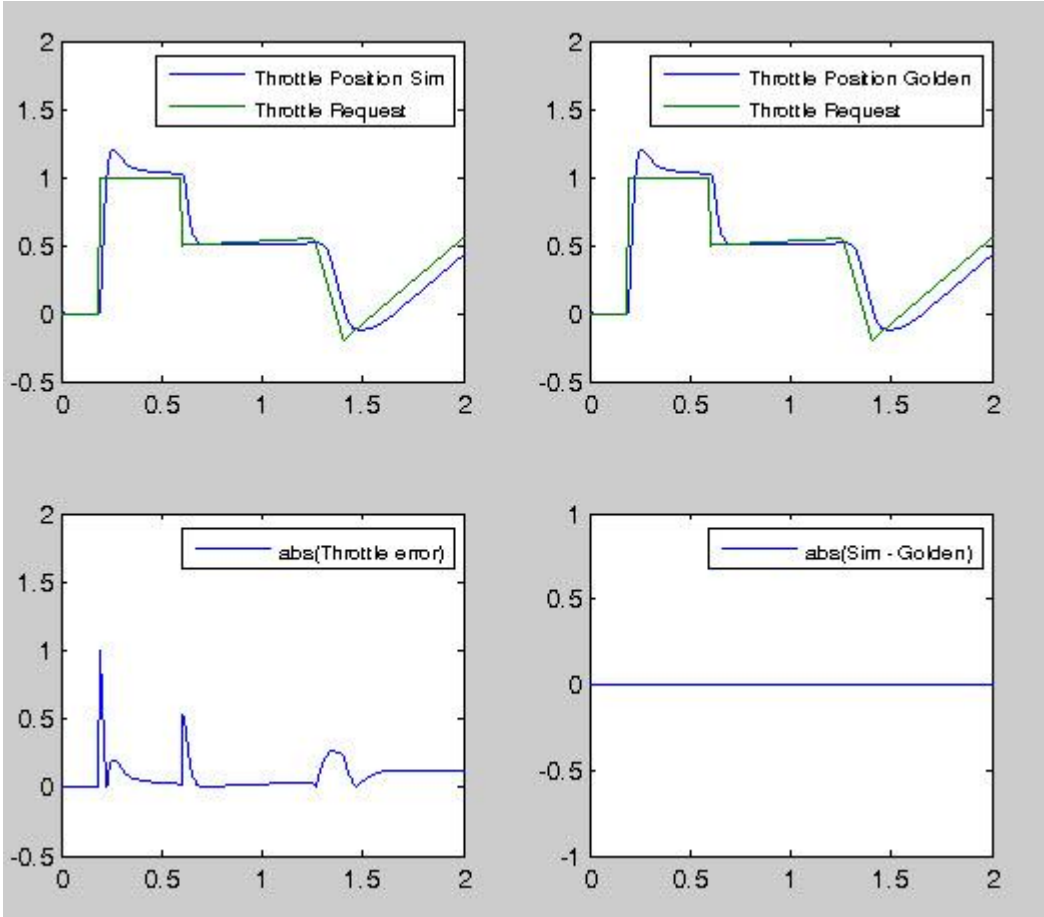
Run Model Block SIL

In this example, the test harness model contains a Model block that is configured for SIL simulation.

Open the test harness.

Run the test harness.

The generated code and the simulation yield the same results.



Configure System for Testing via Test Vector Import and Export

This example extends the integration example in “Build Integrated Code Outside the Simulink Environment” on page 53-53. In this case, `example_main.c` uses simulated hardware I/O.

The augmented `example_main.c` file now uses this execution order:

1. Initialize data (one time)

`while < endTime`

2. Read the simulated hardware inputs

3. `PI_cnrl_1`

4. `PI_ctrl_2`

5. `Pos_Command_Arbitration`

6. Write the simulated hardware outputs

`end while`

View `example_main.c`.

Two functions, `plant` and `hardwareInputs`, supply the input test data.

`Plant.c`: This function, which was generated from the plant portion of the test harness, simulates the throttle body response to throttle commands.

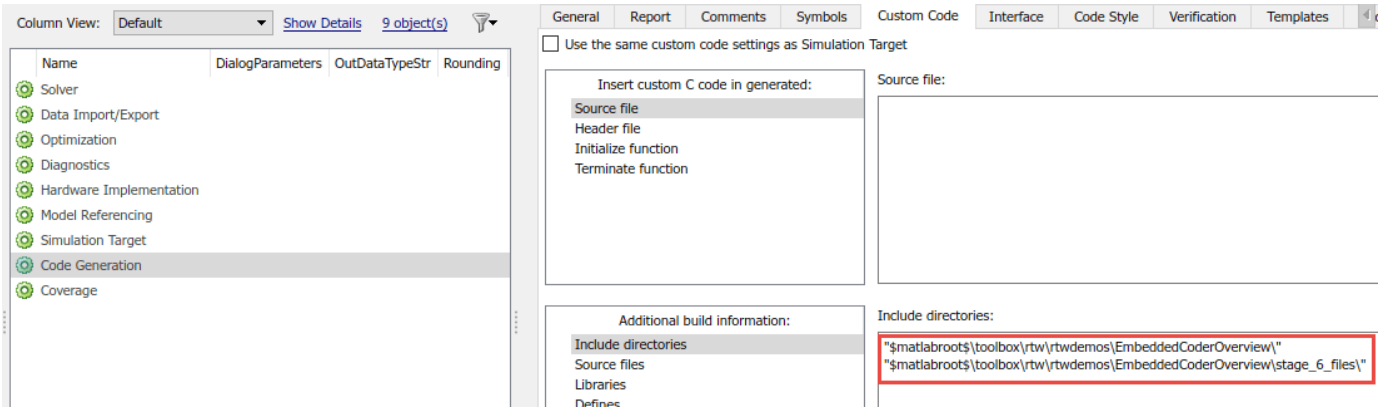
```
void Plant(void)
```

`HardwareInputs.c`: This function provides the `pos_req` signal and adds noise from the `Input_Signal_Scaling` subsystems into the plant feedback signal.

```
void hardwareInputs(void)
```

The handwritten function in `WriteDataForEval.c` logs output data from the test. After the test is finished, the function executes and writes the data to the file `PCG_Eval_ExternSimData.m`. You can run this script file in MATLAB to access the data. You can then compare the data to the simulated data from Simulink.

To enable these additional files, identify them by using **Configuration Parameters > Code Generation > Custom Code > Additional build information**.



Test via Test Vector Import and Export (Eclipse Environment)

Before building an executable in the Eclipse™ environment, regenerate the code without the S-function interface.

Build C code for integration.

To learn how to install and use Eclipse and GCC, see “Install and Use Cygwin and Eclipse” on page 53-68.

To install the files for this example automatically, click the hyperlink.

Automatically set up the build folder.

Alternatively, to manually install the files:

1. Create a build folder (`Eclipse_Build_P6`).
2. Unzip the file `rtwdemo_PCG_Eval_P6.zip` into the build folder.
3. Delete the files `rtwdemo_PCG_Eval_P6.c`, `ert_main.c`, and `rt_logging.c`, which are replaced by `example_main.c`.

Run the control code in Eclipse. The function in `writeDataForEval.c` creates the file `eclipseData.m`, which contains the output test data. In MATLAB, load this Eclipse data by running the script file. Then, compare this data with the data from the Simulink test harness model by using `plot`.

For the next example in this series, see “Evaluate Performance of the Generated Code” on page 53-64.

See Also

More About

- “SIL and PIL Simulations” on page 88-2

Evaluate Performance of the Generated Code

Learn how characteristics of generated code and RAM/ROM data affect the RAM/ROM metric.

For information about the example model and other examples in this series, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Code Evaluation

You can evaluate the generated code based on two primary metrics: *execution speed* and *memory usage*. In some cases, improvement in one metric implies a sacrifice in the other metric. For example, you can often gain faster execution by consuming more memory.

You can further classify memory usage as ROM (read-only memory) and RAM (random access memory).

- Accessing data from RAM is faster than accessing data from ROM.
- Executables and data must be stored on ROM because RAM does not maintain data between power cycles.

This example evaluates memory requirements and divides the memory usage into function and data components. The example does not evaluate execution speed.

Compiler Information

This evaluation uses the Freescale™ CodeWarrior® compiler.

- **Compiler:** Freescale™ CodeWarrior®
- **Version:** 5.5.1.1430
- **Target Processor:** Power PC 565

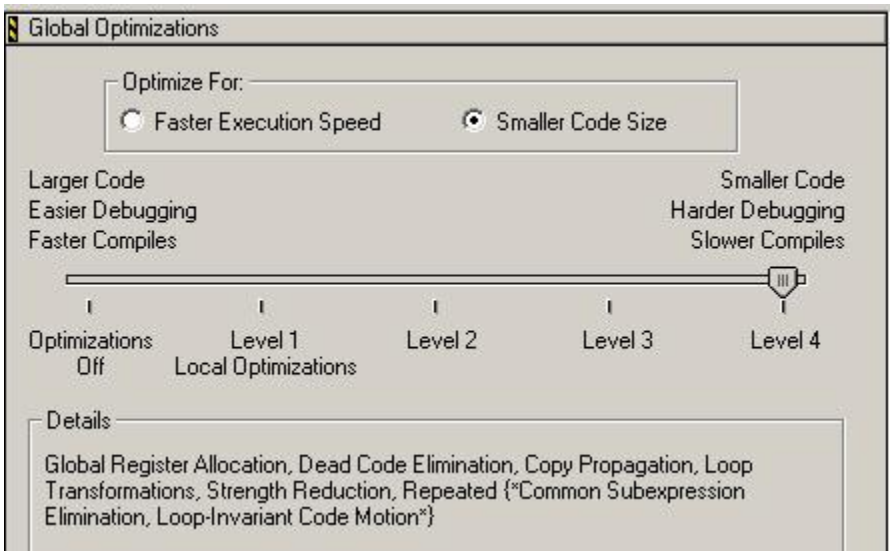
View Code Metrics

As described in “Build Integrated Code Outside the Simulink Environment” on page 53-53 and “Test Generated Code” on page 53-58, the generated code can require the use of utility functions. The utility functions incur a one-time, fixed cost of memory. Due to this fixed overhead, the data in this example show memory usage for:

- Algorithms: The C code generated from the Simulink® block diagrams plus the data definition functions
- Utilities: Functions that are part of the Simulink® Coder™ library source
- Full: The sum of both the algorithm and utilities

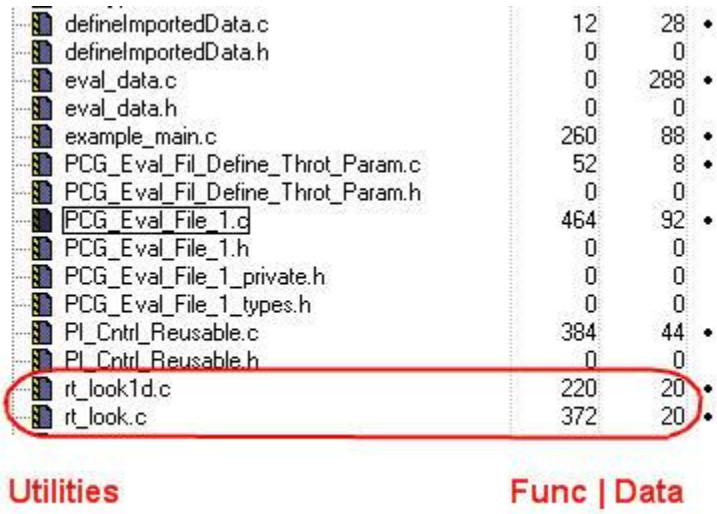
Build Options Configuration

The three evaluations in this example use the same build configuration. Freescale™ CodeWarrior® was configured to minimize memory usage and apply all applicable optimizations.



Configuration 1: Reusable Functions with Data Type Double

- **Source files:** `PCG_Eval_File_1.zip`
- **Data Type:** All doubles
- **Included Data:** The project includes all data that the build requires (including data declared as `extern`: `pos_rqst`, `fbk_1`, and `fbk_2`)
- **Main Function:** A modified version of `example_main` from “Build Integrated Code Outside the Simulink Environment” on page 53-53
- **Function Call Method:** Reusable functions for the PI controllers



Memory Usage

```
            Function    Data
Algorithms  1172 bytes  549 bytes
Utilities   592 bytes   40 bytes
Full        1764 bytes  589 bytes
```

Configuration 2: Reusable Functions with Data Type Single

In this configuration, the model data use a single-precision floating-point data type instead of double precision.

Model Configuration

- **Source files:** `PCG_Eval_File_2.zip`
- **Data Type:** All singles
- **Included Data:** The project includes all data that the build requires (including data declared as `extern`: `pos_rqst`, `fbk_1`, and `fbk_2`)
- **Main Function:** A modified version of `example_main` from "Build Integrated Code Outside the Simulink Environment" on page 53-53
- **Function Call Method:** Reusable functions for the PI controllers

Memory Usage

```
            Function    Data
Algorithms  800 bytes   308 bytes
Utilities   592 bytes   40 bytes
Full        1392 bytes  348 bytes
```

This configuration uses only 56% of the data memory from the first configuration: 308 bytes instead of 549 bytes. This configuration also uses 68% of the function memory: 800 bytes instead of 1172 bytes. For this system, using single precision instead of double precision does not affect the accuracy of the control algorithm, so you can use this configuration to achieve more efficient code.

Configuration 3: Nonreusable Functions with Data Type Single

- **Source files:** PCG_Eval_File_3.zip
- **Data Type:** All singles
- **Included Data:** The project includes all data that the build requires (including data declared as `extern`: `pos_rqst`, `fbk_1`, and `fbk_2`)
- **Main Function:** A modified version of `example_main` from "Build Integrated Code Outside the Simulink Environment" on page 53-53
- **Function Call Method:** The function interface is `void void` so data exchange occurs through global variables

Memory Usage

```
            Function    Data
Algorithms  948 bytes   348 bytes
Utilities   592 bytes   40 bytes
Full        1540 bytes  388 bytes
```

This configuration consumes more data and function memory than the previous configuration.

See Also

More About

- "Generate Static Code Metrics Report for Simulink Model" on page 54-29

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

Install and Use Cygwin and Eclipse

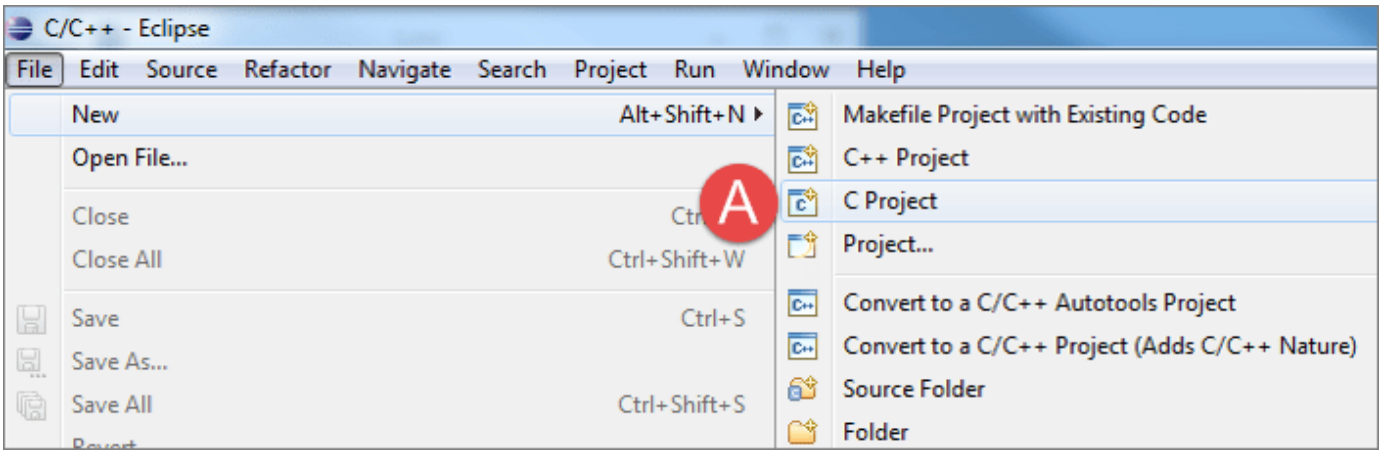
Use Eclipse™ and Cygwin™, a free software integration development environment (IDE) and free compiler, for the examples “Build Integrated Code Outside the Simulink Environment” on page 53-53 and “Test Generated Code” on page 53-58.

For information about the example model and other examples in this series, see “Prepare a Control Algorithm Model for C Code Generation” on page 53-18.

Create an Eclipse CDT Managed Make C Project

1. Open Eclipse.

2. Select **File > New > C Project**.



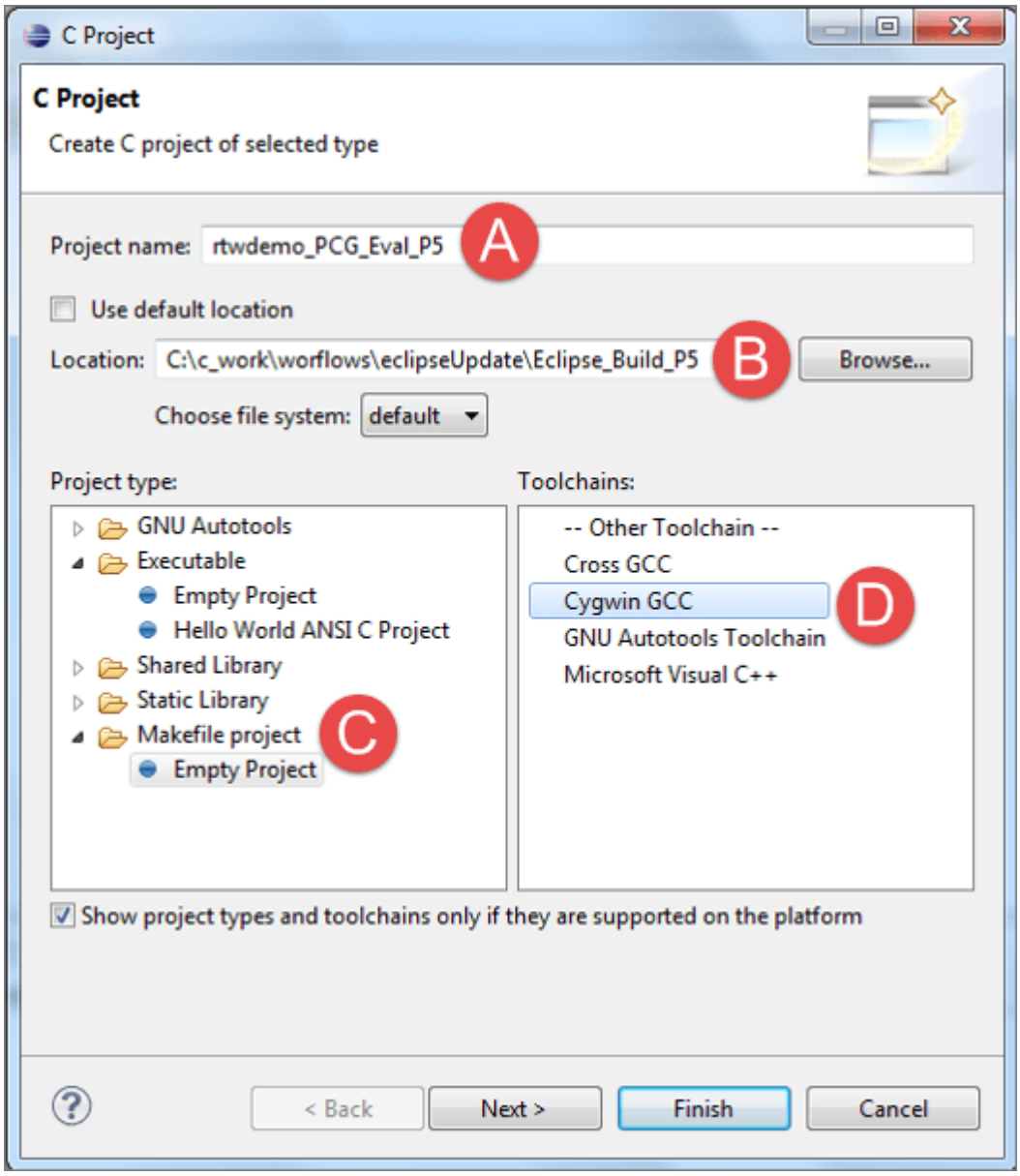
3. Use the **C Project** dialog box to configure the project.

a. Specify the project name.

b. Select the project location.

c. Select the project type as **Makefile**.

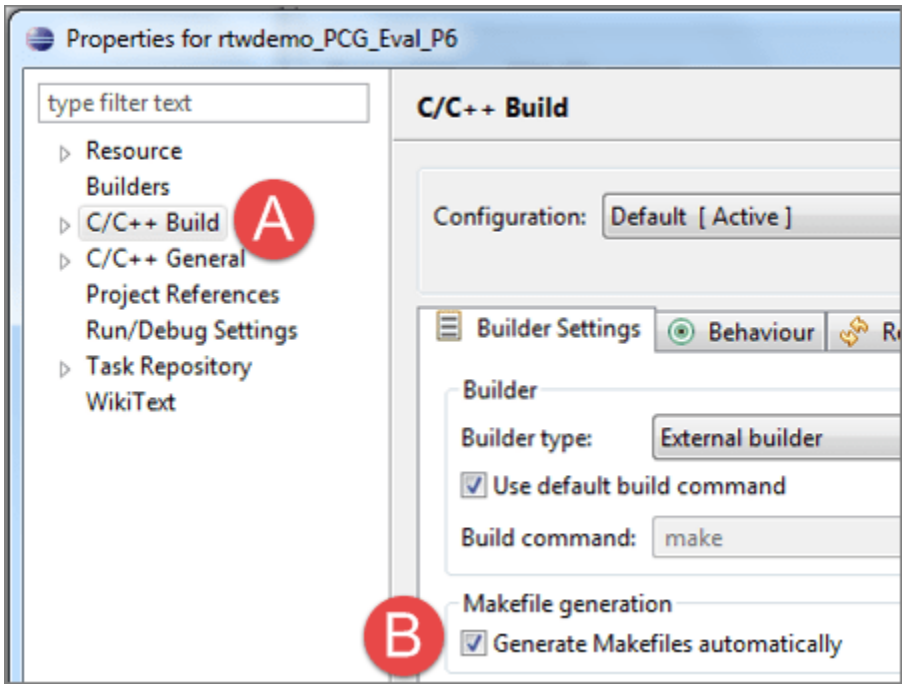
d. Select the Cygwin GCC toolchain and click **Finish**.



4. Select **Project > Properties**. Use **Advanced settings** to fully configure the project.

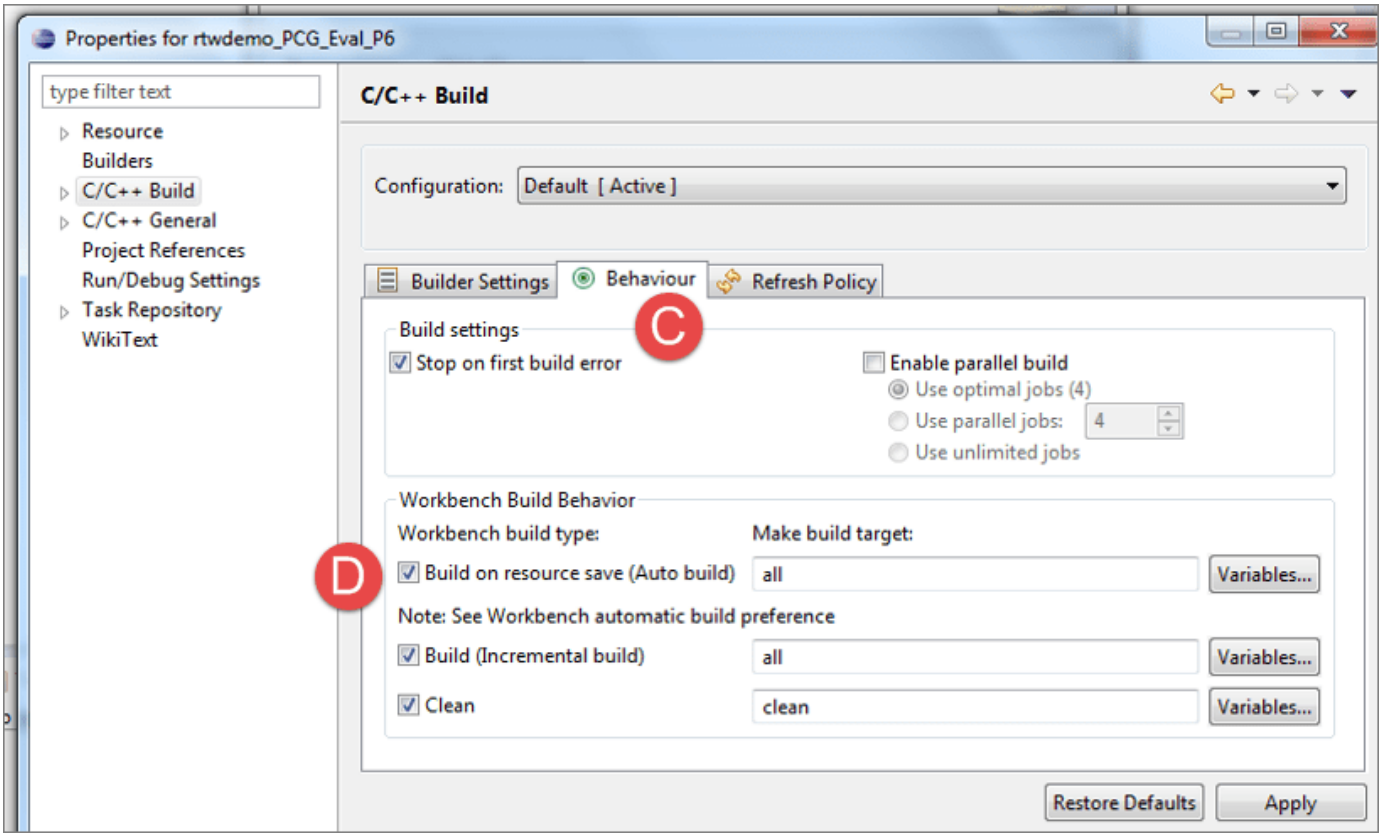
a. Select the **C/C++ Build** properties.

b. On the **Builder Settings** tab, select **Generate Makefiles automatically**.



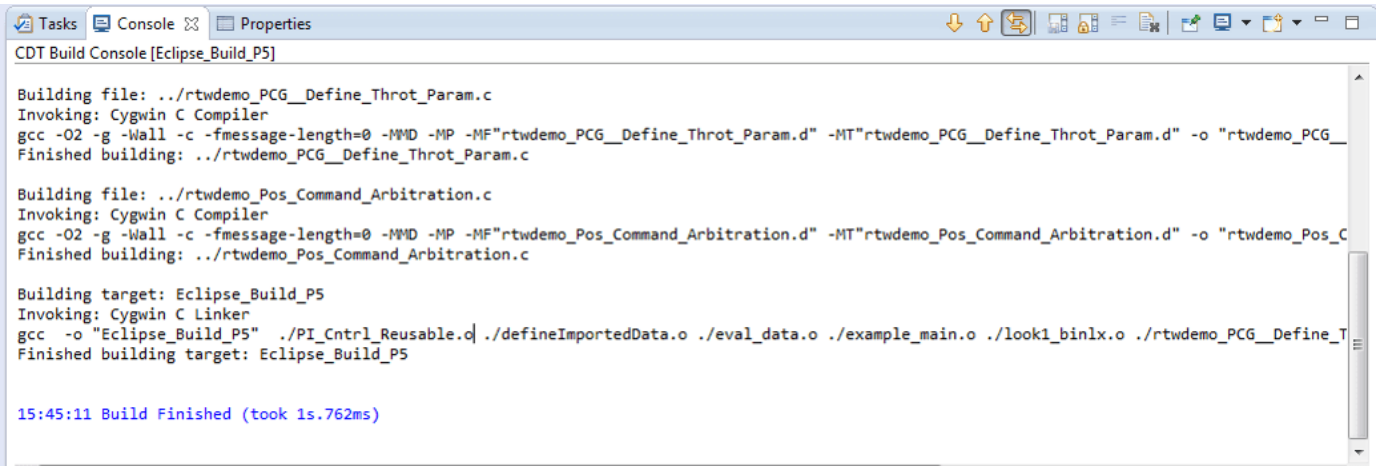
c. View the **Behaviour** tab.

d. Select **Build on resource save (Auto build)**.



5. Compile the generated code with Eclipse.

After you define the files, Eclipse determines the main function and builds the project with the managed make environment. The build details appear in the Console window.



Use Eclipse Debugger with the Generated Code

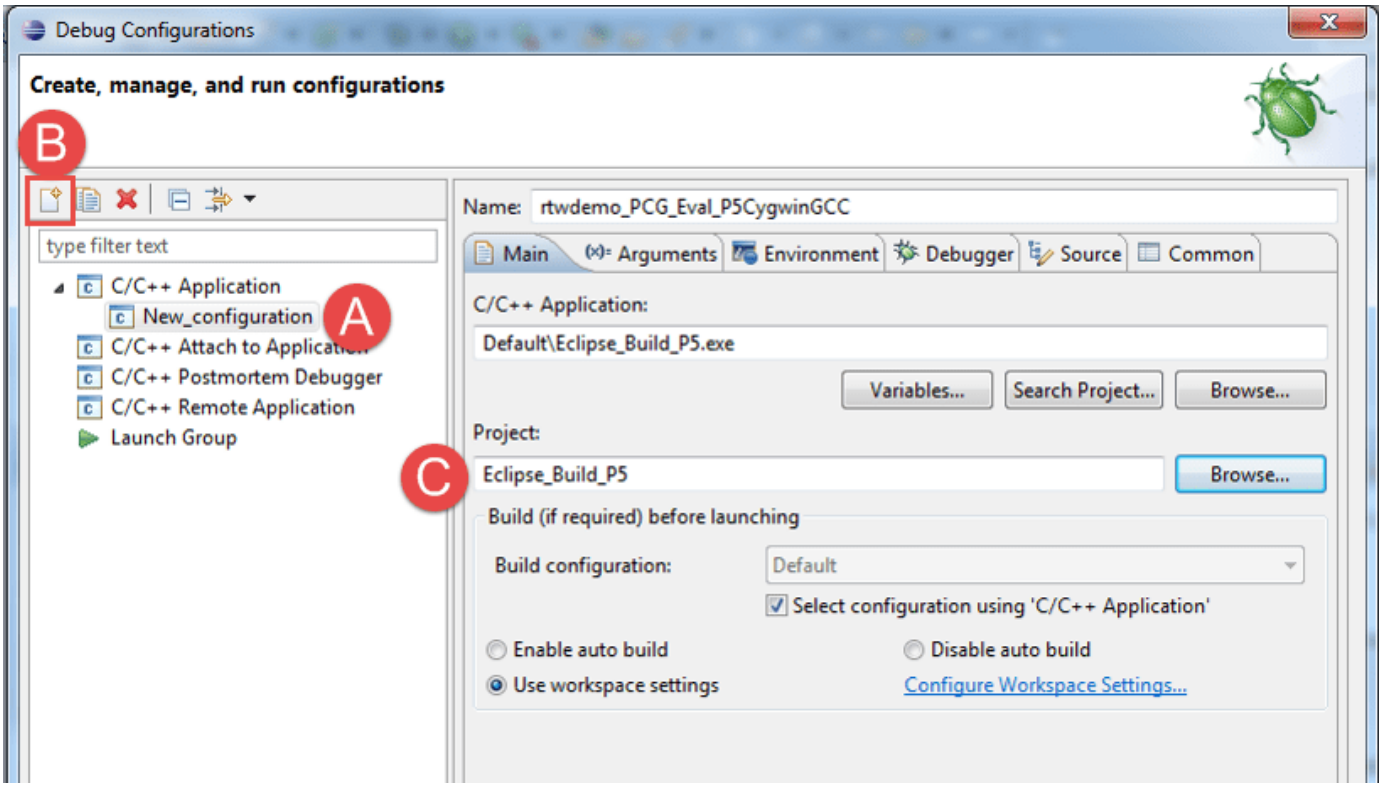
1. Select **Run > Debug Configurations**.

2. Use the **Debug Configurations** dialog box to configure the debugger.

a. Select **C/C++ Application**.

b. Click the new configuration button.

c. Select the project.

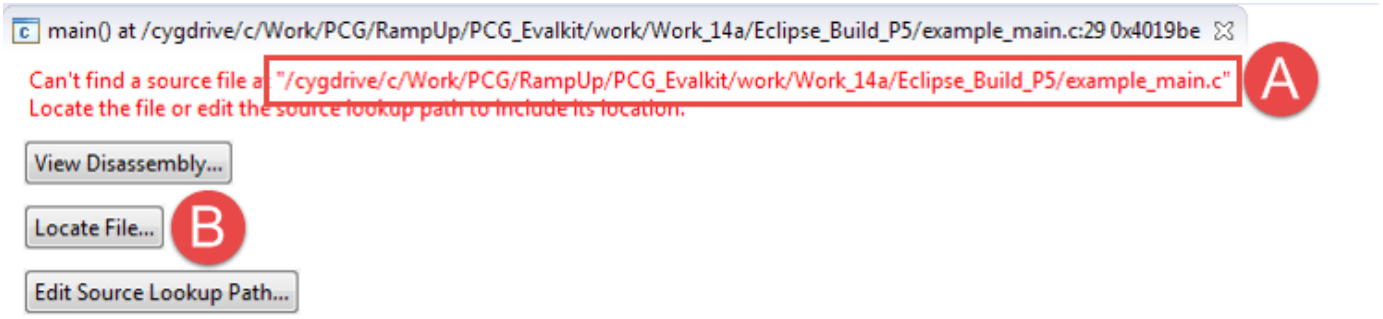


d. Click **Apply**. Do not click **Run**.

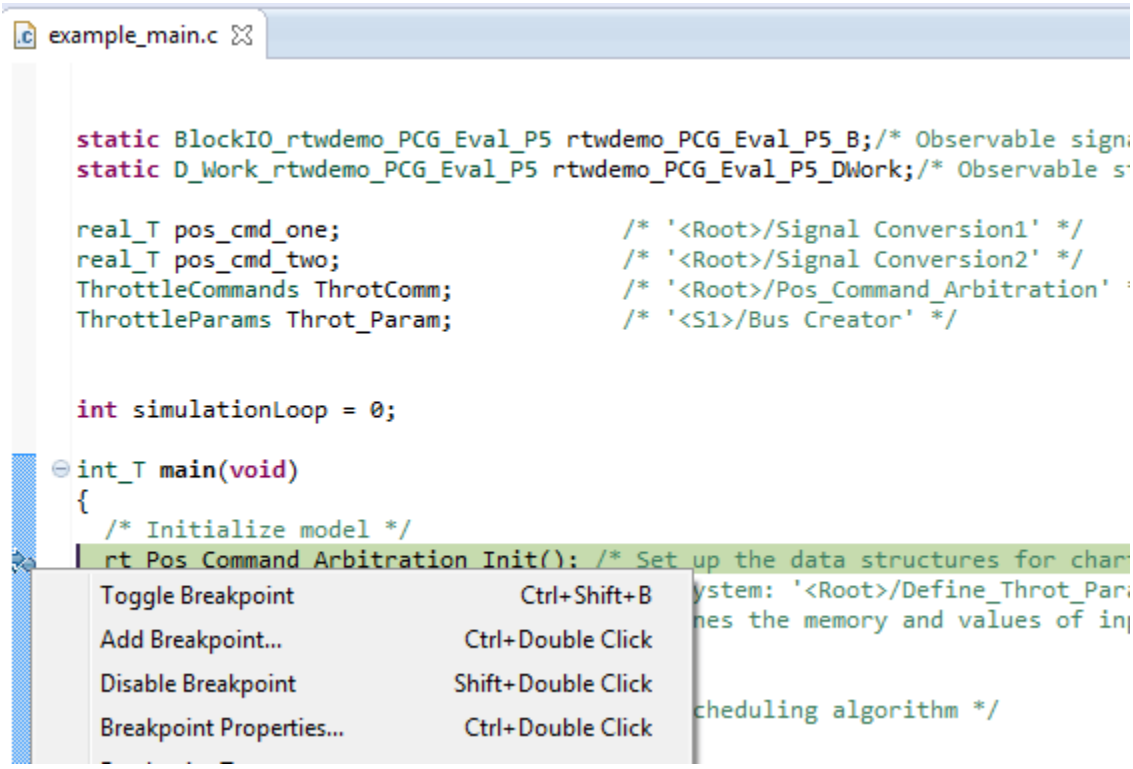
If required, select the executable. Eclipse automatically finds the `exe` file.

3. To start the debugger, from the primary Eclipse window, select **Run > Debug**.

During the build process, Cygwin creates a virtual drive, `/cygdrive/c/`. To run the debugger, Eclipse either remaps the drive or locates the files. After locating the first file, Eclipse finds the remaining files.



4. Run the debugger on the generated code.



Use Eclipse Debugger Commands

You can use these commands from the debugger context menu:

- **F5**: Step into
- **F6**: Step over
- **F7**: Step out
- **F8**: Resume
- **Ctrl+Shift+B**: Toggle break point
- **Ctrl+Shift+R**: Run to line
- **Shift+F5**: Use Step Filters

See Also

More About

- “Manage Build Process Folders” on page 52-30
- “Manage Build Process Files” on page 52-35
- “Manage Build Process File Dependencies” on page 52-43
- “Manage File Packaging of Generated Code Modules” on page 53-12

Report Generation in Embedded Coder

- “Reports for Code Generation” on page 54-2
- “Generate Code Generation Report” on page 54-5
- “Configure and Generate Code Generation Report Programmatically” on page 54-7
- “Web View of Model in Code Generation Report” on page 54-10
- “Analyze Generated Data Code Interface” on page 54-13
- “Analyze Generated Service Code Interface Report” on page 54-21
- “Static Code Metrics” on page 54-26
- “Generate Static Code Metrics Report for Simulink Model” on page 54-29
- “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” on page 54-33
- “Analyze Code Replacements in Generated Code” on page 54-36
- “Document Generated Code with Simulink Report Generator” on page 54-38
- “Get Code Description of Generated Code” on page 54-44
- “Get Generated Type and Name of Inport Structures” on page 54-51
- “Get Metadata About Service Interface” on page 54-54

Reports for Code Generation

In this section...
“HTML Code Generation Report Content” on page 54-2 “HTML Code Generation Report Location” on page 54-2 “HTML Code Generation Report for Referenced Models” on page 54-2 “HTML Code Generation Report Extensions” on page 54-3

The code generator software produces an HTML code generation report so that you can view and analyze the generated code. When you build your model, the code generation process produces an HTML file that is displayed in an HTML browser or in the Model Explorer. For an example, see “Generate Code Generation Report” on page 54-5.

If you have Simulink Report Generator, you can document your code generation project in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. For an example of how to create a Microsoft Word report, see “Document Generated Code with Simulink Report Generator” on page 54-38.

HTML Code Generation Report Content

The code generation report includes:

- The **Summary** section that contains model and code information, including **Author**, **Tasking Mode**, **System Target File**, **Hardware Device Type**, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box. The dialog box shows the Simulink model settings at the time of code generation, including TLC options.
- The **Subsystem Report** section that contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section that provides information about the generated code interface, including model entry-point functions and input/output data. For more information, see “Analyze Generated Data Code Interface”.
- In the **Code** section on the **Contents** pane, you can click the names of source code files generated from your model to view their contents.

HTML Code Generation Report Location

The default location for the code generation report files is in the `html` subfolder of the build folder, *`model_target_`*`rtw/html/`*. *`target`* is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is `index.html`. For more information on the location of the build folder, see “Manage Build Process Folders”.

HTML Code Generation Report for Referenced Models

To generate a code generation report for a top model and code generation reports for each referenced model, you need to specify the **Create code generation report** on the **Code Generation** > **Report** pane for the top model and each referenced model. You can open the code generation report of a referenced model in one of two ways:

- From the top-model code generation report, you can access the referenced model code generation report by clicking the name of the top model above the generated code. In the drop-down list of the model reference hierarchy, click the referenced model. To navigate back to the top model code generation report, use the **Back** button at the top of the left navigation pane or use the model reference hierarchy context menu.
- In the referenced model diagram window, on the **C Code** tab, click **Open Report**.

For more information, see “Generate Code for Model Reference Hierarchy”.

HTML Code Generation Report Extensions

If you have Embedded Coder, the code generator enhances the HTML code generation report. Configure your model to include these sections in the report:

- The **Traceability Report** section enables you to account for **Eliminated / Virtual Blocks** that are untraceable versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**. This section provides a complete mapping between model elements and code. For more information, see “Customize Traceability Reports” on page 85-29.
- The **Static Code Metrics Report** section provides generated code statistics. Metrics are estimated from static analysis of the generated code. To generate a static code metrics report without generating a code generation report, use the function `coder.report.generateCodeMetrics`. For more information, see “Static Code Metrics” on page 54-26.
- The **Code Replacements Report** section enables you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. For more information, see “Analyze Code Replacements in Generated Code” on page 54-36.
- The **Coder Assumptions** section provides a list of:
  - Code generation assumptions for your target hardware that you can check.
  - Expected results for the assumption checks.

  For more information, see “Check Code Generation Assumptions” on page 58-12.
- The model web view displays an interactive model diagram within the code generation report and supports traceability between the source code and the model. You can then share your model and generated code outside of the MATLAB environment. For more information, see “Web View of Model in Code Generation Report” on page 54-10.

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

- If you enable code-to-model traceability, hyperlinks within the displayed source code navigate to the blocks or subsystems from which the code is generated. For more information, see “Code-to-Model Traceability” on page 85-8.
- If you enable model-to-code traceability, you can navigate to the generated code for a block in the model. For more information, see “Model-to-Code Traceability” on page 85-10.
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations. For more information, see “Configure Code Coverage with Third-Party Tools” on page 91-9.

- If you select the **Generate static code metrics** check box on the **Code Generation > Report** pane, you can view code metrics information and navigate to code definitions and declarations in the generated code. For more information, see “View Static Code Metrics and Definitions Within the Generated Code” on page 54-28.

Note To view the contents of your generated code and navigate between model and code, you can also use the Code view in the Code perspective. In the perspective, click the **Code** tab in the bottom-right corner.

See Also

Related Examples

- “Verify Generated Code by Using Code Tracing” on page 85-2

Generate Code Generation Report

You can configure a model to generate a code generation report when you build the model. If you want to separate report generation from code generation, you can manually generate a report after generating code. If you do not change the model after generating code, you can generate a report without regenerating the code.

If you have Embedded Coder, you can include optional sections in the report, such as reports on traceability and code replacement. For more information, see “Reports for Code Generation” on page 54-2.

Configure Model

Configure a model to generate code without generating a report.

1. Open a model, for example, `rtwdemo_sum`.

   ```
   model = 'rtwdemo_sum';
   open_system(model);
   ```

2. On the **C Code** tab, from the **Open Report** list, select **Report Options**. The Configuration Parameters dialog box opens with the **Code Generation > Report** pane visible.
3. Clear the **Create code generation report** parameter. When you generate code, the report is not generated. Instead, you generate the report manually.
4. If you have Embedded Coder, verify that **Eliminated / virtual blocks** is selected. This parameter enables an optional **Traceability Report** section in the code generation report.
5. Generate code.

Generate Report

On the **C Code** tab, select **Open Report**. The report generation process writes the code generation report files to the `html` subfolder of the build folder (see “HTML Code Generation Report Location” on page 54-2) without regenerating the code. Next, a MATLAB web browser window opens and displays the code generation report.

If you selected **Eliminated / virtual blocks**, the code generation report includes a **Traceability Report** section.

To open the code generation report at any time after generating code, on the **C Code** tab, select **Open Report**. If a report already exists, it opens without regenerating the code or report.

Share Report

To share the code generation report, you can package the code generation report files and supporting files into a ZIP file for transfer.

1. In the **Current Folder** browser, select these folders:

   - `/slprj`

- *model_target*_rtw

2. To open the context menu, right click one of the selected folders.
3. In the context menu, select **Create Zip File**. A file appears in the **Current Folder** browser.
4. Name the ZIP file.

If your model is part of a project, you can include the code generation report when archiving the project. For more information, see “Archive Projects”.

Limitation

After building your model or generating the code generation report, if you modify legacy or custom code, you must rebuild your model or regenerate the report for the code generation report to include the updated legacy source files. For example, if you modify your legacy code, and then use the **Open Report** button to open an existing report, the software does not check if the legacy source file is out of date compared to the generated code. The code generation report is not regenerated and the report includes the out-of-date legacy code.

To reflect changes to legacy or custom code in the code generation report, rebuild the model before using the **Open Report** button.

See Also

More About

- “Reports for Code Generation” on page 54-2
- “Configure and Generate Code Generation Report Programmatically” on page 54-7

Configure and Generate Code Generation Report Programmatically

You can programmatically generate, open, and close an HTML code generation report at the MATLAB® command line using the `coder.report.generate` function. When you generate the report, you can configure it rather than tie the report configuration to the model configuration set. If you do not change the model after generating code, you can generate a code generation report without regenerating the code.

Configure Model

Open the model.

```
model='rtwdemo_sum';
open_system(model)
```

Clear the `GenerateReport` configuration parameter programmatically. When you build the code, the report is not generated. Instead, you generate the report manually at the command line.

```
set_param(model,'GenerateReport','off');
```

Build the model.

```
slbuild(model);
```

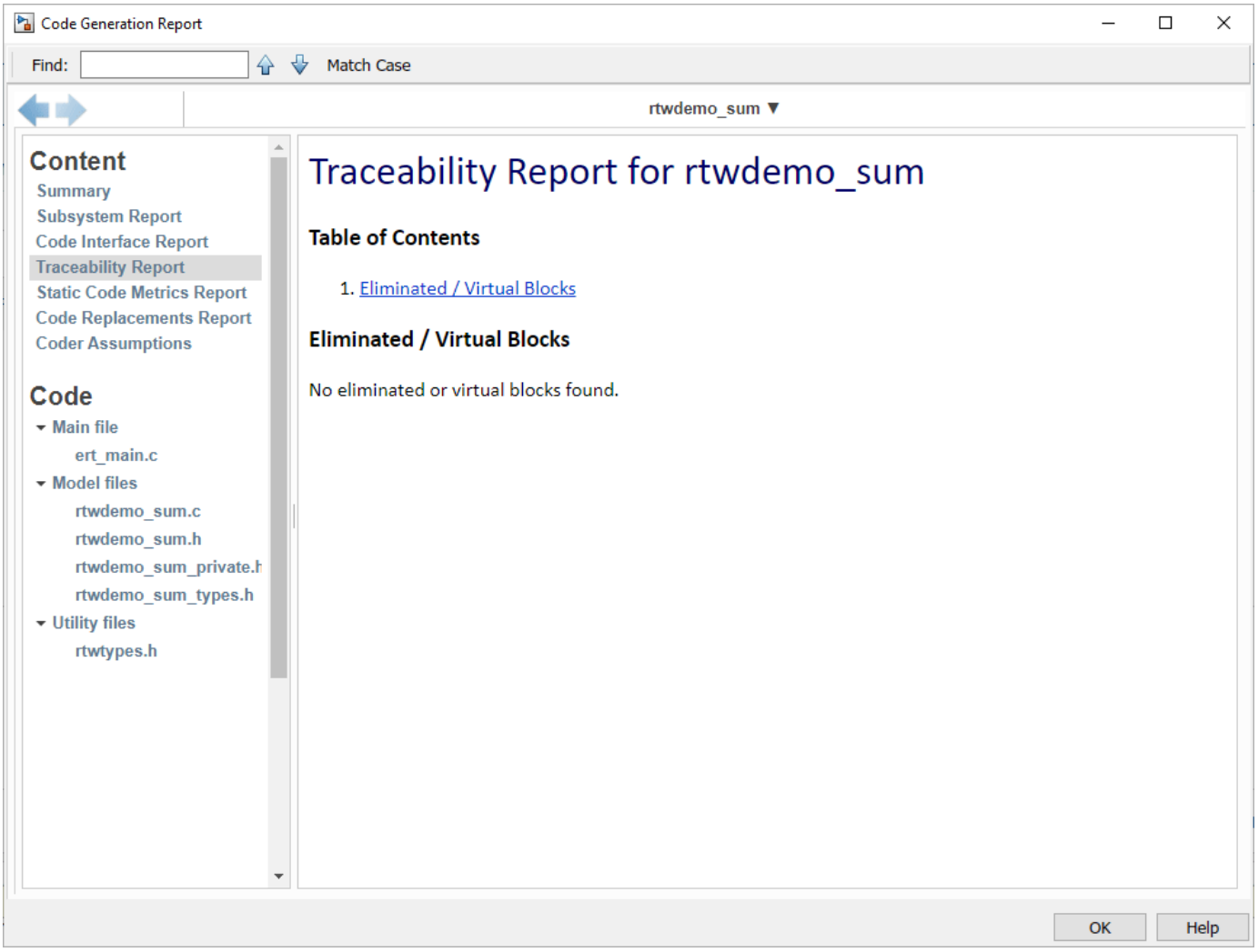
```
### Starting build procedure for: rtwdemo_sum
### Successful completion of build procedure for: rtwdemo_sum

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================================
rtwdemo_sum  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.294s
```

Generate Report

When you generate the report at the command line, you can use name-value arguments to configure the report without changing the model configuration set. Generate the report with `GenerateTraceReport` set to `'on'`.

```
coder.report.generate(model,GenerateTraceReport='on');
```

The code generation report includes a **Traceability Report** section.



Verify that `GenerateTraceReport` is not selected in the model configuration set.

```
get_param(model,'GenerateTraceReport')
```

```
ans =
'off'
```

Close the code generation report.

```
coder.report.close();
```

To reopen the report programmatically, use `coder.report.open`.

```
coder.report.open(model);
```

Share Report

To share the code generation report, package the code generation report files and supporting files into a ZIP file for transfer.

```
zip('myzip',{'slprj','rtwdemo_sum_ert_rtw'});
```

If your model is part of a project, you can include the code generation report when archiving the project. For more information, see “Archive Projects”.

Limitation

After building your model or generating the code generation report, if you modify legacy or custom code, you must rebuild your model or regenerate the report for the code generation report to include the updated legacy source files. For example, if you modify your legacy code, and then use `coder.report.open` to open an existing report, the software does not check if the legacy source file is out of date compared to the generated code. The code generation report is not regenerated and the report includes the out-of-date legacy code.

To reflect changes to legacy or custom code in the code generation report, before using `coder.report.open`, rebuild the model.

See Also

`coder.report.generate` | `coder.report.open` | `coder.report.close`

More About

- “Reports for Code Generation”
- “Generate Code Generation Report”

Web View of Model in Code Generation Report

In this section...

"Generate HTML Code Generation Report with Model Web View" on page 54-10
"Model Web View Limitations" on page 54-12

To review and analyze the generated code, it is helpful to navigate between the code and model. If you have a Simulink Report Generator license, you can include a Web view of the model within the HTML code generation report. You can then share your model and generated code outside of the MATLAB environment. When you generate the report, the Web view includes the block diagram attributes displayed in the Simulink Editor, such as, block sorted execution order, signal properties, and port data types. You can display the generated Web views in a supported Web browser. See "Web Views" (Simulink Report Generator) for browser requirements and to read more about Web views.

Generate HTML Code Generation Report with Model Web View

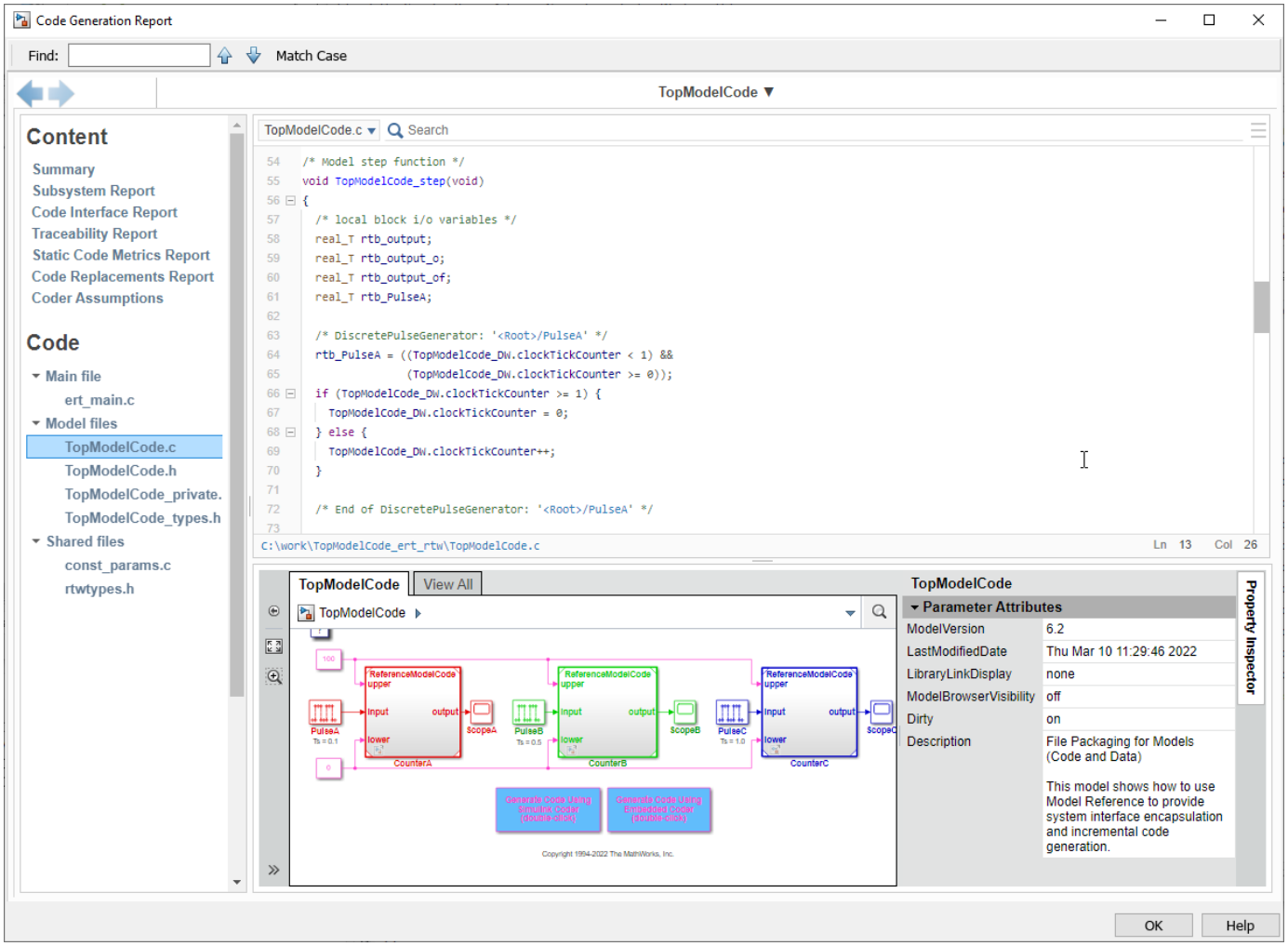
This example shows how to create an HTML code generation report which includes a Web view of the model diagram.

1. Open the `TopModelCode` model. At the command line, enter:

```
openExample('simulinkcoder/FilePackagingModelsCodeAndDataExample', ...
 'supportingFile', 'TopModelCode');
```

2. Open the Configuration Parameters dialog box and navigate to the **Code Generation** pane.
3. Specify `ert.tlc` for the **System target file** parameter.
4. Open the **Code Generation** > **Report** pane.
5. Select the following parameters:
   - **Create code generation report**
   - **Open report automatically**
   - **Generate model Web view**
6. Select the parameters **Code-to-model** and **Model-to-code**.

   Note These settings specify only the top model, not referenced models.
7. Open the Configuration Parameters for the referenced model, `ReferenceModelCode` and perform steps 3–6.
8. Save the models, `TopModelCode` and `ReferenceModelCode`.
9. From the top model diagram, press Ctrl+B. After building the model and generating code, the code generation report for the top model opens in a MATLAB Web browser.
10. In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinks.



11 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.

12 To highlight the generated code for a Model block in your model, click `CounterB`. The corresponding code is highlighted in the source code pane.

Note You cannot open the referenced model diagram in the Web view by double-clicking the Model block in the top model.

---

13 To open the code generation report for a referenced model, at the top of the report, click **TopModelCode**. In the drop-down, click **ReferenceModelCode**. The source files for the referenced model are displayed along with the Web view of the referenced model.

14 To go back to the code generation report for the top model, at the top of the left navigation pane, click the **Back** button until the top model's report is displayed.

For more information about exploring a model in a Web view, see "Explore Web Views" (Simulink Report Generator).

For more information about navigating between the generated code and the model diagram, see "Trace Simulink Model Elements in Generated Code" on page 85-8.

Model Web View Limitations

The HTML code generation report includes the following limitations when using the model Web view:

- Code is not generated for virtual blocks. In the model Web view of the code generation report, when tracing between the model and the code, when you click a virtual block, it is highlighted yellow.
- In the model Web view, you cannot open a referenced model diagram by double-clicking the Model block in the top model. Instead, open the code generation report for the referenced model by using the drop-down menu at the top of the report.
- Stateflow truth tables, events, and links to library charts are not supported in the model Web view.
- Searching in the code generation report does not find or highlight text in the model Web view.
- If you navigate from the actual model diagram (not the model Web view in the report), to the source code in the HTML code generation report, the model Web view is disabled and not visible. To enable the model Web view, open the report again.
- For a subsystem build, the traceability hyperlinks of the root level inport and outport blocks are disabled.
- “Traceability Limitations” on page 85-7 that apply to tracing between the code and the actual model diagram.

See Also

More About

- “Web Views” (Simulink Report Generator)
- “Reports for Code Generation” on page 54-2
- “Generate Code Generation Report” on page 54-5
- “Configure and Generate Code Generation Report Programmatically” on page 54-7

Analyze Generated Data Code Interface

In this section...

"Navigating Code Interface Report Subsections" on page 54-13
"Interpreting the Entry-Point Functions Subsection" on page 54-14
"Interpreting the Inports and Outports Subsections" on page 54-17
"Interpreting the Interface Parameters Subsection" on page 54-18
"Interpreting the Data Stores Subsection" on page 54-19
"Code Interface Report Limitations" on page 54-19

When you generate a code generation report, you can include a code interface report, which documents the generated code interface including model entry-point functions and interface data. The code interface information helps you to review the generated code and integrate it with other code. The information in the report depends on the type of code interface configuration that you use. For models that use service interface code configuration, the code interface report also includes information about services, as described in "Analyze Generated Service Code Interface Report" on page 54-21.

For code configured with data interfaces, the code interface report includes the following subsections:

- **Entry-Point Functions** — interface information about each model entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.
- **Inports** and **Outports** — interface information about each model inport and outport.
- **Interface Parameters** — interface information about tunable parameters that are associated with the model.
- **Data Stores** — interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

To generate a code interface report, select the configuration parameters **Create code generation report** and **Code Interface Report**, as described in "Generate Code Generation Report" on page 54-5. For limitations that apply to code interface reports, see "Code Interface Report Limitations" on page 54-19.

For illustration purposes, this section uses the following models:

- `rtwdemo_configinterface` for examples of report subsections
- `rtwdemo_mrmtbb` for examples of timing information
- `rtwdemo_fcnprotoctrl` for examples of function argument and return value information

Navigating Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides collapse/expand tokens and hyperlinks.

- For a large subsection, the report provides `[-]` and `[+]` symbols that allow you to collapse or expand that section.
- The code interface report provides several forms of hyperlink navigation. For example:

- The **Table of Contents** located at the top of the code interface report provides links to each subsection.
- You can click each function name to go to its definition in *model*.c.
- You can click the header file name for each function to go to the header file source listing.
- If you selected the configuration parameter **Code-to-model**, to go to the corresponding location in the model display, you can click hyperlinks for these constructs:
  - Function argument
  - Function return value
  - Variable representing an inport
  - Variable representing an outport
  - Interface parameter (if the parameter source is a block)
  - Variable representing a data store (if the data store source is a Data Store Memory block)

For backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** buttons above the **Contents** section in the upper-left corner of the report.

Interpreting the Entry-Point Functions Subsection

The **Entry-Point Functions** subsection of the code interface report provides the following interface information about each model entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.

Field	Description
Function	The function name. You can click the function name to go to its definition in *model*.c.
Prototype	Displays the function prototype, including the function return value, name, and arguments.
Description	Provides a text description of the purpose of the function in the application.
Timing	Describes the timing characteristics of the function, such as how many times the function is called, or if it is called periodically, and at what time interval. For a multirate timing example, see the following `rtwdemo_mrmtbb` report excerpt.
Arguments	If the function has arguments, displays the number, name, data type, and Simulink description for each argument. For argument examples, see the `rtwdemo_fcnprotoctrl` report excerpt below.
Return value	If the function has a return value, this field displays the return value data type and Simulink description. If you selected the **Code-to-model** parameter for your model, you can click the hyperlink in the description to go to the block corresponding to the return value in the model display. For a return value example, see the following `rtwdemo_fcnprotoctrl` report excerpt.
Header file	The name of the header file for the function. You can click the header file name to go to the header file source listing.

For example, here is the **Entry-Point Functions** subsection for the model `rtwdemo_configinterface`.

Entry-Point Functions

Function: rtwdemo_configinterface_initialize

Prototype	**void rtwdemo_configinterface_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_configinterface.h

Function: rtwdemo_configinterface_step

Prototype	**void rtwdemo_configinterface_step(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_configinterface.h

Function: rtwdemo_configinterface_terminate

Prototype	**void rtwdemo_configinterface_terminate(void)**
Description	Termination entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_configinterface.h

To illustrate how timing information might be listed for a multirate model, here are the **Entry-Point Functions** and **Inports** subsections for the model `rtwdemo_mrmtbb`. This multirate, discrete-time, multitasking model contains Inport blocks 1 and 2, which specify 1-second and 2-second sample times, respectively. The sample times are constrained to the specified times by the **Periodic sample time constraint** option on the **Solver** pane of the Configuration Parameters dialog box.

Entry-Point Functions

Function: rtwdemo_mrmtbb_initialize

Prototype	**void rtwdemo_mrmtbb_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step0

Prototype	**void rtwdemo_mrmtbb_step0(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step1

Prototype	**void rtwdemo_mrmtbb_step1(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 2 seconds
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Inports

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1_1s	rtU.In1_1s	real_T	1
<Root>/In2_2s	rtU.In2_2s	real_T	1

To illustrate how function arguments and return values are displayed in the report, here is the entry-point function description of the model step function for model `rtwdemo_fcnprotoctrl`.

Function: rtwdemo_fcnprotoctrl_step_custom

Prototype	**boolean_T rtwdemo_fcnprotoctrl_step_custom(const real_T argIn1, const BusObject *const argIn2, BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)**
Description	Output entry point of generated code
Timing	Can be called at any time
Arguments	[-]

#	Name	Data Type	Description
1	argIn1	const real_T	*<Root>/In1*
2	argIn2	const BusObject *const	*<Root>/In2*
3	argOut2	BusObject *	*<Root>/Out2*
4	argIn3	const BusObject *const	*<Root>/In3*
5	argIn4	uint8_T *	*<Root>/In4*

Return value	Data Type	Description
	boolean_T	*<Root>/Out1*

Header file	rtwdemo_fcnprotoctrl.h

Interpreting the Inports and Outports Subsections

The **Inports** and **Outports** subsections of the code interface report provide the following interface information about each inport and outport in the model.

Field	Description
Block Name	Displays the Simulink block name of the inport or outport. If you selected the **Code-to-model** parameter for your model, you can click on each inport or outport **Block Name** value to go to its location in the model display.
Code Identifier	Lists the identifier associated with the inport or outport data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if model configuration parameter **Code interface packaging** is set to `Reusable function` — the field displays the text '`Defined externally`'.
Data Type	Lists the data type of the inport or outport.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
Dimension	Lists the dimensions of the inport or outport (for example, `1` or `[4, 5]`).

For example, here are the **Inports** and **Outports** subsections for the model `rtwdemo_configinterface`.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	*Imported data:* input1	MYTYPE	1
<Root>/In2	*Imported data:* input2	MYTYPE	1
<Root>/In3	*Imported data:* input3	MYTYPE	1
<Root>/In4	*Imported data:* input4	MYTYPE	1

Outputs

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	MYTYPE	1

Interpreting the Interface Parameters Subsection

The **Interface Parameters** subsection of the code interface report provides the following interface information about tunable parameters that are associated with the model.

Field	Description
Parameter Source	Lists the source of the parameter value, as follows: • If the source of the parameter value is a block, the field displays the block name, such as `<Root>/Gain2` or `<S1>/Lookup1`. If you selected the **Code-to-model** parameter for your model, you can click the **Parameter Source** value to go to the parameter's location in the model display. • If the source of the parameter value is a workspace variable, the field displays the name of the workspace variable.
Code Identifier	Lists the identifier associated with the tunable parameter data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label `'Imported data:'`. • If the data is neither defined nor declared in the generated code — for example, if model configuration parameter **Code interface packaging** is set to `Reusable function` — the field displays the text `'Defined externally'`.
Data Type	Lists the data type of the tunable parameter.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
Dimension	Lists the dimensions of the tunable parameter (for example, `1` or `[4, 5, 6]`).

For example, here is the **Interface Parameters** subsection for the model `rtwdemo_configinterface`.

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
'Table1' in model workspace	Table1	const volatile Table1_Type	1
'Table2' in model workspace	Table2	const volatile Table2_Type	1
'K1' in model workspace	K1	const volatile int8_T	1

Interpreting the Data Stores Subsection

The **Data Stores** subsection of the code interface report provides the following interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

Field	Description
Data Store Source	Lists the source of the data store memory, as follows: • If the data store is defined using a Data Store Memory block, the field displays the block name, such as `<Root>/DS1`. If you selected the **Code-to-model** parameter for your model, you can click on the **Data Store Source** value to go to the data store's location in the model display. • If the data store is defined using a `Simulink.Signal` object, the field displays the name of the `Simulink.Signal` object.
Code Identifier	Lists the identifier associated with the data store data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if model configuration parameter **Code interface packaging** is set to `Reusable function` — the field displays the text '`Defined externally`'.
Data Type	Lists the data type of the data store.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
Dimension	Lists the dimensions of the data store (for example, `1` or `[1, 2]`).

For example, here is the **Data Stores** subsection for the model `rtwdemo_configinterface`.

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	volatile boolean_T	1

Code Interface Report Limitations

These limitations apply to the code interface section of the HTML code generation reports.

- The code interface report supports data resolved with most storage classes, except when the storage class properties are set in one of these ways:
  - The storage class property **Type** is set to `FlatStructure`. For example, when the `BitField` and `Struct` storage classes in the Simulink package have **Type** set to `FlatStructure`.
  - The storage class property **Type** is set to `Other`. For example, when the `GetSet` storage class in the Simulink package has **Type** set to `Other`.
  - The storage class property **Data access** is set to `Pointer`, indicating that the code declares imported symbols as pointer variables rather than simple variables. This property is accessible only when the storage class property **Data scope** is set to `Imported` or `Instance-specific`.

  In these cases, the report displays empty **Data Type** and **Dimension** fields.
- For outports, if you optimize memory, the code interface report cannot describe the associated memory (data type and dimensions). In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Code Generation** > **Data Type Replacement** pane of the Configuration Parameters dialog box. The data types in the report link to built-in data types rather than their specified replacement data types.

See Also

Related Examples

- "How Generated Code Exchanges Data with an Environment" on page 36-35

Analyze Generated Service Code Interface Report

When you generate a code generation report, you can include a code interface report, which documents the generated code interface including model entry-point functions and services. The code interface information helps you to review the generated code and integrate it with other code. The information in the report depends on the type of code interface configuration that you use. For models that use a data interface configuration for the generated code, the code interface report includes information about data and functions. See “Analyze Generated Data Code Interface” on page 54-13.

For code that you configure with service interfaces, the code interface report includes these subsections:

- **Introduction** — A high-level description of how the code interface interacts with services provided by the target platform software.
- **Execution Interfaces** — Interface information about each model execution entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.
- **Service Interfaces** — Interface information about platform services that the generated code uses, including data transfer and timer services.

To generate a code interface report, select the configuration parameters **Create code generation report** and **Code Interface Report**. For more information, see “Generate Code Generation Report” on page 54-5. For limitations that apply to code interface reports, see “Code Interface Report Limitations” on page 54-24.

Navigate Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides show/hide tokens and hyperlinks.

- For a large subsection, the report provides **Hide** and **Show** buttons that allow you to collapse or expand that section.
- The code interface report provides several forms of hyperlink navigation:
  - The table of contents contains links to the sections of the report.
  - You can click each function name to go to its definition in *`model`*`.c`.
  - You can click the header file name for each function to go to the header file source listing.
  - If you select the configuration parameter **Code-to-model**, to go to the corresponding location in the model display, you can click hyperlinks for these constructs:
    - Function argument
    - Function return value
    - Variable representing an inport
    - Variable representing an outport
    - Interface parameter (if the parameter source is a block)
    - Variable representing a data store (if the data store source is a Data Store Memory block)

For backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** arrow buttons above the **Content** section in the upper-left corner of the report.

Interpret the Execution Interfaces Section

The **Execution Interfaces** section of the code interface report provides this interface information about each model entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.

- The function name
- Description of the purpose of the function including:
  - Sample time interval for functions that the code calls periodically. If the code does not call the function periodically, the section does not show a sample time.
  - A link to the associated model element for exported functions.
- Header file information
- Function prototype including the function return value, name, and arguments
- A list of services that the function calls, with a link to the report subsection for each service. You can also place your cursor over the link to see a tooltip with information about the service interface.

For example, consider the **Execution Interfaces** subsection for the model `ComponentDeploymentFcn`.

2. Execution Interfaces [hide]

A target platform initiates execution of a component by calling execution entry-point functions generated for the model. The type of function code that the code generator produces for a component model depends on the modeling style and periodic rates represented in the model. For more information, see Periodic and Aperiodic Function Interfaces and Startup, Reset, and Shutdown Function Interfaces.

2.1 Initialize Functions

Initialize entry-point functions implement startup behavior. In a model, Initialize Function blocks represent startup behavior explicitly.

CD_initialize
Initialization entry point of generated code

```
#include "ComponentDeploymentFcn.h"
void CD_initialize(void)
```

Calls services:

- Receiver: get_CD_initialize_InBus_NVM
- Data transfer: set_ModelInitialize_DataTransfer

2.2 Terminate Functions

Terminate entry-point functions implement shutdown behavior. In a model, Terminate Function blocks represent shutdown behavior explicitly.

CD_terminate
Termination entry point of generated code

```
#include "ComponentDeploymentFcn.h"
void CD_terminate(void)
```

Calls services:

- Sender: getref_CD_terminate_OutBus_NVM

2.3 Periodic Functions

Periodic entry-point functions implement component behavior occurring at a fixed sampling rate. In a function-call (export-function) model, Function Call Subsystem blocks that specify a sampling rate represent periodic functions.

CD_accumulator
Exported function: *<Root>/Periodic*. Must be called periodically, every 1 seconds.

```
#include "ComponentDeploymentFcn.h"
void CD_accumulator(void)
```

Calls services:

- Sender: getref_CD_accumulator_OutBus_y
- Data transfer: get_CD_accumulator_DataTransfer

```
const real_T * get_CD_accumulator_DataTransfer(void)
```

Data communication method: Outside execution
Target platform communicates data before or after generated code executes.

2.4 Aperiodic Fun

Aperiodic (asynchron ... g asynchronously. In a function-call (export-function) model, Function Call Subsystem blocks that do not specify a

CD_integrator
Exported function: *<Root>/Aperiodic*

```
#include "ComponentDeploymentFcn.h"
void CD_integrator(void)
```

Calls services:

- Receiver: get_CD_integrator_InBus_u
- Data transfer: set_CD_integrator_DataTransfer
- Timer: get_tick_outside_tick_CD_integrator

Interpret the Service Interfaces Section

The **Service Interfaces** section of the code interface report provides this interface information about each service in the generated code.

- The service name

- Description of the purpose of the service, including a link to the model element if the service is associated with a block.
- Header file information
- Service prototype
- A list of functions that call the service, with a link to the report subsection for each function
- Data communication method. The options are outside execution, during execution, or direct access.

For example, consider this portion of the **Service Interfaces** section for the model `ComponentDeploymentFcn`.

3.2 Sender Service Interfaces

Entry-point functions send data to a platform environment by calling sender service functions. A call to a sender service is represented in a model as an outport.

getref_CD_accumulator_OutBus_y
Sender service interface for *<Root>/Out*.

```
#include "services.h"
real_T * getref_CD_accumulator_OutBus_y(void)
```

Called by:

- CD_accumulator

Data communication method: Outside execution
Target platform communicates data before or after generated code executes.

getref_CD_terminate_OutBus_NVM
Sender service interface for *<Root>/OutNVM*.

```
#include "services.h"
real_T * getref_CD_terminate_OutBus_NVM(void)
```

Called by:

- CD_terminate

Data communication method: Outside execution
Target platform communicates data before or after generated code executes.

3.3 Data Transfer Service Interfaces

Within a platform environment, callable functions within a component exchange data by calling platform data transfer services. A call to data transfer service is represented in a model as a signal line that connects communicating functions.

get_CD_accumulator_DataTransfer
Receiver interface for data transfer

```
#include "services.h"
const real_T * get_CD_accumulator_DataTransfer(void)
```

Called by:

- CD_accumulator

Data communication method: Outside execution
Target platform communicates data before or after generated code executes.

set_CD_integrator_DataTransfer
Sender interface for data transfer

```
#include "services.h"
real_T * set_CD_integrator_DataTransfer(void)
```

Called by:

- CD_integrator

Code Interface Report Limitations

These limitations apply to the code interface section of the HTML code generation reports.

- The code interface report supports data resolved with most storage classes, except when the storage class properties are set in one of these ways:
  - The storage class property **Type** is set to `FlatStructure`. For example, when the `BitField` and `Struct` storage classes in the Simulink package have **Type** set to `FlatStructure`.
  - The storage class property **Type** is set to `Other`. For example, when the `GetSet` storage class in the Simulink package has **Type** set to `Other`.
  - The storage class property **Data access** is set to `Pointer`, indicating that the code declares imported symbols as pointer variables rather than simple variables. This property is accessible only when the storage class property **Data scope** is set to `Imported` or `Instance-specific`.

  In these cases, the report displays empty **Data Type** and **Dimension** fields.
- For outports, if you optimize memory, the code interface report cannot describe the associated memory (data type and dimensions). In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Code Generation** > **Data Type Replacement** pane of the Configuration Parameters dialog box. The data types in the report link to built-in data types rather than their specified replacement data types.

See Also

Related Examples

- “Analyze Generated Data Code Interface” on page 54-13
- “Generate Code Generation Report” on page 54-5

Static Code Metrics

In this section...
“Static Code Metrics” on page 54-26 “Static Code Metrics Analysis” on page 54-26 “View Static Code Metrics and Definitions Within the Generated Code” on page 54-28 “Static Code Metrics Report Limitations” on page 54-28

Static Code Metrics

The code generator performs static analysis on page 54-26 of the generated C or C++ code and provides these metrics in the **Static Code Metrics Report** section of the HTML Code Generation Report. When you place your cursor over a function or a variable in the generated code, you can also see metrics.

You can use the information in the report to:

- Find the number of files and lines of code in each file.
- Estimate the number of lines of code and stack usage per function.
- Compare the difference in terms of how many files, functions, variables, and lines of code are generated every time you change the model or MATLAB algorithm.
- Determine a target platform and allocation of RAM to the stack, based on the size of global variables plus the estimated stack size.
- Determine possible performance slow points, such as the largest global variables or the most costly call path in terms of stack usage.
- View the cyclomatic complexity of a function, which counts the number of linearly independent paths through a function.
- View the function call tree. Determine the longest call path to estimate the worst case execution timing.
- View how target functions, provided by the selected code replacement library, are used in the generated code.

For examples, see:

- “Generate Static Code Metrics Report for Simulink Model” on page 54-29
- “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” on page 54-33

Static Code Metrics Analysis

Static analysis of the generated code is performed only on the source code without executing the program. The results of the static code metrics analysis are included in the **Static Code Metrics** section of the HTML Code Generation Report. The report is not available if you generate a MEX function from MATLAB code.

Static analysis of the generated source code files:

- Uses the specified C data types. For Simulink models, you specify these data types in the **Hardware Implementation** > **Production hardware** pane of the Configuration Parameters dialog box. For code generation from MATLAB code, you specify them in the **Hardware** tab of the MATLAB Coder project settings dialog box or using a code generation configuration object. Actual object code metrics might differ due to target-specific compiler and platform settings.
- Includes custom code only if you specify it. For Simulink models, you specify custom code on the **Code Generation** > **Custom Code** pane in the model configuration. For code generation from MATLAB code, you specify it on the **Debugging** tab of the MATLAB Coder project settings dialog box or using a code generation configuration object. An error report is generated if the generated code includes platform-specific files not contained in the standard C run-time library.
- For Simulink models, includes the generated code from referenced models.
- Uses 1-byte alignment for all members of a structure for estimating global and local data structure sizes. The size of a structure is calculated by summing the sizes of all of its fields. This estimation represents the smallest possible size for a structure.
- Calculates the self stack size of a function as the size of local data within a function, excluding input arguments. The accumulated stack size of a function is the self stack size plus the maximum of the accumulated stack sizes of its called functions. For example, if the accumulated stacks sizes for the called functions are represented as `accum_size1...accum_sizeN`, then the accumulated stack size for a function is

  ```
  accumulated_stack_size = self_stack_size + max(accum_size1,...,accum_sizeN)
  ```

  For example consider this piece of generated code:

  ```
  /* Real-time model */
  RT_MODEL rtM_;
  RT_MODEL *const rtM = &rtM_;
  static void Debug_b(const real_T rtu_In1[6], const real_T rtu_In2[6], real_T
                      rtd_A[6], real_T rtd_B[6]);
  ```

  Now consider the code metrics shown below:

  Static Code Metrics reported stack sizes

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code
[-] local_demo_step	12	0	1
Debug_b	12	12	6
local_demo_initialize	0	0	0

  The accumulated stack size is 12 bytes because the self stack size of the function `Debug_b` is 12 bytes and the 4 input arguments do not contribute to the calculation of accumulated stack size.
- When estimating the stack size of a function, static analysis stops at the first instance of a recursive call. The **Function Information** table indicates when recursion occurs in a function call path. Code generation generates only recursive code for Stateflow event broadcasting and for graphical functions if it is written as a recursive function.
- Calculates the cyclomatic complexity of a function as the number of decisions plus one:

  ```
  CC = Number of decisions + 1
  ```

  The following constructs add a decision:

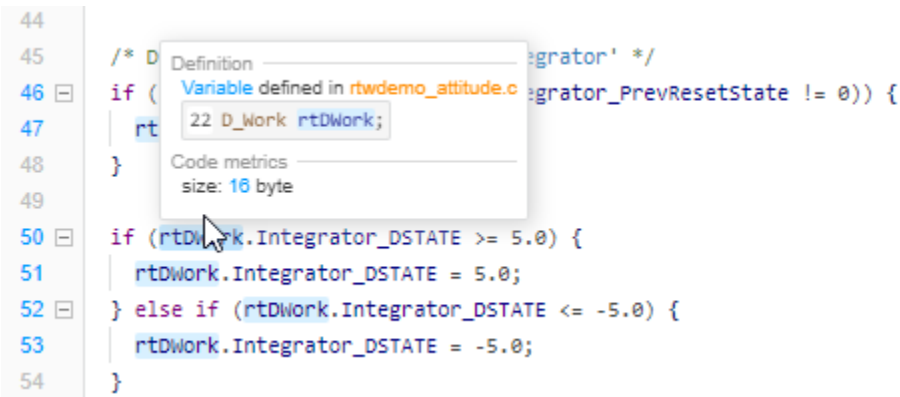
- If statement
- Else-If statement
- Switch statement (1 decision for each `case` branch)
- Loop statements: While, For, Do-while

Note Boolean operators in the preceding constructs do not add extra decisions.

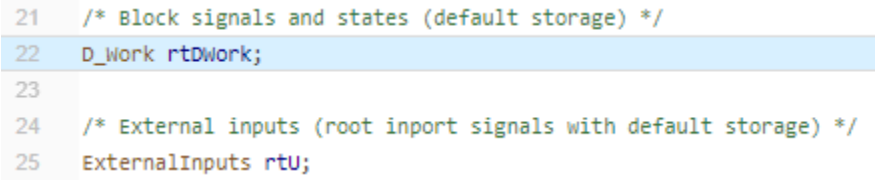
- Does not include `ert_main.c`, because you have the option to provide your own `main.c`.

View Static Code Metrics and Definitions Within the Generated Code

When you view code in the Code view or in the code generation report, place your cursor over the global variables and functions to see code metrics information.



The tooltip provides a link to the definition for the variable or function.



Static Code Metrics Report Limitations

Static code metrics are not available if the target configuration for a model results in generated code that:

- Includes a header file that is not generated by the model hierarchy and is not a system header file
- Uses a macro that is not recognized

Generate Static Code Metrics Report for Simulink Model

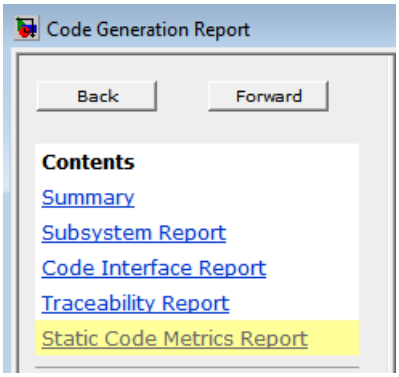
The **Static Code Metrics Report** is a section included in the HTML Code Generation Report. For more information on the static analysis of the generated code, see “Static Code Metrics Analysis” on page 54-26.

1. To generate the static code metrics report in the HTML Code Generation Report, open the Configuration Parameters dialog box for your model. On the **Code Generation** > **Report** pane, select the Generate static code metrics check box. If your model includes referenced models, select the **Generate static code metrics** check box in each referenced model’s configuration set. Otherwise, you cannot view a separate static code metrics report for a referenced model.

   Alternatively, after you generate code, generate only the static code metrics report by using the function `coder.report.generateCodeMetrics`.

2. Press **Ctrl+B** to build your model and generate the HTML code generation report. For more information, see “Generate Code Generation Report” on page 54-5.

3. If the HTML Code Generation Report is not already open, open the report. On the left navigation pane, in the **Contents** section, select **Static Code Metrics Report**.



   If you generated only the static code metrics report, open the report. By default, the static code metrics report file `metrics.html` is generated in the current folder.

4. To see the generated files and how many lines of code are generated per file, look at the **File Information** section.

1. File Information [hide]

[-] Summary (excludes ert_main.c)

Number of .c files :	2
Number of .h files :	4
Lines of code :	1,132
Lines :	1,958

[-] File details

File Name	Lines of Code	Lines	Generated On
fuel_rate_control.c	665	1,107	06/10/2016 4:17 PM
fuel_rate_control_data.c	253	342	06/10/2016 4:17 PM
rtwtypes.h	97	177	06/10/2016 4:17 PM
fuel_rate_control.h	88	249	06/10/2016 4:17 PM
fuel_rate_control_types.h	22	52	06/10/2016 4:17 PM
fuel_rate_control_private.h	7	31	06/10/2016 4:17 PM

5. Hover your cursor over column titles and some column values to see a description of the corresponding data.
6. If your model includes referenced models, the File information section includes a **Referenced Model** column. In this column, click the referenced model name to open its static code metrics report. If the static code metrics report is not available for a referenced model, specify the **Generate static code metrics** parameter in the referenced model's configuration set and rebuild your model.
7. To view the global variables in the generated code, their size, and the number of accesses, see the **Global Variables** section.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] fuel_rate_control_DW	23	175	78
[+] fuel_rate_control_B	21	42	26
[+] fuel_rate_control_U	16	27	24
[+] fuel_rate_control_M_	8	0*	0*
[+] fuel_rate_control_Y	4	4	3
Total	**72**	**248**	

* The global variable is not directly used in any function.

The **Reads/Writes** column displays the total number of read and write accesses to the global variable. The **Reads/Writes in a Function** column displays the maximum number of read and write accesses to the global variable within a function. You use this information is to estimate the benefit of turning on optimizations, which reduce the number of global references. For more information, see "Optimize Global Variable Usage" on page 78-2.

Click [+] to expand structures.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] fuel_rate_control_DW	23	175	78
DiscreteIntegrator_DSTATE	4	5	4
ThrottleTransient_states	4	4	3
DiscreteFilter_states	4	4	3
DiscreteFilter_states_i	4	4	3
[+] bitsForTID0	5	156	60
temporalCounter_i1	2	7	5
[-] fuel_rate_control_B	21	42	26
[+] es_o	16	23	18
fuel_mode	4	15	8
O2_normal	1	6	4
[-] fuel_rate_control_U	16	27	24
[+] sensors	16	27	24
[-] fuel_rate_control_M_	8	0*	0*
errorStatus	8	0	0
[-] fuel_rate_control_Y	4	4	3
fuel_rate	4	4	3
Total	**72**	**248**	

* The global variable is not directly used in any function.

8. To navigate from the report to the source code, click a global variable or function name. These names are hyperlinks to their definitions.
9. To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree** at the top of the table.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View:Call Tree | Table

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[-] **fuel_rate_control_step**	52	16	272	504	43
look2_iflf_linlca	36	36	55	100	12
[-] fuel_rate_control_Fail	4	4	67	103	13
fuel_rate_control_Fueling_Mode	0	0	166	240	19
fuel_rate_control_Fueling_Mode	0	0	166	240	19
[+] **fuel_rate_control_initialize**	0	0	43	67	1
fuel_rate_control_terminate	0	0	0	4	1

`ert_main.c` is not included in the code metrics analysis, therefore it is not shown in the call tree format. The **Complexity** column includes the cyclomatic complexity of each function.

10. To view the functions in a table format, click **Table**.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: Call Tree | Table

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
fuel_rate_control_Fail	fuel_rate_control_step (9)	4	4	67	103	13
fuel_rate_control_Fueling_Mode	fuel_rate_control_step fuel_rate_control_Fail (2)	0	0	166	240	19
fuel_rate_control_initialize		0	0	43	67	1
fuel_rate_control_step		52	16	272	504	43
fuel_rate_control_terminate		0	0	0	4	1
look2_iflf_linlca	fuel_rate_control_step (5)	36	36	55	100	12
memset	fuel_rate_control_initialize (2)	*Not available*	-	-	-	-

The second column, **Called By**, lists functions that call the function listed in the first column, using the following criteria:

- If a function is called by multiple functions, all functions are listed.
- If a function has no called function, this column is empty.

For example, `Fueling_Mode` is called by `Fail` and `fuel_rate_control_step`. The number of call sites is included in parentheses. `Fail` calls `Fueling_Mode` twice.

See Also

More About

- “Static Code Metrics Analysis” on page 54-26
- “Reports for Code Generation” on page 54-2
- “Generate Code Generation Report” on page 54-5
- “Configure and Generate Code Generation Report Programmatically” on page 54-7

Generating a Static Code Metrics Report for Code Generated from MATLAB Code

The static code metrics report contains the results of static analysis of the generated C/C++ code, including generated file information, number of lines, and memory usage. For more information, see “Static Code Metrics” on page 54-26. To produce a static code metrics report, you must use Embedded Coder to generate standalone C/C++ code and produce a code generation report. See “Code Generation Reports”.

By default, static code metrics analysis does not run at code generation time. Instead, if and when you want to run the analysis and view the results, click **Code Metrics** on the **Summary** tab of the code generation report.

Example Static Code Metrics Report

This example runs static code metrics analysis and examines a static code metrics report.

Create the example function `averaging_filter`.

```
function y = averaging_filter(x) %#codegen
% Use a persistent variable 'buffer' that represents a sliding window of
% 16 samples at a time.
persistent buffer;
if isempty(buffer)
    buffer = zeros(16,1);
end
y = zeros(size(x), class(x));
for i = 1:numel(x)
    % Scroll the buffer
    buffer(2:end) = buffer(1:end-1);
    % Add a new sample value to the buffer
    buffer(1) = x(i);
    % Compute the current average value of the window and
    % write result
    y(i) = sum(buffer)/numel(buffer);
end
```

Create sample data.

```
v = 0:0.00614:2*pi;
x = sin(v) + 0.3*rand(1,numel(v));
```

Enable production of a code generation report by using a configuration object for standalone code generation (static library, dynamically linked library, or executable program).

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

Alternatively, use the `codegen -report` option.

Generate code by using `codegen`. Specify the type of the input argument by providing an example input with the `-args` option. Specify the configuration object by using the `-config` option.

```
codegen averaging_filter -config cfg -args {x}
```

To open the code generation report, click **View report**.

To run the static code metrics analysis and view the code metrics report, on the **Summary** tab of the code generation report, click **Code Metrics**.

Explore the code metrics report.

1. To see the generated files and the number of lines of code per file, click **File Information**.

averaging_filter.m | Code Metrics ×

1. File Information [hide]

[-] Summary

Number of .c files : 3
Number of .h files : 5
Lines of code : 86
Lines : 281

[-] File details

File Name	Lines of Code	Lines
averaging_filter.c	24	65
rtwtypes.h	22	46
averaging_filter.h	9	30
averaging_filter_initialize.h	8	29
averaging_filter_terminate.h	8	29
averaging_filter_initialize.c	6	30
averaging_filter_terminate.c	5	30
averaging_filter_types.h	4	22

2. To see the global variables in the generated code, go to the **Global Variables** section.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
buffer	128	5	4
Total	**128**	**5**	

To navigate from the report to the source code, click a global variable name.

3. To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree**.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.
View:Call Tree | Table

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[+] _averaging_filter	136	136	14	24	3
_double_to_uint64_helper	8	8	6	8	1
[+] _averaging_filter_initialize	0	0	1	4	1
_averaging_filter_terminate	0	0	0	4	1

To navigate from the report to the function code, click a function name.

4. To view the functions in a table format, click **Table**.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.
View: Call Tree | Table

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
_averaging_filter		136	136	14	24	3
_averaging_filter_init	_averaging_filter_initialize	0	0	1	4	1
_averaging_filter_initialize		0	0	1	4	1
_averaging_filter_terminate		0	0	0	4	1

The second column, **Called By**, lists functions that call the function listed in the first column. If multiple functions call that function, all functions are listed. If no functions call that function, this column is empty.

Requirements for Running Static Code Metrics Analysis After Code Generation

By default, static code metrics analysis does not run at code generation time. Instead, you can run the analysis later by clicking **Code Metrics** in the code generation report. Running static code metrics analysis after code generation has these requirements and limitations:

- You must have Embedded Coder and use the platform that you used for code generation. Once you run the static code metrics analysis, you can open the code metrics report without Embedded Coder or open it on a different platform.
- If you make the code generation report read-only before you run the analysis, each time that you click **Code Metrics**, the analysis runs.

Running Static Code Metrics at Code Generation Time

If you want the code generator to run static code metrics analysis and produce the code metrics report at code generation time:

- In an Embedded Coder code generation configuration object, set `GenerateCodeMetricsReport` to `true`.
- In the MATLAB Coder app, on the **Debugging** tab, set **Static code metrics** to `Yes`.

See Also

More About

- “Code Generation Reports”

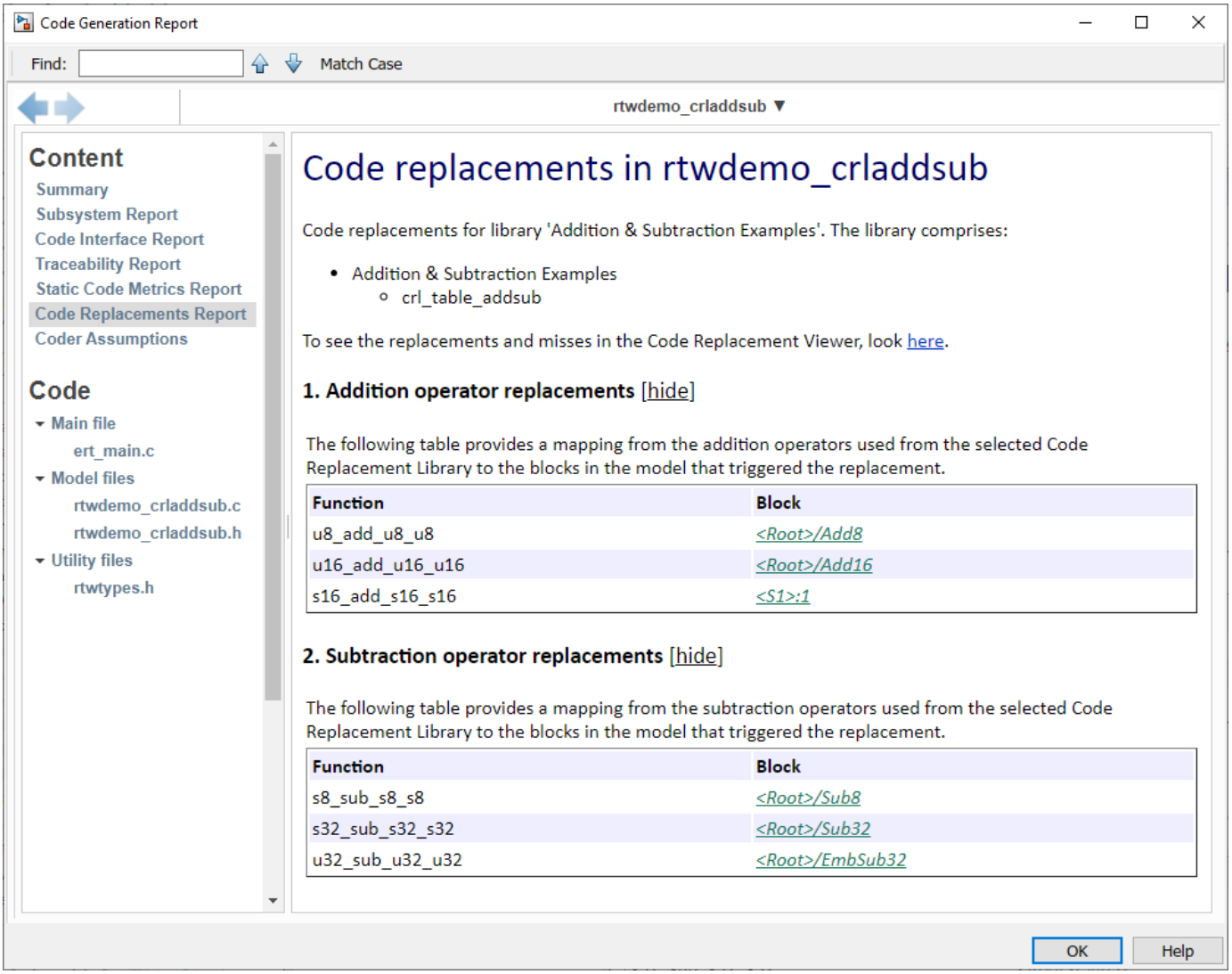
Analyze Code Replacements in Generated Code

When you select the check box **Summarize which blocks triggered code replacements** for an ERT-based model, a Code Replacements Report section is automatically included in the generated HTML report. The Code Replacements Report section documents the SIMD instruction sets and code replacement library (CRL) functions that were used for code replacements during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. To enable display of the Simulink block information, select the **Code Generation > Comments** check box **Include comments**. On the same pane, select either the **Simulink block comments** checkbox or the **Simulink block descriptions** check box if present, or both.

You can use the report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the block that triggered the replacement.

The figure below shows a Code Replacements Report generated for the CRL model `rtwdemo_crladdsub`. Each replacement function used is listed with a link to the block that triggered the replacement.



If you click a block path in the report, the block that triggered the replacement is highlighted in the model diagram. If the replacement was triggered by a Stateflow chart or a MATLAB function, a window opens to display the chart or function.

For more information, see “Verify Code Replacement Library” on page 72-55.

Document Generated Code with Simulink Report Generator

In this section...

"Generate Code for the Model" on page 54-38
"Open the Report Generator" on page 54-39
"Set Report Name, Location, and Format" on page 54-41
"Include Models and Subsystems in a Report" on page 54-42
"Customize the Report" on page 54-42
"Generate the Report" on page 54-43

The Simulink Report Generator software creates documentation from your model in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. This example shows one way to document a code generation project in Microsoft Word. The generated report includes:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Simulink Coder and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- System target file selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the source code

To adjust Simulink Report Generator settings to include custom code and then generate a report for a model, complete the following tasks:

1. "Generate Code for the Model" on page 54-38
2. "Open the Report Generator" on page 54-39
3. "Set Report Name, Location, and Format" on page 54-41
4. "Include Models and Subsystems in a Report" on page 54-42
5. "Customize the Report" on page 54-42
6. "Generate the Report" on page 54-43

A Simulink Report Generator license is required for the following report formats: PDF, RTF, Microsoft Word, and XML. For more information on generating reports in these formats, see the Simulink Report Generator documentation.

Generate Code for the Model

Before you use the Report Generator to document your project, generate code for the model.

1. In the MATLAB Current Folder browser, navigate to a folder where you have write access.
2. Create a working folder from the MATLAB command line by typing:

```
mkdir report_ex
```

3. Make `report_ex` your working folder:

   ```
   cd report_ex
   ```

4. Open the `slexAircraftExample` model. In the Command Window, enter:

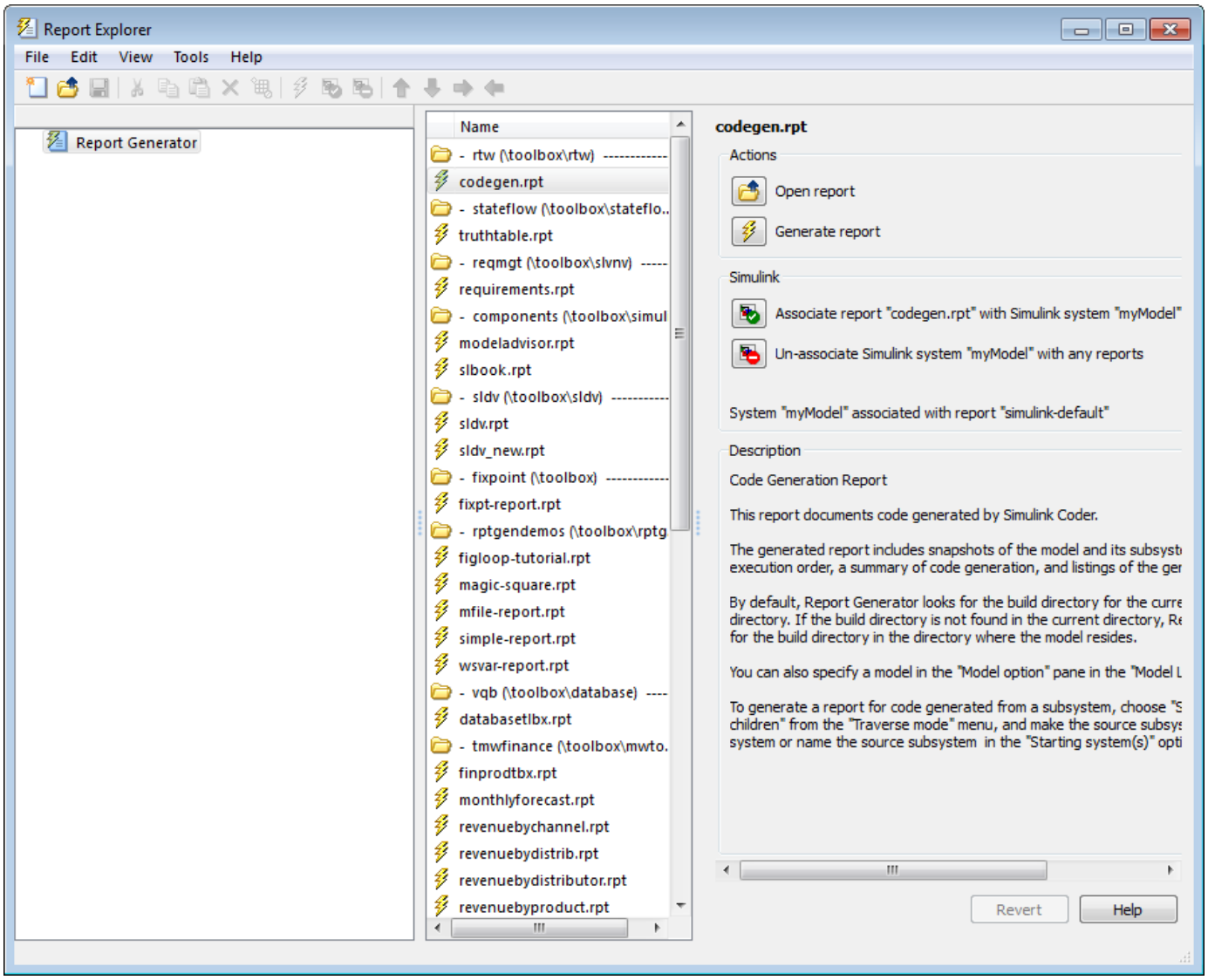
   ```
   openExample('slexAircraftExample')
   ```

5. On the **Simulation** tab, in the **Save** list, select **Save as**. Navigate to the working folder, `report_ex`, and save a copy of the `slexAircraftExample` model as `myModel`.
6. Open the Configuration Parameters dialog box. Set the **Solver** > **Solver selection** > **Type** parameter to `Fixed-step`.
7. Select the **Code Generation** pane. Select **Generate code only**.
8. Click **Apply**.
9. In the model window, press **Ctrl+B**. The build process generates code for the model.

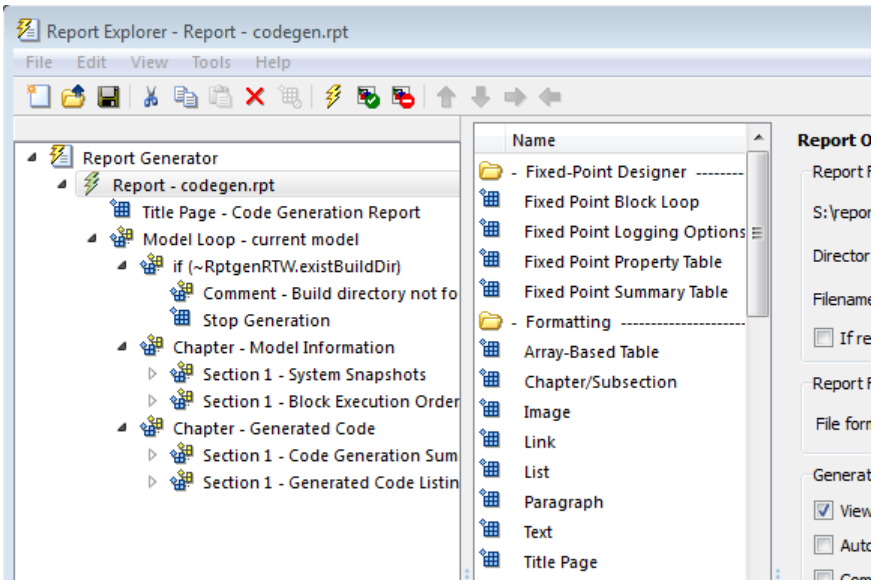
Open the Report Generator

After you generate the code, open the Report Generator.

1. Open the Report Generator app. In the model diagram window, in the **Apps** gallery, click **Report Generator**.
2. In the Report Explorer window, in the options pane (center), click the folder **rtw (\toolbox\rtw)**. Click the setup file that it contains, **codegen.rpt**.



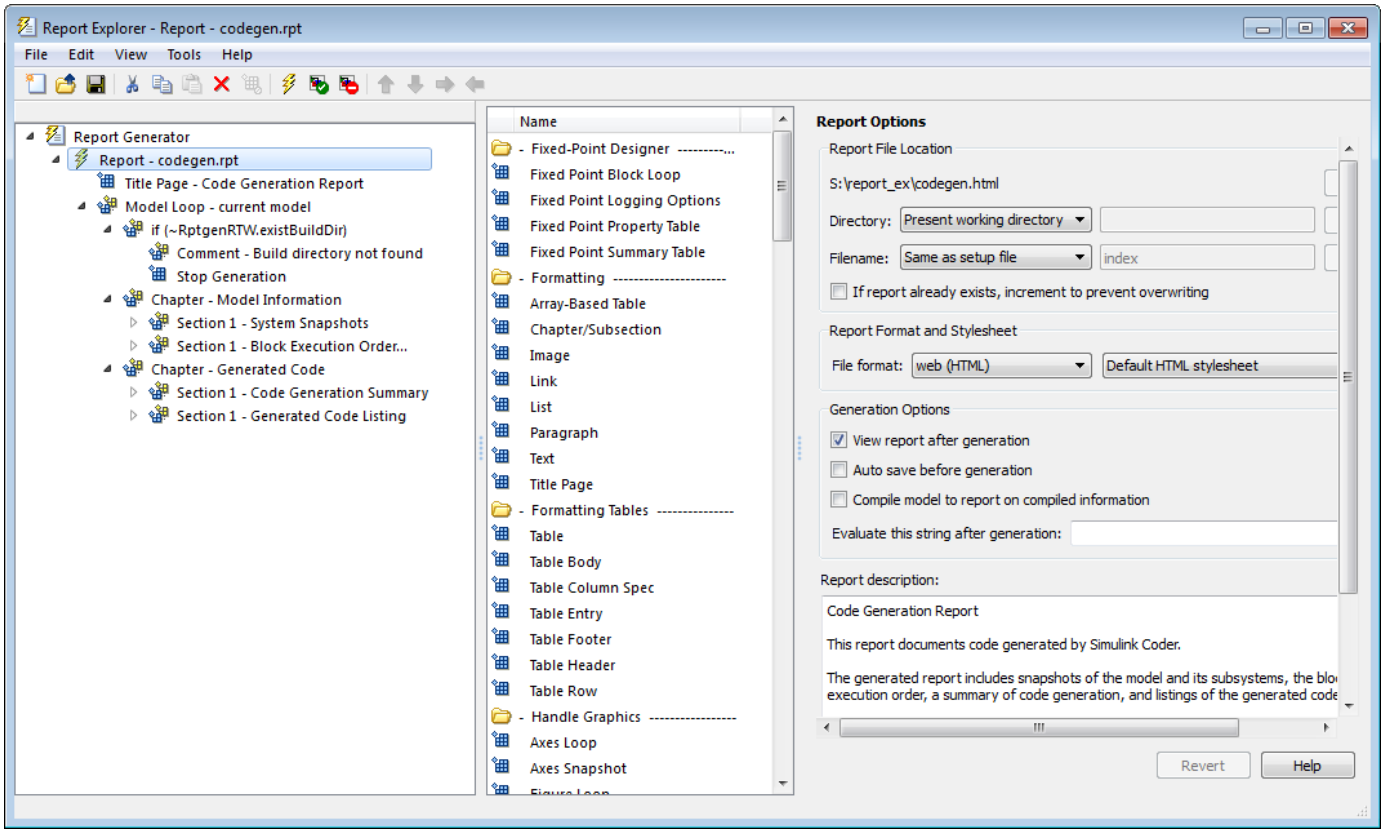
3 Double-click **codegen.rpt** or select it and click the **Open report** button. The Report Explorer displays the structure of the setup file in the outline pane (left).



Set Report Name, Location, and Format

Before generating a report, you can specify report output options, such as the folder, file name, and format. For example, to generate a Microsoft Word report named `MyCGModelReport.rtf`:

1 In the properties pane, under **Report Options**, review the options listed.



2 Leave the **Directory** field set to `Present working directory`.

3 For **Filename**, select `Custom:` and replace `index` with the name `MyModelCGReport`.

4 For **File format**, specify `Rich Text Format` and replace `Standard Print` with `Numbered Chapters & Sections`.

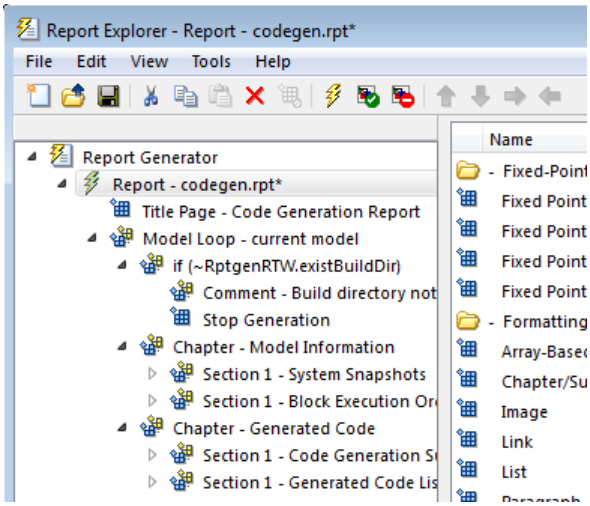
Include Models and Subsystems in a Report

Specify the models and subsystems that you want to include in the generated report by setting options in the Model Loop component.

1 In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.

2 If not already selected, select `Current block diagram` for the **Model name** option.

3 In the outline pane, click **Report - codegen.rpt***.



Customize the Report

After specifying the models and subsystems to include in the report, you can customize the sections included in the report.

1 In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two report components.

2 Expand the node **Section 1 — Code Generation Summary**.

3 Select **Code Generation Summary**. Options for the component are displayed in the properties pane.

4 Click **Help** to review the report customizations that you can make with the Code Generation Summary component. For this example, do not customize the component.

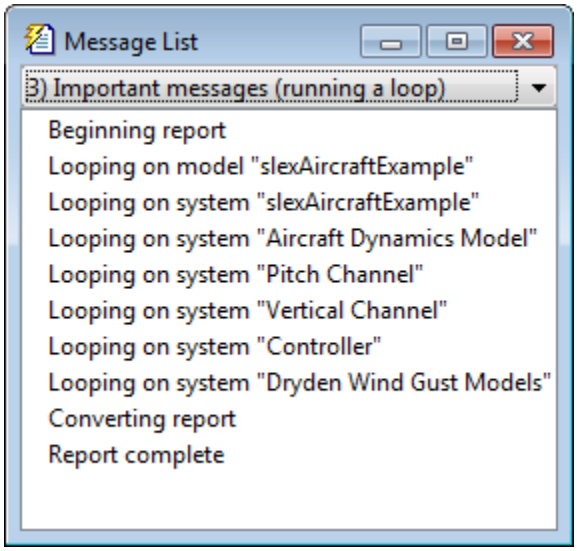
5 In the Report Explorer window, expand the node **Section 1 — Generated Code Listing**.

6 Select **Import Generated Code**. Options for the component are displayed in the properties pane.

7 Click **Help** to review the report customizations that you can make with the Import Generated Code component.

Generate the Report

After you adjust the report options, from the **Report Explorer** window, generate the report by clicking **File** > **Report**. A **Message List** dialog box opens, which displays messages that you can monitor as the report is generated. Model snapshots also appear during report generation. The **Message List** dialog box might be hidden behind other dialog boxes.



When the report is complete, open the report, `MyModelCGReport.rtf` in the folder `report_ex` (in this example).

See Also

More About

- “Reports for Code Generation” on page 54-2
- “Generate Code Generation Report” on page 54-5
- “Configure and Generate Code Generation Report Programmatically” on page 54-7

Get Code Description of Generated Code

You can use the code descriptor API to obtain metadata about the generated code. For each model build, the code generator, by default, creates a `codedescriptor.dmr` file in the build folder. When simulating the model in Accelerator and Rapid Accelerator modes, the `codedescriptor.dmr` is not generated.

You can use the code descriptor API once the code is generated. Use the code descriptor API to describe these items in the generated code:

- Data Interfaces: inports, outports, parameters, data stores, and internal data.
- Function Interfaces: initialize, output, update, and terminate.
- Run-time information of the data and function interfaces, such as timing requirements of each interface entity.
- Model hierarchy information and code description of referenced models.

Get Data Interface Information

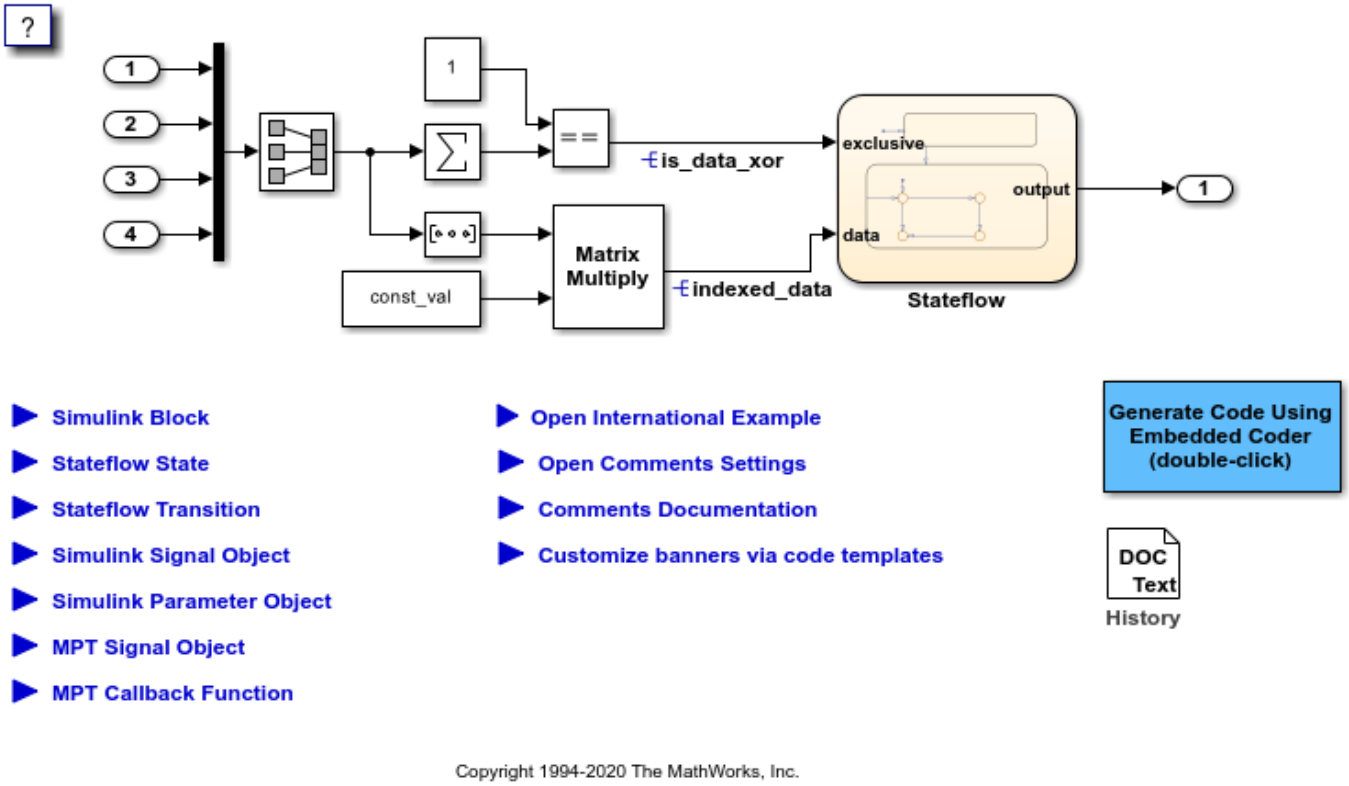
The `coder.descriptor.DataInterface` object describes various properties for a specified data interface in the generated code. In the model `rtwdemo_comments`, there are four inports, one outport, and a tunable external parameter. For more information about the data interfaces in your model, use the `coder.codedescriptor.CodeDescriptor` object and its methods.

1. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

2. Open and build the model.

```
open_system('rtwdemo_comments');
evalc('slbuild(''rtwdemo_comments'')');
```



3. Create a `coder.codedescriptor.CodeDescriptor` object for the required model by using the `getCodeDescriptor` function.

```
codeDescriptor = coder.getCodeDescriptor('rtwdemo_comments');
```

4. To obtain a list of all the data interface types in the generated code, use the `getDataInterfaceTypes` method.

```
dataInterfaceTypes = codeDescriptor.getDataInterfaceTypes()
```

```
dataInterfaceTypes =

  5x1 cell array

    {'Inports'                 }
    {'Outports'                }
    {'Parameters'              }
    {'ExternalParameterObjects'}
    {'InternalData'            }
```

To obtain a list of all the supported data interfaces, use the `getAllDataInterfaceTypes` method.

5. To obtain more information about a particular data interface type, use the `getDataInterfaces` method.

```
dataInterface = codeDescriptor.getDataInterfaces('Inports');
```

This method returns properties of the Inport blocks in the generated code.

6. Because this model has four inports, `dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first Inport of the model by accessing the first location in the array.

```
dataInterface(1)
```

```
ans =

  DataInterface with properties:
              Type: [1x1 coder.descriptor.types.Type]
               SID: 'rtwdemo_comments:99'
     GraphicalName: 'In1'
       VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
            Timing: [1x1 coder.descriptor.TimingInterface]
              Unit: ''
             Range: [1x0 coder.descriptor.Range]
```

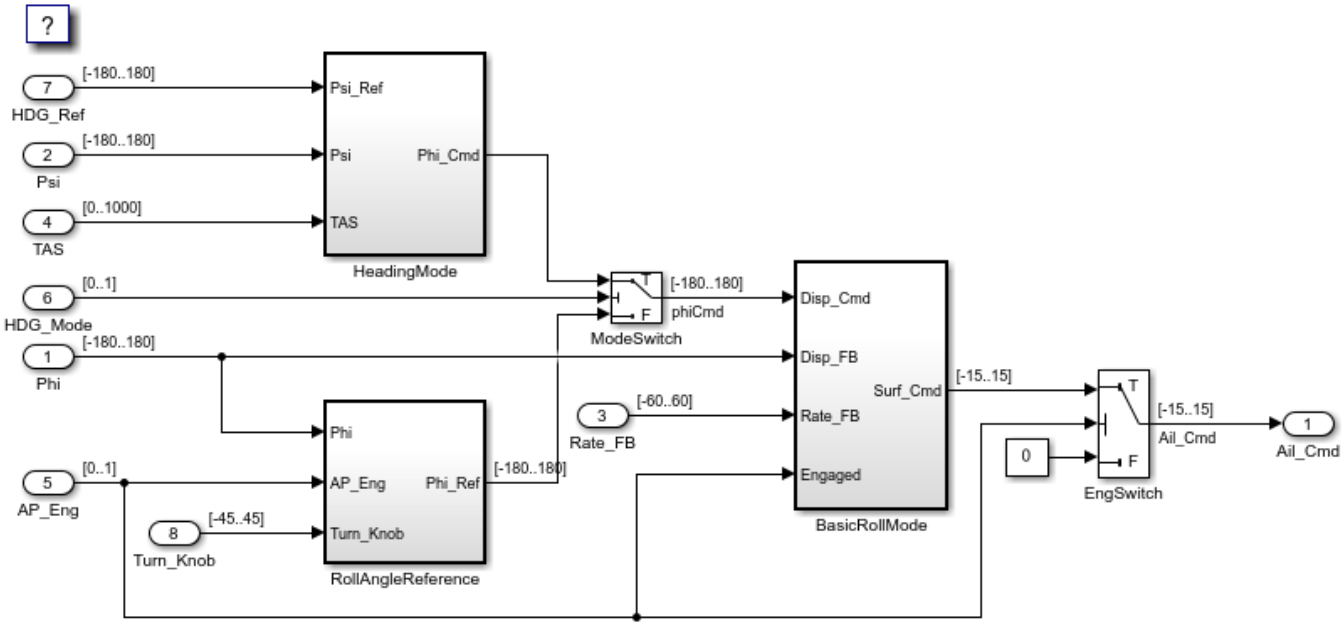
Get Function Interface Information

The function interfaces are the entry-point functions in the generated code. In the model `RollAxisAutopilot`, the entry-point functions are `model_initialize`, `model_step`, and `model_terminate`. For more information about the function interfaces in your model, use the `coder.descriptor.FunctionInterface` object.

2. Open and build the model.

```
cd(currentDir)
open_system('RollAxisAutopilot')
slbuild('RollAxisAutopilot')
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
========================================================================================================
RollAxisAutopilot  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.721s
```



3. Create a `coder.codedescriptor.CodeDescriptor` object for the required model by using the `getCodeDescriptor` function.

```
codeDescriptor = coder.getCodeDescriptor('RollAxisAutopilot');
```

4. To obtain a list of all the function interface types in the generated code, use the `getFunctionInterfaceTypes` method.

```
functionInterfaceTypes = codeDescriptor.getFunctionInterfaceTypes()
```

```
functionInterfaceTypes =

  2x1 cell array

    {'Initialize'}
    {'Output'    }
```

To obtain a list of all the supported function interfaces, use the `getAllFunctionInterfaceTypes` method.

5. To obtain more information about a particular function interface type, use the `getFunctionInterfaces` method.

```
functionInterface = codeDescriptor.getFunctionInterfaces('Initialize')
```

```
functionInterface =

  FunctionInterface with properties:
        Prototype: [1x1 coder.descriptor.types.Prototype]
```

```
     ActualReturn: [1x0 coder.descriptor.DataInterface]
      VariantInfo: [1x0 coder.descriptor.VariantInfo]
    FunctionOwner: [1x0 coder.descriptor.TypedRegion]
           Timing: [1x1 coder.descriptor.TimingInterface]
       ActualArgs: [1x0 coder.descriptor.DataInterface Sequence]
```

6. You can further expand on the properties to obtain detailed information. To get the function return value, name, and arguments:

```
functionInterface.Prototype
```

```
ans =

  Prototype with properties:
          Name: 'RollAxisAutopilot_initialize'
        Return: [1x0 coder.descriptor.types.Argument]
    HeaderFile: 'RollAxisAutopilot.h'
    SourceFile: 'RollAxisAutopilot.c'
     Arguments: [1x0 coder.descriptor.types.Argument Sequence]
```

Get Model Hierarchy Information

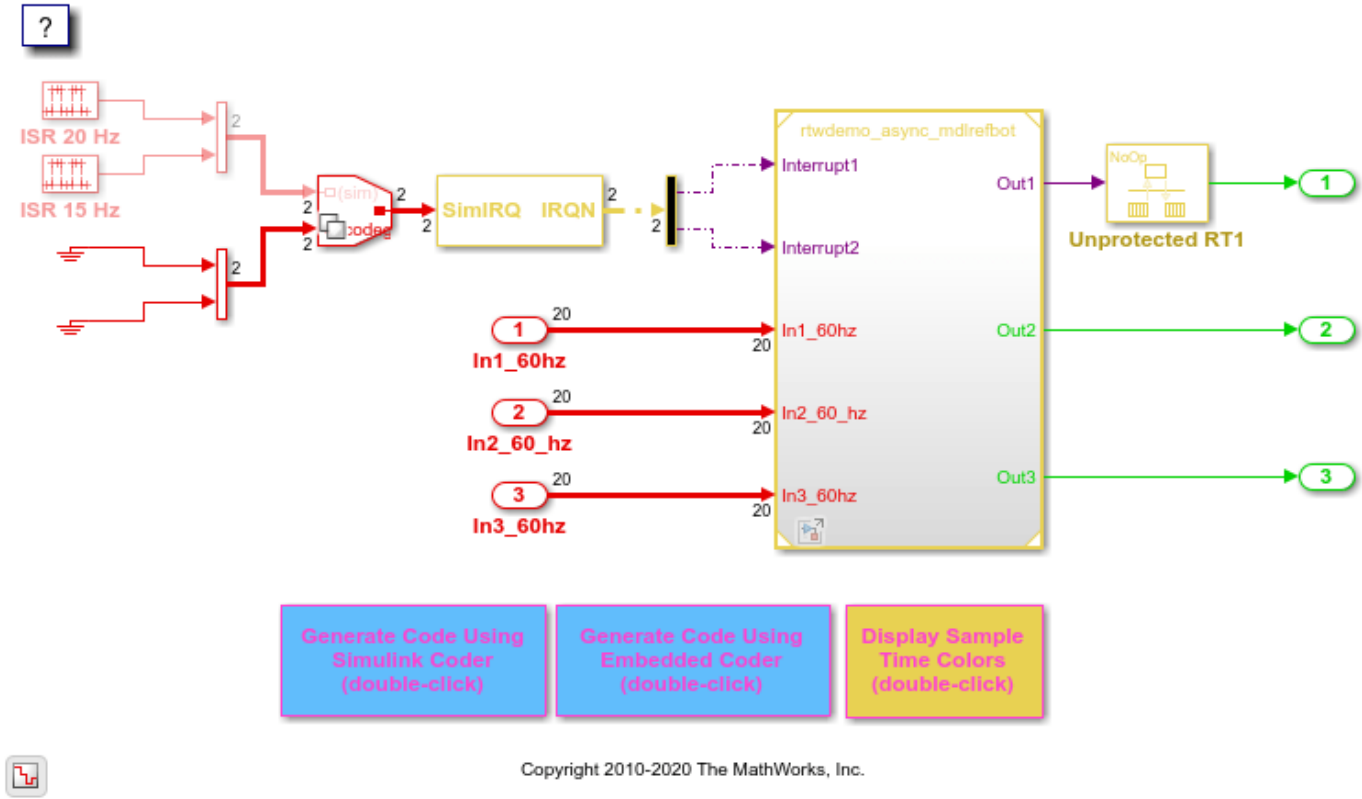
Use the `coder.codedescriptor.CodeDescriptor` object to get the entire model hierarchy information. The model `rtwdemo_async_mdlreftop` has model `rtwdemo_async_mdlrefbot` as the referenced model.

1. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

2. Open and build the model.

```
open_system('rtwdemo_async_mdlreftop');
evalc('slbuild(''rtwdemo_async_mdlreftop'')');
```



3. Create a `coder.codedescriptor.CodeDescriptor` object for the required model by using the `getCodeDescriptor` function.

```
codeDescriptor = coder.getCodeDescriptor('rtwdemo_async_mdlreftop');
```

4. Get a list of all the referenced models by using the `getReferencedModelNames` method.

```
refModels = codeDescriptor.getReferencedModelNames()
```

```
refModels =

  1x1 cell array

    {'rtwdemo_async_mdlrefbot'}
```

5. To obtain the `coder.codedescriptor.CodeDescriptor` object for the referenced model, use the `getReferencedModelCodeDescriptor` method.

```
refCodeDescriptor = codeDescriptor.getReferencedModelCodeDescriptor('rtwdemo_async_mdlrefbot');
```

You can now use the `refCodeDescriptor` object to obtain more information about the referenced model by using all the available methods in the Code Descriptor API.

See Also

Related Examples

- “Analyze Generated Data Code Interface” on page 54-13
- `coder.codedescriptor.CodeDescriptor`
- “Get Generated Type and Name of Inport Structures” on page 54-51

Get Generated Type and Name of Inport Structures

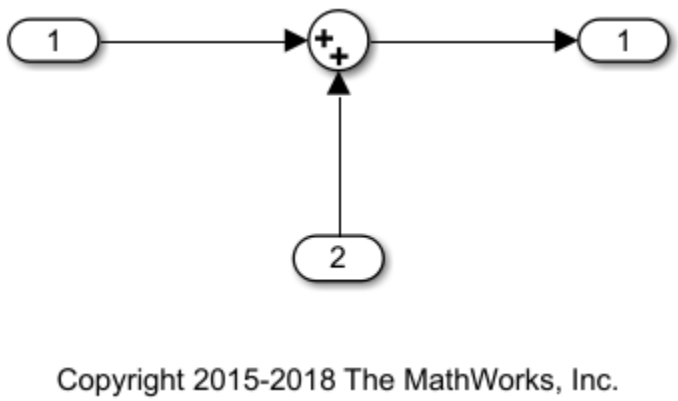
To obtain the generated type and name of inport and outport structures, use the code descriptor API. The code generator stores this information within a `DataImplementation` object field on a `DataInterface` object when it generates the code.

To interface your generated code with other components of your application, use the code descriptor API.

Open and Build Model

Open the model.

```
open_system('rtwdemo_sum');
```



Build the model.

```
evalc('slbuild(''rtwdemo_sum'')');
```

The generated code includes a structure representing the inports.

```
/* External inputs (root inport signals with default storage) */
ExtU rtU;
```

Get Code Descriptor and Inport Structure Names and Types

Get the code descriptor object.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_sum')
```

```
codeDescObj 
  CodeDescriptor with properties:

    ModelName: 'rtwdemo_sum'
     BuildDir: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\4\tp50229dff\simulinkcoder-ex78186698\rtwde
```

Get the list of data interface types in the generated code.

```
dataInterfaceTypes = codeDescObj.getDataInterfaceTypes()
```

```
dataInterfaceTypes = 2x1 cell
    {'Inports' }
    {'Outports'}
```

Get the `DataInterface` object that represents the inports.

```
inports = codeDescObj.getDataInterfaces('Inports')
```

```
inports =
  1x2 DataInterface array with properties:

    Type
    SID
    GraphicalName
    VariantInfo
    Implementation
    Timing
    Unit
    Range
```

Get the `Implementation` object that represents the first inport. In this example, the inports are each represented as a `StructExpression`, which is a subclass of `DataImplementation`.

```
impl = inports(1).Implementation
```

```
impl =
  StructExpression with properties:
                 Type: [1x1 coder.descriptor.types.Type]
           BaseRegion: [1x1 coder.descriptor.TypedRegion]
    ElementIdentifier: 'In1'
              Variant: ''
```

Get the `Type` of the first inport. The `Type` of the inport is `real_T`.

```
implType = impl.Type
```

```
implType =
  Double with properties:
    Identifier: 'real_T'
          Name: 'double'
      ReadOnly: 0
      Volatile: 0
    WordLength: 64
    Signedness: 1
```

Get the `BaseRegion` of the first inport. `BaseRegion` represents the structure that contains the inport. The `BaseRegion` is `rtU`.

```
baseRegion = impl.BaseRegion
```

```
baseRegion =
  Variable with properties:
                 Type: [1x1 coder.descriptor.types.Type]
           Identifier: 'rtU'
             VarOwner: 'rtwdemo_sum'
```

```
    StorageSpecifier: ''
```

Get the `Type` of the `BaseRegion`. The `Type` of the `BaseRegion` is `ExtU`.

```
baseRegionType = baseRegion.Type
```

```
baseRegionType = 
  Struct with properties:
    Identifier: 'ExtU'
          Name: 'ExtU'
      ReadOnly: 0
      Volatile: 0
      Elements: [1x2 coder.descriptor.types.AggregateElement Sequence]
      Checksum: [1x0 Integer Sequence]
```

Close Model

```
bdclose('rtwdemo_sum');
```

See Also

Related Examples

- “Get Code Description of Generated Code” on page 54-44

Get Metadata About Service Interface

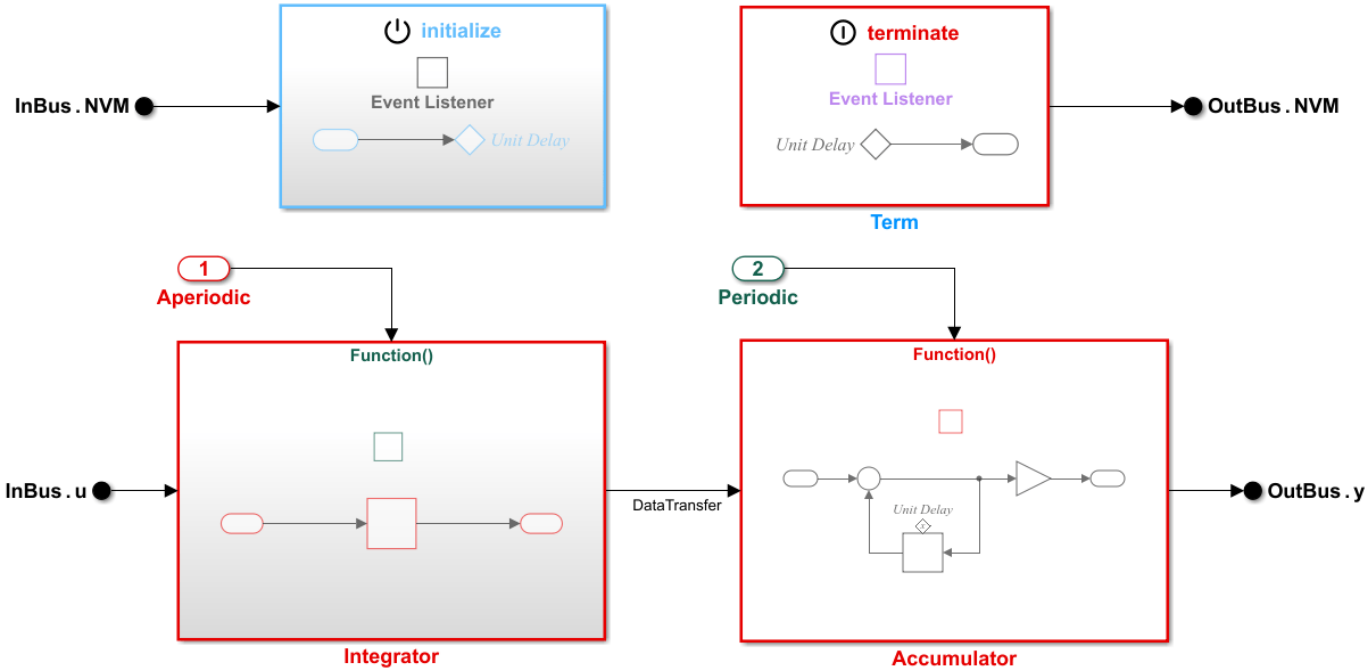
This example shows how you can use the code descriptor programming interface to retrieve metadata about the code interfaces generated for a model. You can use this metadata to declare and define your target platform service functions.

Use the Embedded Coder Dictionary to create code interface definitions for services that your generated code uses. For more information, see “Create a Service Interface Configuration” on page 32-11.

In this example, you build code for a model, create a `coder.codedescriptor.CodeDescriptor` object, and then use the various methods in the code descriptor programming interface.

Open example model `ComponentDeploymentFcn`. This model uses the shared Embedded Coder Dictionary `ComponentDeploymentCoderDictionary.sldd`.

```
model = 'ComponentDeploymentFcn';
open_system(model);
```



To use the code descriptor programming interface, build the model and create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
evalc('slbuild(model)');
codeDescObj = coder.getCodeDescriptor(model);
```

Get Service Function Declaration

1. Retrieve the function prototype for a specified service function by using the `getServiceFunctionPrototype` method. You can locate the name of a service function in the code generation report. To open the report, in the **Embedded Coder** app, on **C Code** tab, click the **Open**

Report button. In the report, select the **Code Interface Report** tab and view the service functions listed under **Service Interface** section.

```
serviceFunctionName = 'set_CD_integrator_DataTransfer';
serviceFunctionPrototype = getServiceFunctionPrototype(codeDescObj, ...
    serviceFunctionName)
```

```
serviceFunctionPrototype = 
  Prototype with properties:
          Name: 'set_CD_integrator_DataTransfer'
        Return: [1x1 coder.descriptor.types.Argument]
    HeaderFile: ''
    SourceFile: ''
     Arguments: [1x0 coder.descriptor.types.Argument Sequence]
```

The code generator returns a `coder.descriptor.types.Prototype` object.

2. Retrieve the function declaration of the specified service function by using the `getServiceFunctionDeclaration` method.

```
serviceFunctionDeclaration = getServiceFunctionDeclaration(codeDescObj, ...
    serviceFunctionPrototype)
```

```
serviceFunctionDeclaration = 
'real_T * set_CD_integrator_DataTransfer(void)'
```

The code generator returns the function declaration as produced in the generated code. You can use this function declaration to generate your own service header files to ease component code integration with target platform services.

Get Service Interface Information

Retrieve the service interface object, also referred as a code configuration object, by using the `getServices` method.

```
serviceObj = codeDescObj.getServices();
```

The code generator returns a `coder.descriptor.ServiceInterface` object. Use this object to retrieve metadata about the service interface configuration that you define in the shared Embedded Coder Dictionary.

You can retrieve information of these types of service interface by specifying the corresponding enumerated values as input:

- Sender and Receiver — `coder.descriptor.Services.SenderReceiver`
- Data Transfer — `coder.descriptor.Services.DataTransfer`
- Timer — `coder.descriptor.Services.Timer`
- Measurement — `coder.descriptor.Services.Measurement`
- ParameterTuning — `coder.descriptor.Services.ParameterTuning`
- Parameter Argument Tuning — `coder.desciptor.Services.ParameterArgumentTuning`

For example, to retrieve information about the data transfer service interface by using the `getServiceInterface` method, use these commands:

```
serviceInterfaceType = coder.descriptor.Services.DataTransfer;
serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)
```

```
serviceInterfaceTypeObj =
  DataTransferServiceInterface with properties:
    DataTransferElements: [1x1 coder.descriptor.DataTransferElement Sequence]
```

The code generator returns a `coder.descriptor.DataTransferServiceInterface` object. Use this object to further retrieve metadata about `coder.descriptor.DataTransferElement` and `coder.descriptor.DataTransferFunction` objects.

Similarly, you can get information for the `coder.descriptor.SenderReceiverServiceInterface` and `coder.descriptor.TimerServiceInterface` objects.

Get Service Function Information

Use the various methods of the `coder.desriptor.ServiceInterface` class to retrieve information about service functions.

- To retrieve the name of the header file that contains the service interface prototypes, use the `getServicesHeaderFileName` method. The code generator returns a character vector that represents the header file name.

```
serviceHeaderFile = getServicesHeaderFileName(serviceObj)
```

```
serviceHeaderFile =
'services.h'
```

- To retrieve a list of entry-point functions that call a specified service function, use the `getCallableFunctionsThatCallServiceFunction` method. The code generator returns the list of entry-point functions.

```
serviceFunctionName = 'set_CD_integrator_DataTransfer';
entryPointFunctions = getCallableFunctionsThatCallServiceFunction(serviceObj, serviceFunctionNam
```

```
entryPointFunctions = 1x1 cell array
    {'CD_integrator'}
```

- To retrieve a list of service functions that call a specified entry-point function, use the `getCalledServiceFunctions` method. The code generator returns the `coder.descriptor.ServiceFunctions` object that contains a representation of the number of service functions of each type present in the generated code.

```
entryPointFunctionName = 'CD_integrator';
serviceFunctions = getCalledServiceFunctions(serviceObj, entryPointFunctionName)
```

```
serviceFunctions =
  ServiceFunctions with properties:
        ReceiverFunctions: [1x1 StdString Set]
          SenderFunctions: [1x0 StdString Set]
    DataTransferFunctions: [1x1 StdString Set]
           TimerFunctions: [1x1 StdString Set]
```

- To retrieve information about the data communication method, as specified in the Embedded Coder Dictionary, for a specified service function, use the `getServiceDataCommMethod` method. The code generator returns an enumerated value of type `DataCommunicationMethodEnum`.

```
serviceFunctionName = 'get_CD_accumulator_DataTransfer';
dataCommMethod = getServiceDataCommMethod(serviceObj, serviceFunctionName)
```

```
dataCommMethod =
  DataCommunicationMethodEnum enumeration

    OutsideExecution
```

See Also

Related Examples

- “Service Interfaces” on page 3-21
- “Create a Service Interface Configuration” on page 32-11
- “Deploy Export-Function Component Configured for C Service Interface Code Generation” on page 3-89

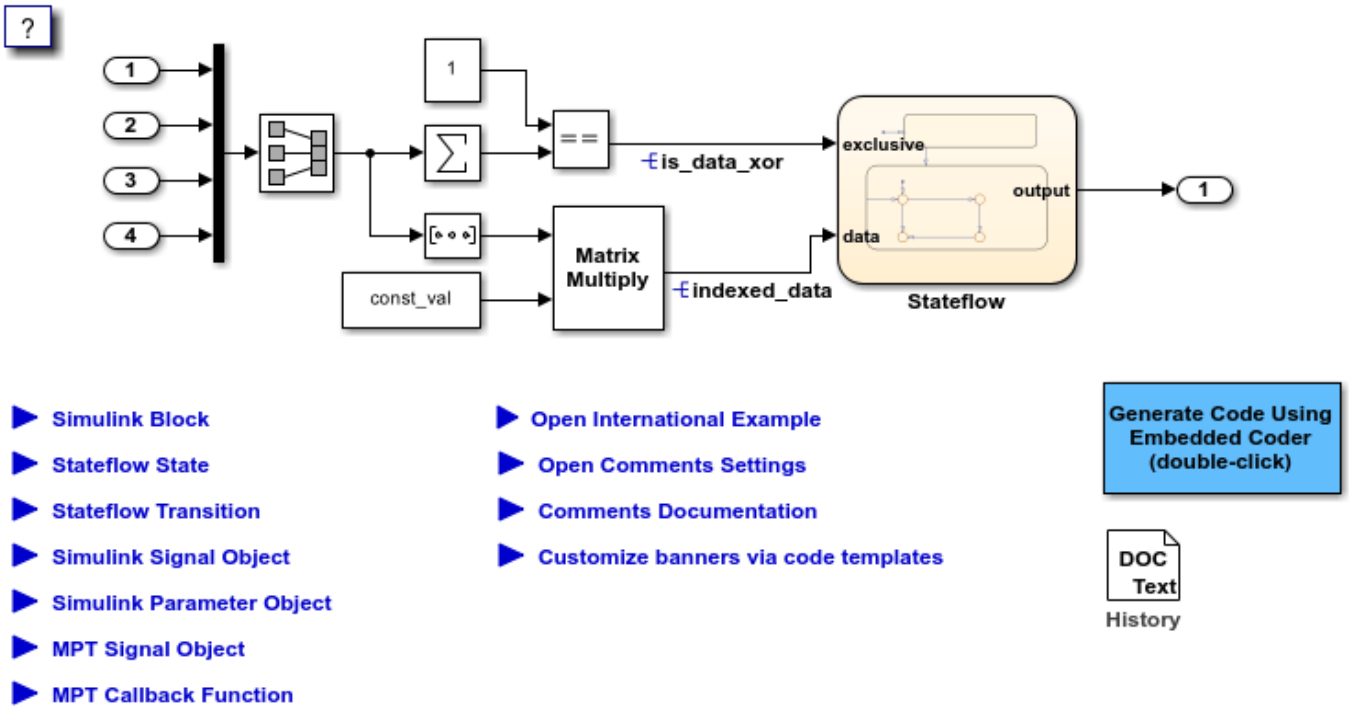
Code Appearance in Embedded Coder

- “Customize Code Comments to Enhance Readability and Traceability” on page 55-2
- “Add Custom Comments for Variables in the Generated Code” on page 55-3
- “Add Global Comments in the Generated Code” on page 55-5
- “Customize Generated Identifier Naming Rules” on page 55-10
- “Identifier Format Control” on page 55-17
- “Control Name Mangling in Generated Identifiers” on page 55-25
- “Avoid Identifier Name Collisions with Referenced Models” on page 55-26
- “Maintain Traceability for Generated Identifiers” on page 55-27
- “Exceptions to Identifier Formatting Conventions” on page 55-28
- “Identifier Format Control Parameters Limitations” on page 55-29
- “Control Parentheses in Generated Code” on page 55-31
- “Optimize Code by Reordering Commutable Operands” on page 55-33
- “Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements” on page 55-35
- “Replace Multiplication by Powers of Two with Signed Bitwise Shifts” on page 55-38
- “Generate Code Containing Right Shifts on Signed Integers” on page 55-40
- “Control Cast Expressions in Generated Code” on page 55-42
- “Customize Code Organization and Format” on page 55-46
- “Specify Templates For Code Generation” on page 55-48
- “Code Generation Template (CGT) Files” on page 55-49
- “Format Generated Code Files by Using Templates” on page 55-53
- “Custom File Processing (CFP) Templates” on page 55-55
- “Change the Organization of a Generated File” on page 55-58
- “Customize Generated File Names” on page 55-60
- “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 55-62
- “Comparison of a Template and Its Generated File” on page 55-68
- “Code Template API Summary” on page 55-71
- “Generate Custom File and Function Banners” on page 55-74
- “Template Symbols and Rules” on page 55-81
- “Annotate Code for Justifying Polyspace Checks” on page 55-86
- “Enhance Readability of Code for Flow Charts” on page 55-95
- “Enhance Code Readability for MATLAB Function Blocks” on page 55-105
- “Improve Data Coherency in Generated Code” on page 55-111

Customize Code Comments to Enhance Readability and Traceability

This example shows how to add comments to numerous types of objects in Simulink® and Stateflow®.

```
model='rtwdemo_comments';
open_system(model)
```




```
% Cleanup
rtwdemoclean;
close_system(model,0)
```

See Also

More About

- “Model Configuration Parameters: Comments”

Add Custom Comments for Variables in the Generated Code

To control code generation options for signals, states, and parameters in a model, you can create data objects in a workspace or data dictionary. You can generate comments in the code that help you to document the purpose and properties of the data in each object. Associate handwritten comments with each object, or write a function that generates comments based on the properties of the object.

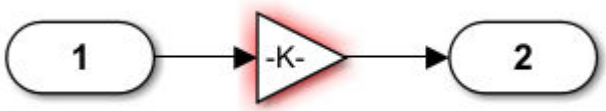
For more information about data objects, see “Data Objects”.

Embed Handwritten Comments for Signals or Parameters

To embed handwritten comments in the generated code near the definition of a signal, state, or parameter:

1. Open a Simulink model.

   ```
   openExample('ex_param_array)';
   ```

2. Create a data object to represent a signal, state, or parameter. You can use a data object from any package such as Simulink. For example, use a data object of the classes `Simulink.Signal` or `Simulink.Parameter`, which are defined in the package `Simulink`.

   ```
   myParam = Simulink.Parameter(15.23);
   ```

3. In the model, under the **PREPARE** section select **Property Inspector**. Click the target Gain block. Set its parameter value to myParam.
4. Set the storage class of the data object so that optimizations do not eliminate the signal or parameter from the generated code. For example, use the storage class `ExportedGlobal`.

   ```
   myParam.StorageClass = 'ExportedGlobal';
   ```

5. Set the `Description` property of the object. The description that you specify appears in the generated code as lines of comments.

   ```
   myParam.Description = 'This parameter represents multiplication';
   ```

6. In the **Model Configuration Parameters** select **Simulink data object descriptions**.
7. Generate code from the model. In the code, the data object description appears near the definition of the corresponding variable.

   ```
   /* Exported block parameters */
   real_T myParam = 15.23;     /* Variable: myParam
                                * Referenced by: '<S1>/Gain'
                                * This parameter represents multiplication.
                                */
   ```

Generate Dynamic Comments Based on Data Properties

You can generate dynamic comments that include the properties of a data object such as data type, units, and dimensions. If you change the properties of the data object in Simulink, the code generator

maintains the accuracy of the comments. For example, this comment displays some of the property values for a data object named `MAP` for the Simulink model in the previous example:

1. Create a data object from the package `mpt` and apply a custom storage class to the object. The default storage class is the custom storage class `Global (Custom)`.

   ```
   myParam = mpt.Parameter;
   ```

   To generate dynamic comments, you must use a data object from the package `mpt` and you must apply a custom storage class to the object.
2. Write a MATLAB or TLC function that generates the comment text. For an example MATLAB function, see the function *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwdemo_comments_mptfun.m`.

   The function must accept three input arguments that correspond to `objectName`, `modelName`, and `request`. If you write a TLC file, you can use the library function `LibGetSLDataObjectInfo` to get the property values of the data object.
3. Save the function as a MATLAB file or a TLC file. Place the file in a folder that is on your MATLAB path.
4. In the **Model Configuration Parameters** select **Custom comments (MPT objects only)**.
5. Set **Custom comments function** to the name of the MATLAB file or TLC file that you created.
6. Generate code from the model. The comments that your function generates appear near the code that represents each data object.

   ```
   /* Object: myParam - user description:
      DataType -- auto
      Units    --
      CSC      -- Global */
   ```

Limitations

- To generate comments by using the **Custom comments (MPT objects only)** and **Custom comments function** options, you must create data objects from the package `mpt`. The data objects must use a custom storage class.
- Only the custom storage classes from the `mpt` package that create unstructured variables support the custom comments function.

See Also

More About

- “Organize Parameter Data into a Structure by Using Struct Storage Class” on page 33-67
- “Model Configuration Parameters: Comments”
- “Data Objects”
- “MPT Data Object Properties” on page 39-2

Add Global Comments in the Generated Code

In this section...

"Use a Simulink DocBlock to Add a Comment" on page 55-5
"Use a Simulink Annotation to Add a Comment" on page 55-7
"Use Sorted Notes to Add Comments" on page 55-8
"Limitations" on page 55-9

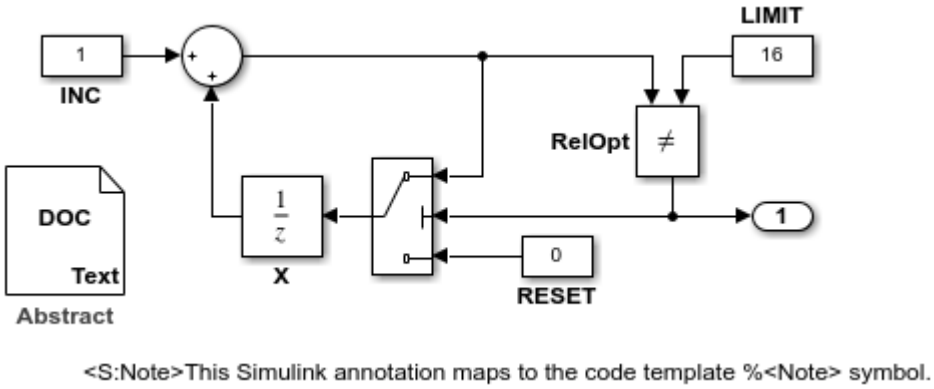
These examples show how to add a global comment to a Simulink model so that the comment text appears in the generated file or files. Specify a template symbol name with a Simulink DocBlock, a Simulink annotation, or a Stateflow note. You can also use a sorted-notes capability that works with Simulink annotations or Stateflow notes, but not DocBlocks. For more information about template symbols, see "Template Symbols and Rules" on page 55-81.

Note Template symbol names `Description` and `Model Information` are fields in the **Property Inspector** dialog box. If you use one of these symbol names for global comment text and its **Property Inspector** fields also has text in it, both names appear in the generated files.

Use a Simulink DocBlock to Add a Comment

1. Open a Simulink model.

```
model='rtwdemo_codetemplate';
open_system(model)
```



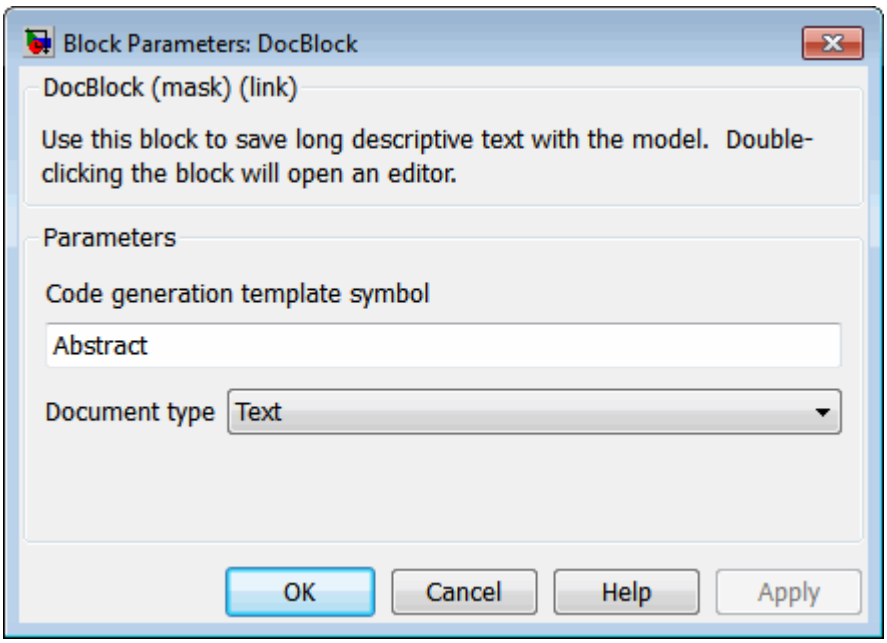
2. Under Configuration Parameters, set the **Code templates** to `rtwdemocodetemplate.cgt`. This code template contains the required customizations to generate global comments.
3. Right-click the DocBlock and select **Mask > Mask Parameters**.
4. In the **Code generation template symbol** box, type one of the following:

   - `Abstract`
   - `Description`
   - `History`

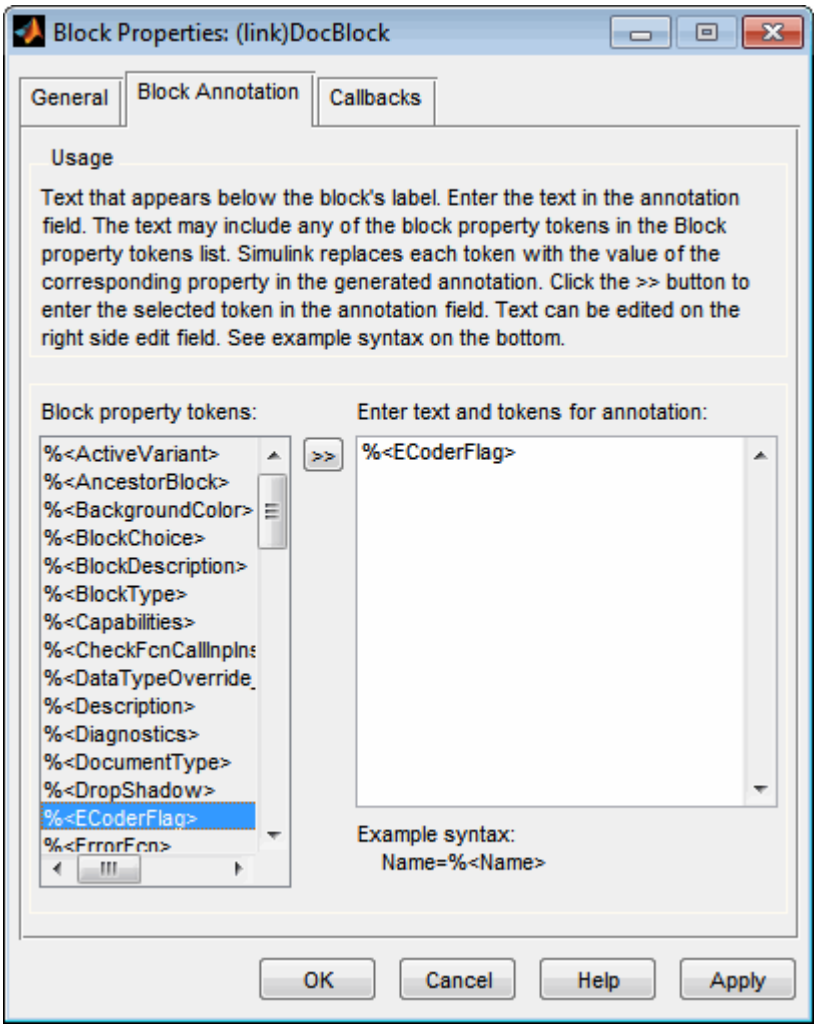
- `Notes`

Click **OK**. Template symbol names are case-sensitive.

If you are using a DocBlock to add comments to your code, set the **Document type** to `Text`. If you set **Document type** to `RTF` or `HTML`, your comments do not appear in the code.



5 In the Block Properties dialog box, on the **Block Annotation** tab, select `%<ECoderFlag>` and click **OK**. The symbol name that you typed in the previous step now appears under the DocBlock in the model.



6 Save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you designated. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

The relevant fragment from the generated file for this model is:

```
*
* ABSTRACT:
* This text maps to the code template %<Abstract> symbol.
```

7 To add more comments to the generated files, repeat steps 2–6.

Use a Simulink Annotation to Add a Comment

1 Open the model `rtwdemo_codetemplate`.

2 Double-click the unoccupied area on the model where you want to place the comment. See “Annotate Models”.

3 Type `<S:Symbol_name>` followed by the comment. `Symbol_name` is one of the following:

- `Abstract`
- `Description`
- `History`
- `Notes`

For example, type `<S:Description>This is the description I want`. Template symbol names are case-sensitive. If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Use Sorted Notes to Add Comments" on page 55-8.

4. Click outside the annotation rectangle and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you assigned. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file. If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Use Sorted Notes to Add Comments" on page 55-8.
5. To add one or more other comments to the generated files, repeat steps 2–4.

Use Sorted Notes to Add Comments

The sorted-notes capability enables you to add sorted comments to the generated files. The code generator places these comments in each generated file at the location that corresponds to where the `Notes` symbol is located in the template file.

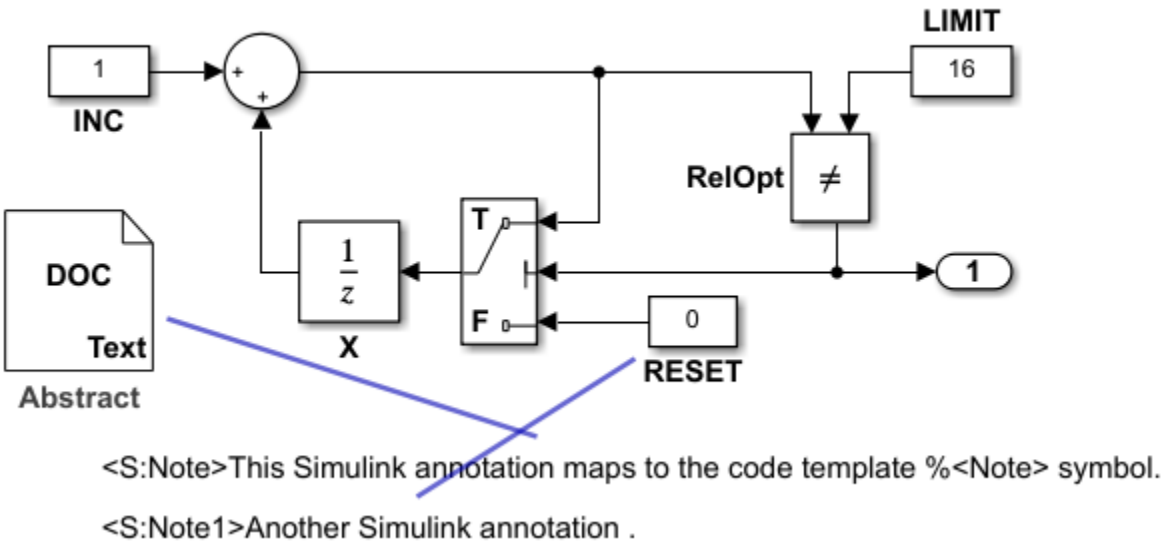
The code generator uses this sorting order:

- Numbers before letters.
- Among numbers, zero is placed first.
- Among letters, uppercase letters are placed before lowercase letters.

You can use sorted notes for a Simulink annotation or a Stateflow note, but not with a DocBlock.

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment. `Y` is a number or a letter.
- Repeat this process for as many additional comments you want. Replace `Y` with a subsequent number or letter.

This figure illustrates sorted notes on a model. It shows where the code generator places each note in a generated file.



The relevant fragment from the generated file for this model is:

```
*
* NOTES:
* Note: This Simulink annotation maps to the code template %<Note> symbol.
* Note1: Another Simulink annotation .
*
```

Limitations

You cannot generate global comments using template symbol names inside a Simulink DocBlock, when the **Language** for code generation is `C++`.

See Also

More About

- “Model Configuration Parameters: Comments”

Customize Generated Identifier Naming Rules

In this section...
“Apply Naming Rules to Identifiers Globally” on page 55-10 “Apply Custom Naming Conventions to Identifiers” on page 55-11

For GRT and RSim targets, the code generator constructs identifiers for variables and functions in the generated code. For ERT targets, you can customize the naming of identifiers in the generated code by specifying parameters on the **Code Generation > Identifiers** pane in the Configuration Parameters dialog box. You can also specify parameters that control identifiers generated from Simulink data objects. For detailed information about these parameters, see “Model Configuration Parameters: Code Generation Identifiers”.

Apply Naming Rules to Identifiers Globally

Goal	Specification
Set the maximum number of characters that the code generator uses for function, `typedef`, and variable names (default 31).	An integer value for the Maximum identifier length parameter. For more information, see “Specify Identifier Length to Avoid Naming Collisions”. If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or if identifiers are mangled more than you expect, increase the value of this parameter.
Define a macro that specifies certain text included within generated identifiers for: • Global variables • Global types • Field names of global types • Subsystem methods • Subsystem method arguments • Local temporary variables • Local block output variables • Constant macros • Shared utilities identifier format	A macro for the **Identifier format control** parameters. For more information, see “Identifier Format Control” on page 55-17. See also “Exceptions to Identifier Formatting Conventions” on page 55-28 and “Identifier Format Control Parameters Limitations” on page 55-29.
Set the minimum number of characters that the code generator uses for the mangling text.	An integer value for the Minimum mangle length parameter. For more information, see “Control Name Mangling in Generated Identifiers” on page 55-25

Goal	Specification
Control whether the software uses shortened names for system-generated identifiers.	`Shortened` for the System-generated identifiers parameter. This setting: • Provides more space for user names. • Provides a more predictable and consistent naming system that uses camel case. • Does not include underscores or plurals. • Provides consistent abbreviations for both a type and a variable.
Control whether the generated code expresses scalar inlined parameter values as literal values or as macros.	The value `Literals` or `Macros` for the Generate scalar inlined parameters as parameter. • `Literals`: If you set **Default parameter behavior** to `Inlined`, parameters are expressed as numeric constants. • `Macros`: Parameters are expressed as variables (with `#define` macros). This setting makes code more readable.

Apply Custom Naming Conventions to Identifiers

This example shows how to apply uniform naming rules for Simulink® data objects, including signals, parameters, and data store memory variables.

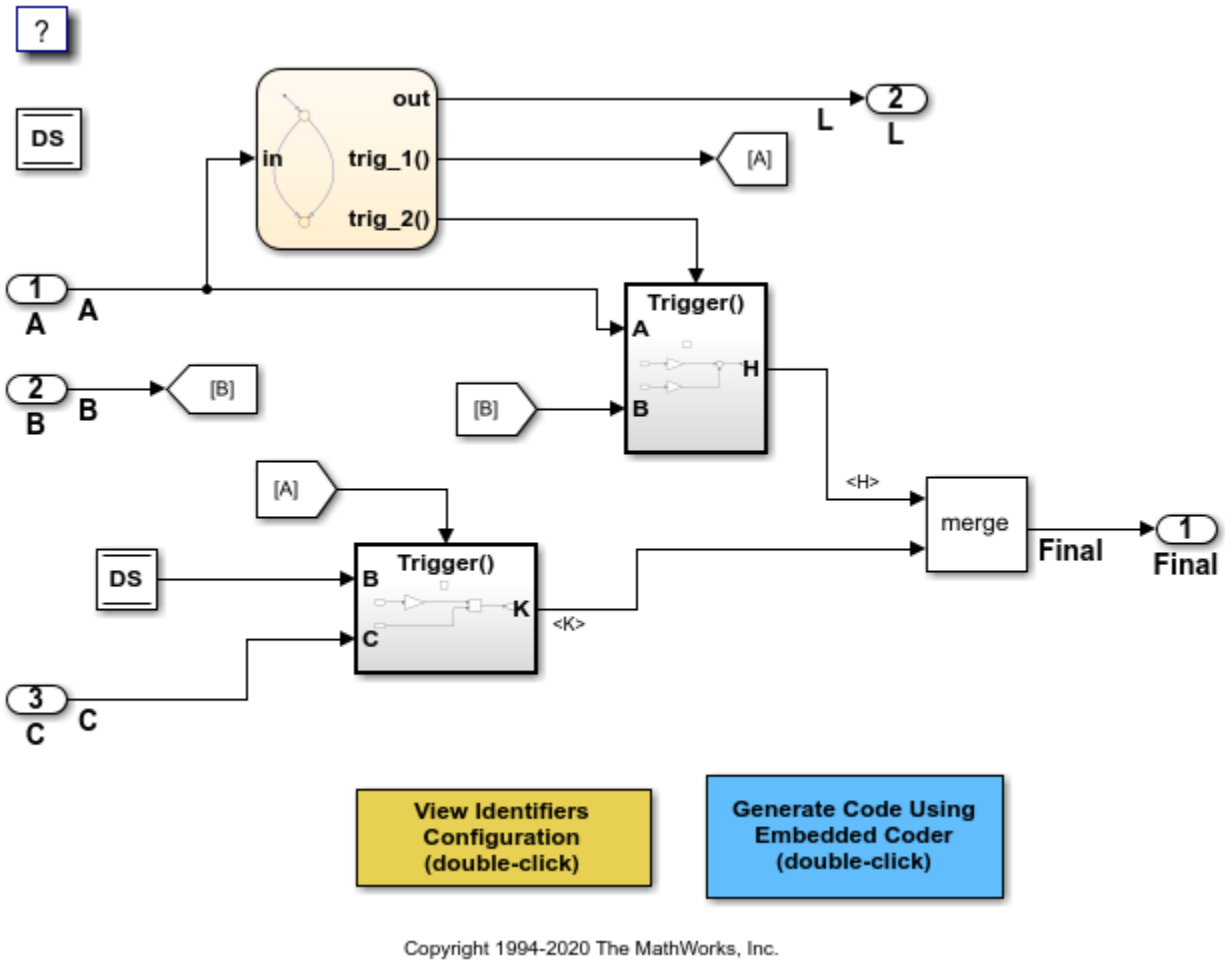
When your model uses Simulink data objects from the `Simulink` package, identifiers in the generated code copy the names of the objects by default. For example, a `Simulink.Signal` object named `Speed` appears as the identifier `Speed` in generated code.

You can control these identifiers by specifying naming rules that are specific to Simulink data objects. When you specify naming rules for generated code, follow ANSI® C/C++ rules for naming identifiers.

Apply Lowercase Naming Rule

1. Open the model.

```
model='rtwdemo_namerules';
open_system(model)
```



2. Open the **C Code** tab and click **View Code**.

3. In the **Model Configuration Parameters**, navigate to **Identifiers > Advanced parameters > Simulink data object naming rules**. In this model, all three rules are set to `Force lower case`.

4. Generate code for the model. The generated file `rtwdemo_namerules.c` uses lowercase identifiers for parameters.

```
/* Exported block parameters */
real_T f1 = 2.0;                      /* Variable: F1
                                       * Referenced by: '<Root>/Chart'
                                       */
real_T g1 = 3.0;                      /* Variable: G1
                                       * Referenced by:
                                       *   '<Root>/Chart'
                                       *   '<S3>/Gain1'
                                       */
real_T g2 = 4.0;                      /* Variable: G2
                                       * Referenced by: '<S3>/Gain2'
                                       */
real_T g3 = 5.0;                      /* Variable: G3
                                       * Referenced by: '<S2>/Gain'
                                       */
```

Apply Naming Rule Using a Function

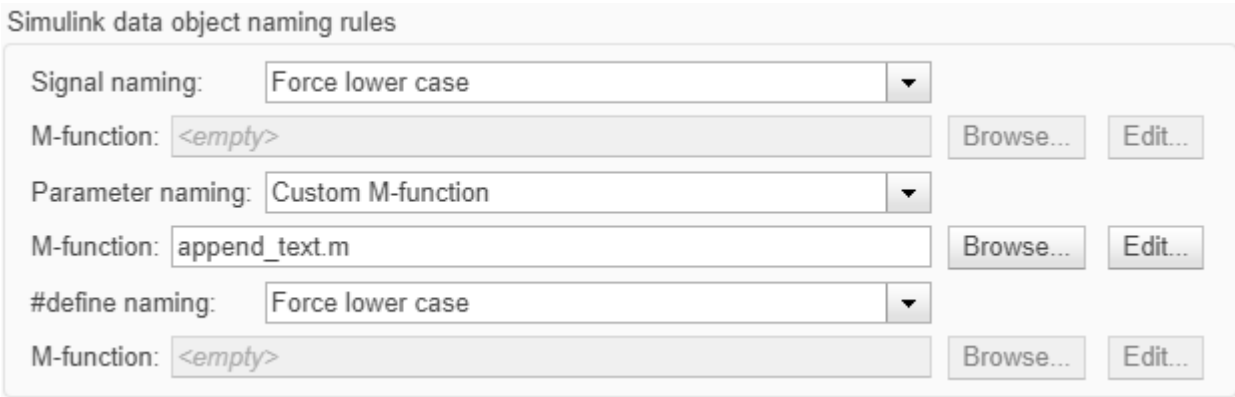
You can customize identifiers in generated code by defining a MATLAB® function.

1. Write a MATLAB function that returns an identifier by modifying a data object name, then save the function in your working folder. For example, the following function returns an identifier name by appending a data object's data type to that data object's name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: information about model
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
DataType = Simulink.data.evalinGlobal(object.modelName,[name,'.DataType']);

revisedName = [name,'_',DataType];
```

2. In the **Model Configuration Parameters**, navigate to **Identifiers > Advanced parameters > Simulink data object naming rules**. From the **Parameter naming** drop-down list, select `Custom M-function`.

3. In the **M-function** field under `Parameter naming`, type the name of the file that defines the MATLAB function, `append_text.m`, then click **Apply**.



4. Generate code for the model. The generated code implements the parameter object naming rule in `rtwdemo_namerules.c`.

```
/* Exported block parameters */
real_T F1_auto = 2.0;                  /* Variable: F1
                                        * Referenced by: '<Root>/Chart'
                                        */
real_T G1_auto = 3.0;                  /* Variable: G1
                                        * Referenced by:
                                        *   '<Root>/Chart'
                                        *   '<S3>/Gain1'
                                        */
real_T G2_auto = 4.0;                  /* Variable: G2
                                        * Referenced by: '<S3>/Gain2'
                                        */
real_T G3_auto = 5.0;                  /* Variable: G3
                                        * Referenced by: '<S2>/Gain'
                                        */
```

Specify Naming Rule for Storage Class `define`

You can specify a naming rule that applies only to Simulink data objects whose storage class you set to `Define`.

1. Open the **Code Mappings Editor**. On the **C Code** tab, select **Code Interface > Default Code Mappings**.

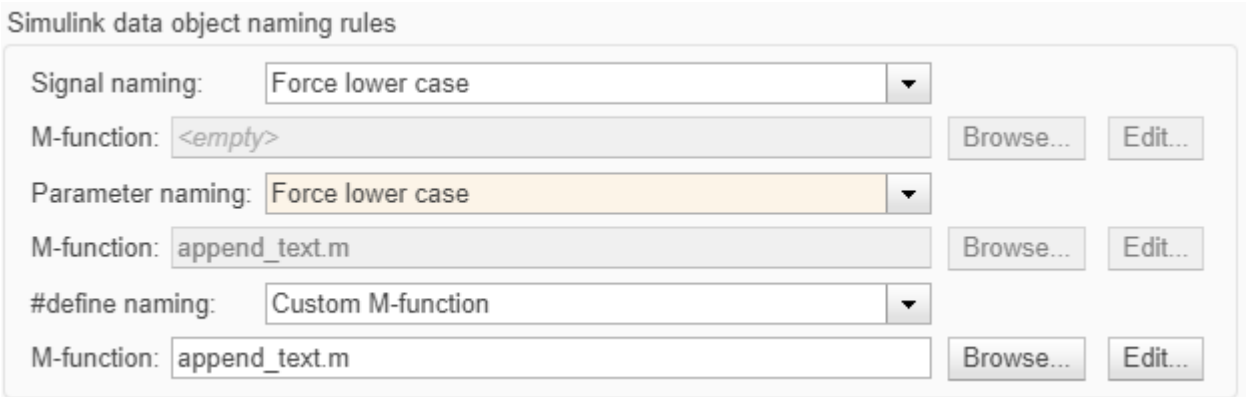
2. In the **Parameters** tab of the **Code Mappings Editor**, find the data table row labeled `External Parameter Objects`. Click the **Refresh** button in that row.

3. Find the row in the data table that corresponds to the `Simulink.Parameter` object `G1`, which resides in the base workspace. From the drop-down menu, set the Storage Class for `G1` to `Define`.

4. In the **Model Configuration Properties**, navigate to **Identifiers > Advanced parameters > Simulink data object naming rules**. From the `#define naming` drop-down list, select **Custom M-function**.

5. In the **M-function** field under `#define naming`, type the name of the file that defines the MATLAB function, `append_text.m`.

6. From the **Parameter naming** drop-down list, select `Force lower case`, then click **Apply**.



7. Generate code for the model. The generated file `rtwdemo_namerules.h` represents `G1` with `G1_auto` and `K1` with `K1_auto`.

```
/* Definition for custom storage class: Define */
#define G1_auto                        3.0                      /* Referenced by:
                                                                 * '<Root>/Chart'
                                                                 * '<S3>/Gain1'
                                                                 */
#define K1_auto                        6.0                      /* Referenced by: '<Root>/Chart
```

Override Data Object Naming Rules

You can override data object naming rules for individual data objects by specifying the `Identifier` property. Generated code uses the text that you specify as the identifier to represent the data object, regardless of naming rules.

1. In the **Parameters** tab of the **Code Mappings Editor**, find the row in the data table that corresponds to the `Simulink.Parameter` object `G2`. Right click the row and select **Open**.

2. In the Property Inspector, go to the **Code Generation** tab. Specify the `Identifier` property as `myIdentifier`. Click **Apply**.

3. Generate code for the model. The generated file `rtwdemo_namerules.c` represents `G2` with the variable `myIdentifier`.

```
/* Exported block parameters */
real_T f1 = 2.0;                      /* Variable: F1
                                       * Referenced by: '<Root>/Chart'
                                       */
real_T myIdentifier = 4.0;            /* Variable: G2
                                       * Referenced by: '<S3>/Gain2'
                                       */
real_T g3 = 5.0;                      /* Variable: G3
```

```
* Referenced by: '<S2>/Gain'
*/
```

See Also

Signal naming

Identifier Format Control

You can customize generated identifiers by specifying the **Identifier format control** parameters on the **Code Generation > Identifiers** pane in the Configuration Parameters dialog box. For each parameter, you can enter a macro that specifies whether, and in what order, certain text is included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier using the `$R` token.

The macro can include:

- Valid tokens, which are listed in Identifier Format Tokens. You can use or omit tokens depending on what you want to include in the identifier name. The **Shared utilities identifier format** parameter requires you to specify the checksum token, `$C`. The other parameters require the mangling token, `$M`. For more information, see “Control Name Mangling in Generated Identifiers” on page 55-25. The mangling token is subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values.
- Token decorators, which are listed in “Control Case by Using Token Decorators” on page 55-22. You can use token decorators to control the case of generated identifiers for each token.
- Valid C or C++ language identifier characters (`a-z, A-Z, _ , 0-9`).

The build process generates each identifier by expanding tokens and inserting the resultant text into the identifier. The tokens are expanded in the order listed in Identifier Format Tokens. Groups of characters are inserted in the positions that you specify around tokens directly into the identifier. Contiguous token expansions are separated by the underscore (`_`) character.

Identifier Format Tokens

Token	Description
`$C`	This token is required for **Shared utilities identifier format**. If the identifier exceeds the **Maximum identifier length**, the code generator inserts an 8-character checksum to avoid naming collisions. For shared utilities that you generate from a MATLAB Function block, the code generator replaces the `$C` token with a checksum regardless of the length of the identifier. The position of the `$C` token in the **Identifier format control** parameter specification determines the position of the checksum in the generated identifier. For example, if you use the specification `$N$C`, the checksum is appended to the end of the identifier. This token is available only for shared utilities.
`$M`	This token is required. If necessary, the code generator inserts name-mangling text to avoid naming collisions. Modify checksum character length by using **Shared checksum length** parameter. The position of the `$M` token in the **Identifier format control** parameter specification determines the position of the name-mangling text in the generated identifier. For example, if you use the specification `$R$N$M`, the name-mangling text is appended (if required) to the end of the identifier. For more information, see "Control Name Mangling in Generated Identifiers" on page 55-25.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.
`$F`	Insert method name (for example, `_Update` for update method). This token is available only for subsystem methods.
`$N`	Insert name of object (block, signal or signal object, state, parameter, shared utility function or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. When you use referenced models, this token is required in addition to `$M` (see "Avoid Identifier Name Collisions with Referenced Models" on page 55-26). **Note:** This token replaces the **Prefix model name to global identifiers** option in previous releases.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text `root_`. For blocks at the subsystem level, the tag is of the form `sN_`. `N` is a unique system number assigned by the Simulink software. This token is available only for subsystem methods and field names of global types. **Note:** This token replaces the **Include System Hierarchy Number in Identifiers** option in previous releases.
`$A`	Insert data type acronym (for example, `i32` for integers) to signal and work vector identifiers. This token is available for local block output variables, local temporary variables, and field names of global types. **Note:** This token replaces the **Include data type acronym in identifier** option in previous releases.

Token	Description
`$I`	• Insert `u` if the argument is an input. • Insert `y` if the argument is an output. • Insert `uy` if the argument is an input and output. For example, `rtu_` for an input argument, `rty_` for an output argument, and `rtuy_` for an input and output argument. This token is available only for subsystem method arguments.
`$G`	Insert the name of a storage class that is associated with the data item. This token is also available in the naming rule that you specify for the **Header File** for a storage class in the Embedded Coder Dictionary.
`$E`	Insert the file type. `$E` represents these instances of file types: • `capi` • `capi_host` • `dt` • `testinterface` • `private` • `types` This token is required for **Header files** and **Source files**.
`$X`	This token applies only to sender and receiver service function naming rules. Insert the name of the entry-point callable function that encloses the sender or receiver service function call.

Identifier Format Control Parameter Values lists the default macro value, the supported tokens, and the applicable restrictions for each **Identifier format control** parameter.

Identifier Format Control Parameter Values

Parameter	Default Value	Supported Tokens	Restrictions
Global variables	`$R$N$M`	`$M`, `$R`,`$N`,`$U`	`$F`, `$H`, `$A`, `$E`, `$I`, and `$G` are not allowed.
Global types	`$N$R$M_T`	`$M`, `$R`,`$N`,`$U`	`$F`, `$H`, `$A`, `$E`, `$I`, and `$G` are not allowed.
Field name of global types	`$N$M`	`$M`, `$N`, `$H`, `$A`, `$U`	`$R`, `$F`, `$G`, `$E`, and `$I` are not allowed.
Subsystem methods	`$R$N$M$F`	`$M`, `$R`, `$N`, `$H`, `$F`, `$U`	`$F` and `$H` are empty for Stateflow functions; `$A`, `$G`, `$E`, and `$I` are not allowed.
Subsystem method arguments	`rt$I$N$M`	`$M`, `$N`, `$I`, `$U`	`$R`, `$F`, `$H`, `$G`, `$E`, and `$A` are not allowed.
Local temporary variables	`$N$M`	`$M`, `$R`, `$N`, `$A`, `$U`	`$F`, `$H`, `$G`, `$E`, and `$I` are not allowed.
Local block output variables	`rtb_$N$M`	`$M`, `$N`, `$A`, `$U`	`$R`, `$F`, `$H`, `$G`, `$E`, and `$I` are not allowed.
Constant macros	`$R$N$M`	`$M`, `$R`, `$N`, `$U`	`$F`, `$H`, `$A`, `$G`, `$E`, and `$I` are not allowed.
Shared utilities identifier format	`$N$C`	`$N`, `$C`, `$R`, `$U`	`$C` is required. `$M`, `$F`, `$H`, `$A`, `$G`, `$E`, and `$I` are not allowed.
EMX array utility functions identifier format	`emx$M$N`	`$M`, `$N`,`$R`	`$C`, `$U`, `$F`, `$H`, `$A`, `$G`, `$E`, and `$I` are not allowed.
EMX array types identifier format	`emxArray_ $M$N`	`$M`, `$N`,`$R`	`$C`, `$U`, `$F`, `$H`, `$A`, `$G`, `$E`, and `$I` are not allowed.
Header files	`$R$E`	`$R`,`$U`,`$E`	`$C`, `$M`, `$N`, `$F`, `$H`, `$A`, `$G`, and `$I` are not allowed.
Source files	`$R$E`	`$R`,`$U`,`$E`	`$C`, `$M`, `$N`, `$F`, `$H`, `$A`, `$G`, and `$I` are not allowed.
Data files	`$R_data`	`$R`,`$U`	`$C`, `$M`, `$N`, `$F`, `$H`, `$A`, `$G`, `$E`, and `$I` are not allowed.

Non-ERT-based targets (such as the GRT target) implicitly use a default `$R$N$M` specification. This default specification consists of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by name-mangling text.

For limitations that apply to **Identifier format control** parameters, see “Exceptions to Identifier Formatting Conventions” on page 55-28 and “Identifier Format Control Parameters Limitations” on page 55-29.

$N Embedded Coder Dictionary Resolution

The `$N` token resolves to the name of the object for which an identifier is being generated. The object type varies by the context of the naming rule. This table lists the objects to which `$N` resolves for items in the Embedded Coder Dictionary.

Category	Service Interface	Customization Entity	Default Naming Rule	Resolution of $N
Execution	**Initialize and Terminate Functions**	**Function Naming Rule**	`$R$N`	`init` for Initialize Function blocks, `term` for Terminate Function blocks
	Periodic and Aperiodic Functions	**Function Naming Rule**	`$R$N`	`reset` for periodic functions, `step` for aperiodic functions
Service Interfaces	**Receiver**	**Function Naming Rule**	`get_$X$N`	Inport block name
	Sender	**Function Naming Rule for Value**	`set_$X$N`	Outport block name
		Function Naming Rule for Reference		
	Data Transfer	**Receiver Function Naming Rule**	`get_$X$N`	Data transfer signal name, such as `DataTransferAt`*`<Block><Outport><PortNumber>`*
		Sender Function Naming Rule	`set_$X$N`	
Internal Functions	**Subcomponent Functions**	**Function Naming Rule**	`$R$N`	Associated function element, such as the name of the port for an exported function
	Shared Utility Functions	**Function Naming Rule**	`$N$C`	Name of the function
Memory	**Storage Class**	**Header File**	`$N.h`	Model or subsystem with which the data is associated
		Definition File	`$N.c`	Model or subsystem with which the data is associated

Category	Service Interface	Customization Entity	Default Naming Rule	Resolution of $N
		Type Name	`$R$$N$G$M`	Model or subsystem with which the data is associated
		Instance Name	`$G$N$M`	Model or subsystem with which the data is associated
		Name of Getter	`get_$N$M`	Associated data element
		Name of Setter	`set_$N$M`	Associated data element

Control Case by Using Token Decorators

On the **Code Generation** > **Identifiers** pane, you can use token decorators to control the case of generated identifiers. Place a decorator immediately after the target token and enclose the decorator in square brackets `[ ]`. For example, you can set **Global variables** to `$R[uL]$N$M`, which capitalizes the first letter of the model name and forces the remaining characters in the model name to lowercase.

The table shows how to manipulate the expansion of the `$R` token for a model whose name is `modelName`.

Desired Expansion	Description	Token and Decorator
`ModelName`	First letter of model name is uppercase. Remaining characters are not modified.	`$R[u]`
`Modelname`	First letter of model name is uppercase. Remaining characters are lowercase.	`$R[uL]`
`MODELNAME`	All characters are uppercase.	`$R[U]`
`modelname`	All characters are lowercase.	`$R[L]`
`mODELNAME`	First letter of model name is lowercase. Remaining characters are uppercase.	`$R[lU]`
`modelName`	First letter of model name is lowercase. Remaining characters are not modified.	`$R[l]`

When you use a decorator, the code generator removes the underscore character (_) that appears between tokens by default. However, you can append each decorator with an underscore: `$R[U_]$N`. For example, if you set the **Global variables** parameter to `$R[u_]$N[uL]$M` for a model named `modelName` and a `DWork` structure represented by `DW`, the result is `ModelName_Dw`.

Modify Identifiers by Using Regular Expression Decorators

To modify identifiers in ways other than by case, you can use regular expressions. To use a regular expression, enclose the decorator in double quotes. A regular expression decorator contains two

regular expressions separated by a forward slash. The code generator searches for substrings of the token that match the first regular expression and replaces those substrings using the second regular expression. For example, the following identifier naming rule takes the root model name `$R` and replaces instances of `a` with `b`: `$R["a/b"]`. The code generator interprets regular expression syntax in a way that is consistent with the `regexprep` function, except regular expression decorators do not support dynamic expressions, which use MATLAB commands.

You can configure a regular expression decorator by appending a list of options to the end of the decorator. Separate the options from the second regular expression by using a forward slash and separate the options from each other by using the pipe character. For example, the following identifier naming rule takes the root model name `$R` and replaces only the first instance of `a` with `b`, ignoring case: `$R["a/b/once|ignorecase"]`.

Category	Option	Description	Example Decorator	Example Input	Example Output
Match/replace occurrence	`all` (default)	Match and replace the expression as many times as possible.	`["a/-/all"]`	`afabcswbc`	`-f-bcswbc`
	`once`	Match and replace the expression only once.	`["a/-/once"]`	`afabcswbc`	`-fabcswbc`
	`N`	Where `N` is an integer, match and replace the `N`th occurrence of a match.	`["a/-/2"]`	`afabcswbc`	`af-bcswbc`
Case matching	`matchcase` (default)	Match letter case.	`["aBcD/wXyZ/matchcase"]`	`abcdABCD`	`abcdABCD`
	`ignorecase`	Ignore letter case.	`["aBcD/wXyZ/ignorecase"]`	`abcdABCD`	`wXyZwXyZ`
	`preservecase`	Ignore letter case while matching but preserve the case of corresponding characters in the original text while replacing.	`["aBcD/wXyZ/preservecase"]`	`abcdABCD`	`wxyzWXYZ`
Empty matching	`noemptymatch` (default)	Ignore zero-length matches.	`["^/prefix/noemptymatch"]`	`abcdABCD`	`abcdABCD`
	`emptymatch`	Include zero-length matches.	`["^/prefix/emptymatch"]`	`abcdABCD`	`prefixabcdABCD`

If a regular expression decorator results in uncompilable code, the code generator produces an error before building the code.

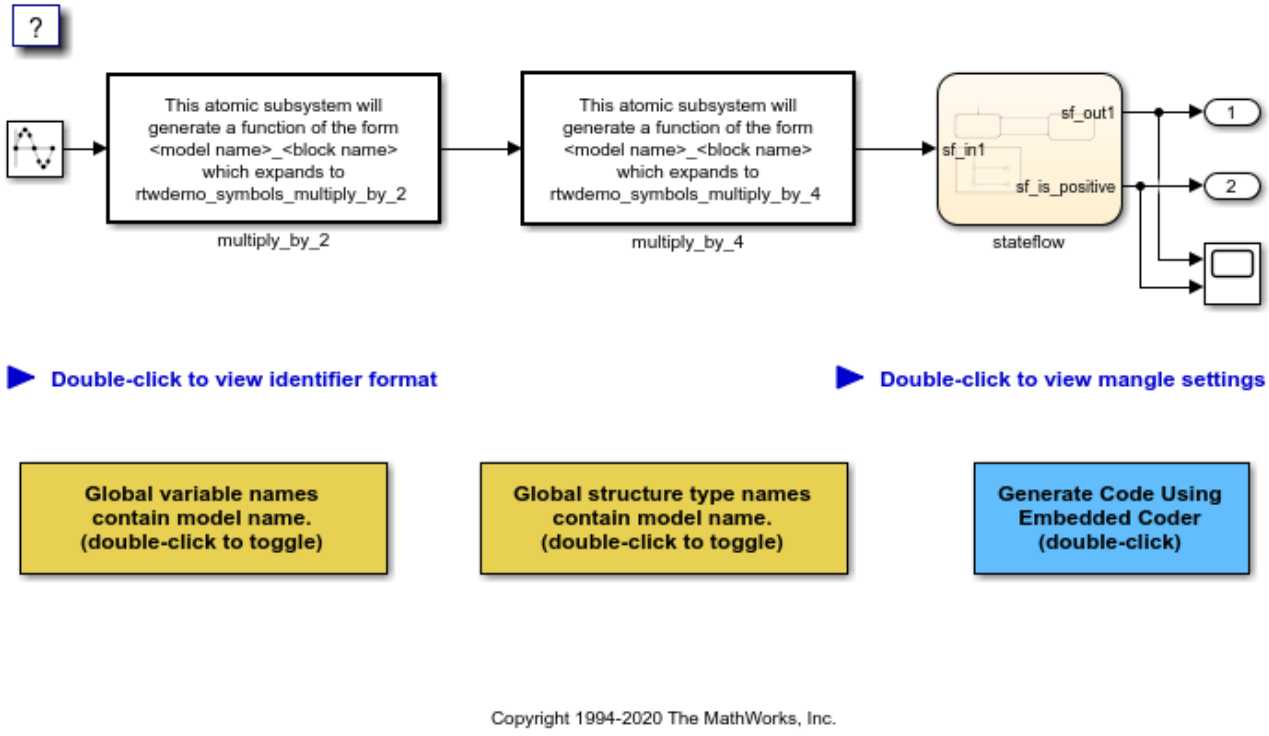
Regular Expression Limitations

- Regular expression decorators are applicable only to `$G`, `$N`, and `$R` tokens.
- Unlike `regexprep`, regular expression decorators do not support dynamic expressions, which use MATLAB commands.
- Regular expression decorators are not supported for function mappings in the **Code Mappings** editor.
- The code generator cannot evaluate whether a regular expression is valid. An invalid regular expression does not result in a change to the affected token and does not produce a warning.

Control Formatting of Identifiers

This example shows how to customize generated identifiers by specifying the **Identifier format control** parameters on the **Code Generation > Identifiers** pane in the Configuration Parameters dialog box. To maintain model traceability, it is important that incremental revisions to a model have minimal impact on the identifier names that appear in the generated code. This example shows how to use name mangling by specifying the **Identifier format control** parameters to minimally impact the identifier names.

```
model='rtwdemo_symbols';
open_system(model)
```




```
% Cleanup
rtwdemoclean;
close_system(model,0)
```

Control Name Mangling in Generated Identifiers

The position of the `$M` token in the **Identifier format control** parameter specification determines the position of the name-mangling text in the generated identifiers. For example, if you use the specification `$R$N$M`, the name-mangling text is appended (if required) to the end of the identifier. For more information, see “Identifier Format Control” on page 55-17.

Name-Mangling Text Per Object

Object Type	Source of Mangling Text
Block diagram	Name of block diagram
Simulink block	Automatically assigned Simulink identifier
Simulink parameter	Full name of parameter owner (model or block) and parameter name
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name-mangling text is specified by the Minimum mangle length parameter. The default value is `1`, but this automatically increases during code generation as a function of the number of collisions. To minimize disturbance to the generated code during development, specify a larger **Minimum mangle length**. A **Minimum mangle length** of `4` is a conservative value. A value of `4` allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimize Name Mangling

The length of generated identifiers is limited by the **Maximum identifier length** parameter. When a name collision exists, the `$M` token is expanded to the minimum number of characters required to avoid the collision. Other tokens are expanded in the order listed in Identifier Format Tokens. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. To avoid partial expansions, it is good practice to:

- Avoid name collisions. One way to avoid name collisions is to not use default block names (for example, `Gain1`, `Gain2...`) when there are many blocks of the same type in the model.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.
- Set the **Maximum identifier length** parameter to reserve at least three characters for the name-mangling text. The length of the name-mangling text increases as the number of name collisions increases.

  If changes to the model create more or fewer collisions, existing name-mangling text increases or decreases in length. If the length of the name-mangling text increases, additional characters are appended to the existing text. For example, the mangling text `'xyz'` can change to `'xyzQ'`. For fewer collisions, the name-mangling text `'xyz'` changes to `'xy'`.

Avoid Identifier Name Collisions with Referenced Models

Within a model that uses referenced models, collisions between the names of the models are not allowed. When generating code from a model that uses model referencing:

- You must include the `$R` token in the **Identifier format control** parameter specifications (in addition to the `$M` token).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens. If **Maximum identifier length** is too small, a code generation error occurs.

When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the identifier from the referenced model is preserved. Name mangling is performed on the identifier from the higher-level model.

If your model contains two referenced models with the same input or output port names, and one of the referenced models contains an atomic subsystem with "Function packaging" set to `Nonreuseable function`, a name conflict can occur and the build process produces an error.

Use Model Advisor to Detect Identifier Names Changed During Code Generation

For a referenced model, if the following parameters on have specifications that do not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

- Global variables
- Global types
- Subsystem methods
- Constant macros

You can use the Model Advisor to identify referenced models in a model referencing hierarchy for which code generation changes these configuration parameter settings.

1. In the **Modeling** tab, click **Model Advisor**.
2. Select **By Task**.
3. Run the **Check code generation identifier formats used for model reference** check.

See Also

Related Examples

- "Identifier Format Control" on page 55-17
- "Specify Identifier Length to Avoid Naming Collisions" on page 48-29

Maintain Traceability for Generated Identifiers

To verify your model, you can trace back and forth between generated identifiers and corresponding entities within the model. To maintain traceability, it is important that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways to minimally impact the identifier names:

- Choose unique names for Simulink objects (blocks, signals, states, and so on) as much as possible.
- Use name mangling when conflicts cannot be avoided.

The position of the name-mangling text is specified by the placement of the `$M` token in the **Identifier format control** parameters. Mangle characters consist of alphanumeric characters that are unique to each object. For more information, see “Control Name Mangling in Generated Identifiers” on page 55-25.

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described in “Identifier Format Control” on page 55-17.

- Type name generation: name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. If the `$R` token is included in the **Identifier format control** parameter specification, the model name is included in the `typedef`. When generating type definitions, the **Maximum identifier length** parameter is not respected.
- Non-`Auto` storage classes: the **Identifier format control** parameters specification does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For shared utilities, code generation inserts the checksum specified by `$C` to prevent name collisions in the following situations:
  - `$C` is specified without `$N`.
  - The length of `$N` plus the length of the text that you specify exceeds the **Maximum identifier length**. Code generation truncates `$N` and inserts an 8-character checksum where you specified `$C` in the formatting scheme.

.

Identifier Format Control Parameters Limitations

The following limitations apply to the **Identifier format control** parameters:

- The following autogenerated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Code Generation > Identifiers** pane of the Configuration Parameters dialog box.
  - Model methods
    - The applicable format scheme is `$R$F`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can be up to 19 characters without exceeding the default **Maximum identifier length** of 31.
  - Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - `DW` identifiers generated by S-functions in referenced models
  - Fixed-point shared utility macros or shared utility functions
  - Simulink `rtm` macros
    - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
  - Define protection guard macros
    - Header file guards, such as `_RTW_HEADER_$(filename)_h_`, which can exceed the default **Maximum identifier length** of 31 given a filename such as `$R_private.h`.
    - Include file guards, such as `_$R_COMMON_INCLUDES_`.
    - `typedef` guards, such as `_CSCI_$R_CHARTSTRUCT_`.
- In some situations, the following identifiers potentially can conflict with others.
  - Model methods
  - Reentrant model function arguments
  - Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Fixed-point shared utility macros or shared utility functions
  - Include header guard macros
- The following external identifiers that are unknown to the Simulink software might conflict with autogenerated identifiers.
  - Identifiers defined in custom code
  - Identifiers defined in custom header files

  - Identifiers introduced through a non-ANSI C standard library
  - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets might exceed the **Maximum identifier length**. Simulation targets include the model reference simulation target, the accelerated simulation target, the RSim target, and the S-function target.
- Identifiers generated using a model name and bus object data type name, which are both long names, might exceed the **Maximum identifier length**. For example, a ground value variable name is generated as `<model_name>_rtZ<bus_name>`. If the *model_name* and *bus_name* are close to the maximum identifier length, the name exceeds the maximum identifier length.

Control Parentheses in Generated Code

C code contains some syntactically required parentheses, and can contain additional parentheses that change semantics by overriding default operator precedence. C code can also contain optional parentheses that have no functional significance, but only increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and precedence overrides so that C precedence rules specify all semantics unless overridden by parentheses.
- Include the maximum parentheses that can exist without duplication so that C precedence rules become irrelevant. Parentheses alone completely specify all semantics.

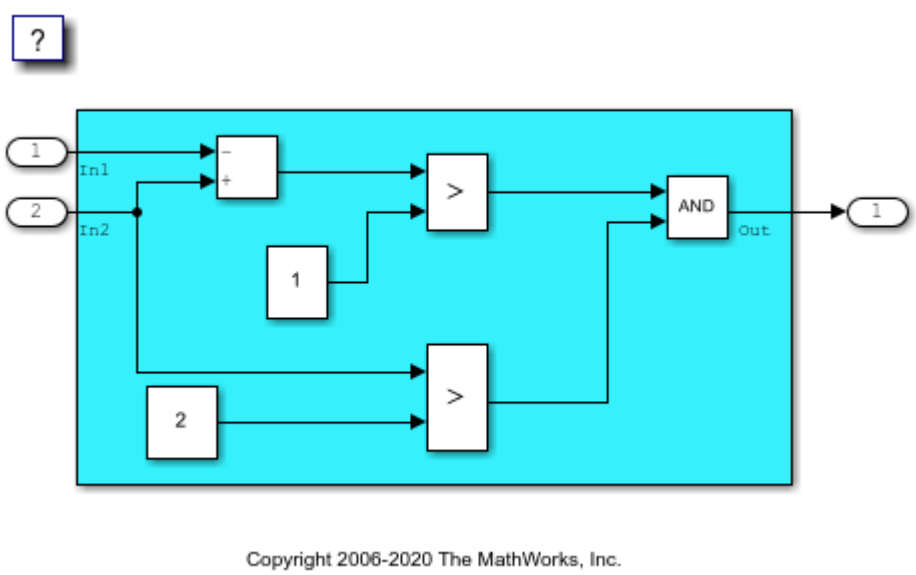
Understanding code with minimum parentheses can require applying nonobvious precedence rules. Maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style useful to people who read code.

Control Use of Parentheses

This example shows how you can control parentheses level in the generated code by configuring the **Parentheses level** parameter.

Open the example model `ParenthesizationStyle`.

```
model='ParenthesizationStyle';
open_system(model)
```



Open the Configuration Parameter dialog box. On the **Code Style** pane, configure the **Parentheses level** parameter based on your requirement.

Build the generated code by pressing **Ctrl+B**.

```
rtwdemoclean;
close_system(model,0)
```

See Also

Related Examples

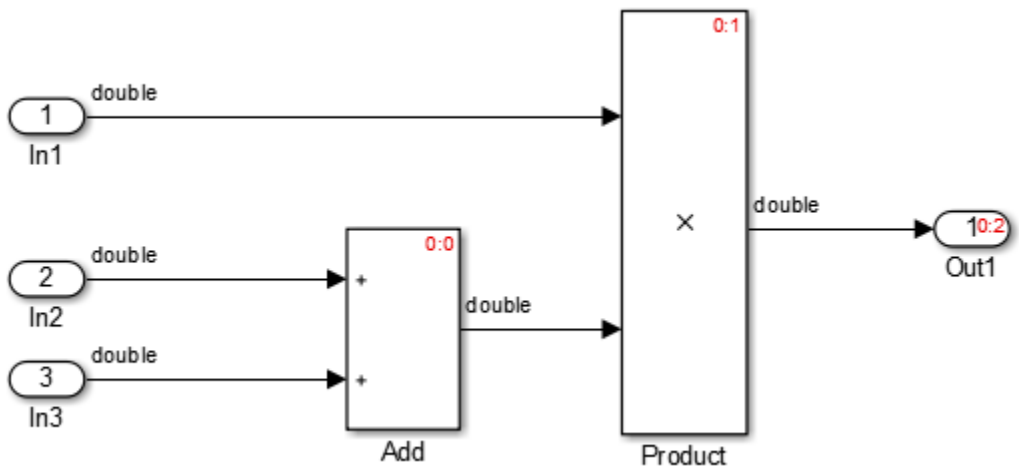
- Parentheses level
- “Model Configuration Parameters: Code Style”

Optimize Code by Reordering Commutable Operands

This example shows how to reorder commutable operands to make expressions left-recursive. Left - recursive reordering saves stack space in memory. This optimization improves code efficiency.

Example Model

To reorder commutable operands, create this model and name it `operand_order`. The output signal is the result of multiplying the signal from Inport block `In1` by the sum of the signals from Inport blocks `In2` and `In3`.



Generate Code

1. Open the Model Configuration Parameters dialog box. On the **Code Style** tab, select the **Preserve operand order in expression** parameter.
2. Generate code for the model.

In the `operand_order.c` file, the `operand_order_step` function contains this code:

```
operand_order_Y.Out1 = operand_order_U.In1 * (operand_order_U.In2 +
    operand_order_U.In3);
```

The code generator preserves the specified expression order in the model, which increases the readability of the code for code traceability.

Generate Code with Optimization

1. Open the Model Configuration Parameters dialog box. On the **Code Style** tab, clear the **Preserve operand order in expression** parameter.
2. Generate code for the model.

In the `operand_order.c` file, the `operand_order_step` function contains this code:

```
operand_order_Y.Out1 = (operand_order_U.In2 + operand_order_U.In3) *
    operand_order_U.In1;
```

The code generator optimizes the code by reordering the commutable operands to make the expression left-recursive, which improve code efficiency.

See Also

Related Examples

- Preserve operand order in expression
- “Model Configuration Parameters: Code Style”

Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements

This example shows how to specify whether to generate default cases for switch-case statements in the code for Stateflow charts. Generated code that does not contain default cases conserves ROM consumption and enables better code coverage because every branch in the generated code is falsifiable.

Some coding standards, such as MISRA, require the default case for switch-case statements. If you want to increase your chances of producing MISRA C compliant code, generate default cases for unreachable Stateflow switch statements.

Example

Figures 1, 2, and 3 show relevant portions of the `sldemo_fuelsys` model, a closed-loop system containing a plant and controller. The Air-fuel rate controller logic is a Stateflow chart that specifies the different operation modes.

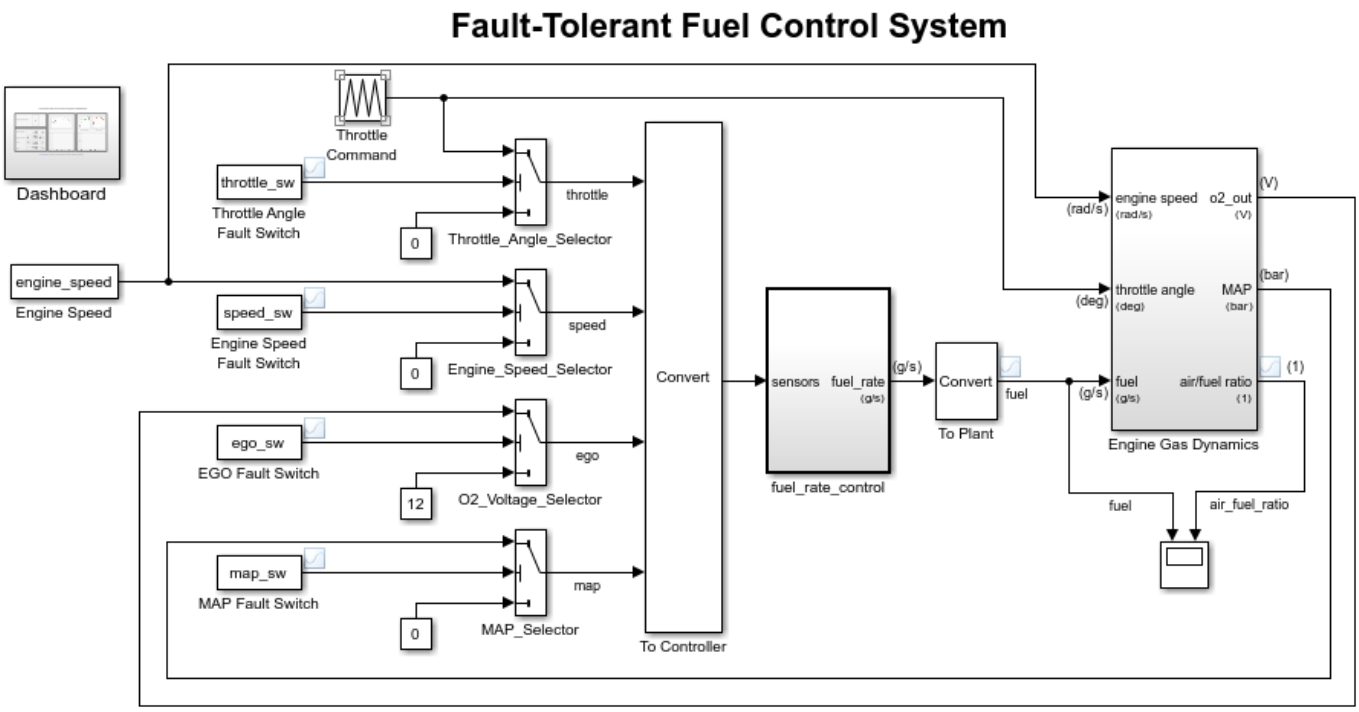


Figure 1: Top-level model of the plant and controller

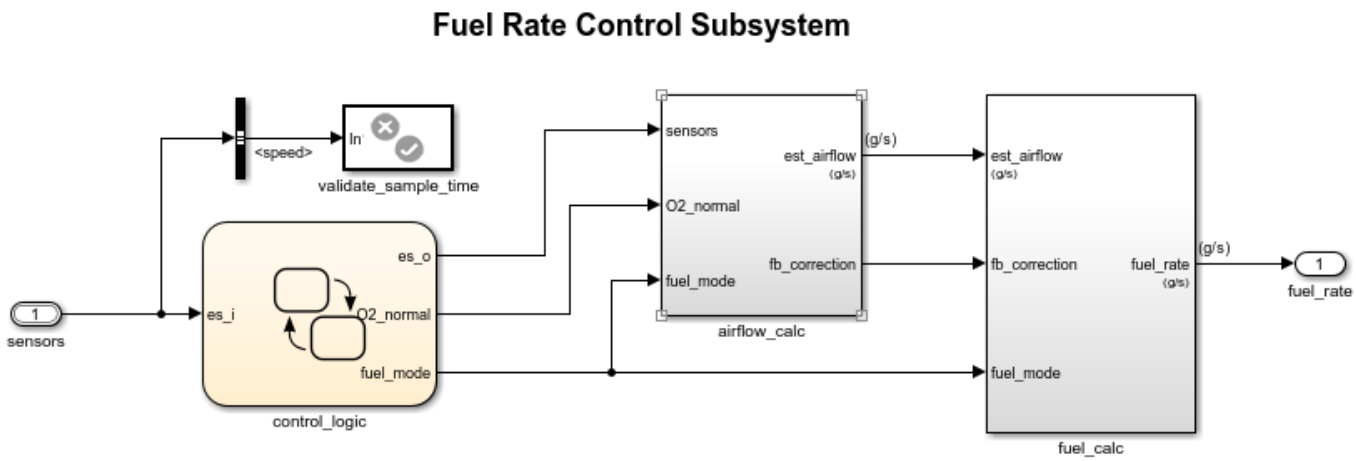


Figure 2: Fuel rate controller subsystem

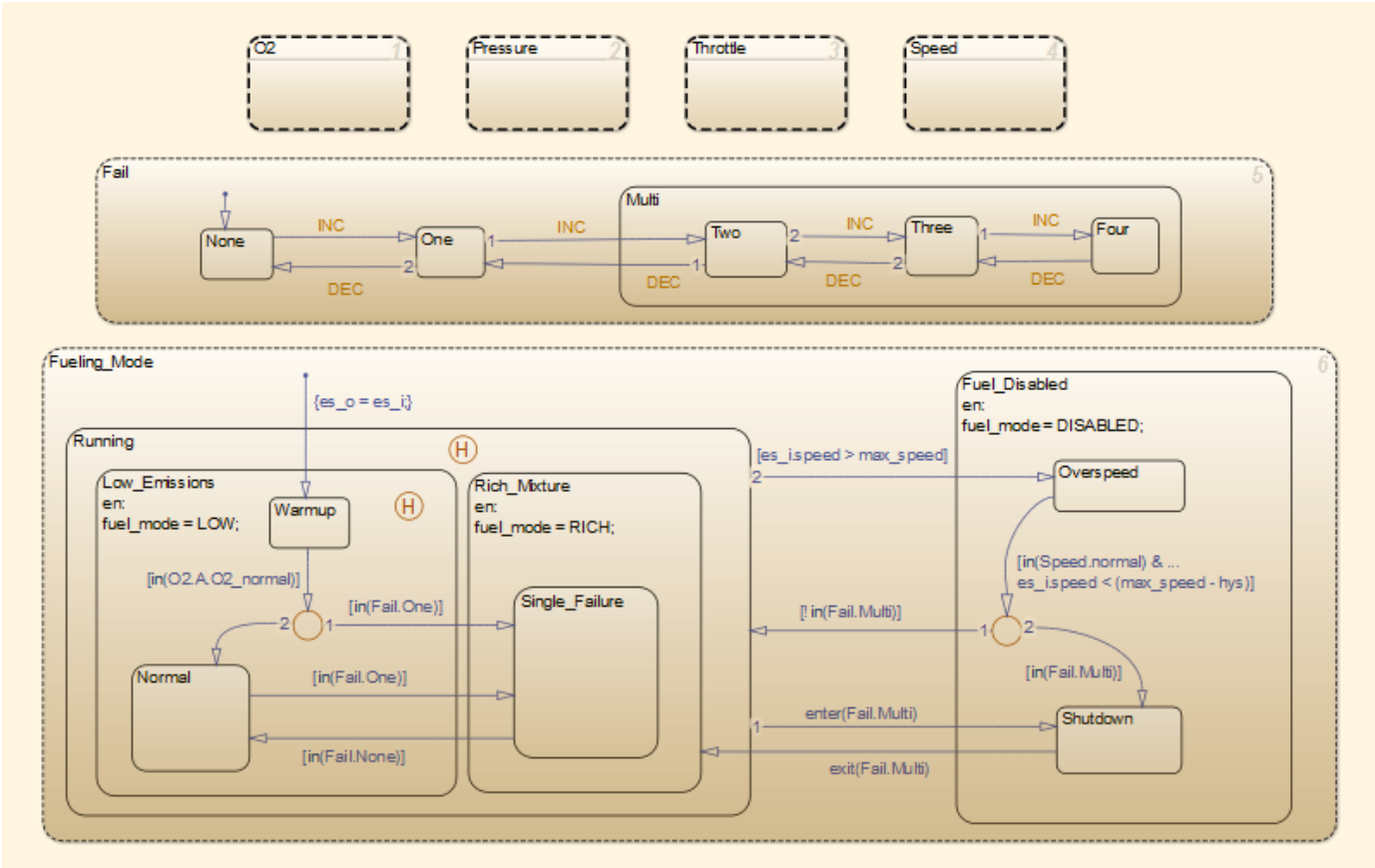


Figure 3: Fuel rate controller logic

Generate Code with Default Cases for Unreachable Stateflow Switch Statements

1. In the MATLAB Command Window, to open `sldemo_fuelsys` enter:

   ```
   openExample('sldemo_fuelsys');
   ```

2. Open the Model Configuration parameters dialog box. On the **Code Generation** > **Code Style** tab, clear the **Suppress generation of default cases for Stateflow switch statements if unreachable** parameter.
3. In the MATLAB Command Window, to build the model, enter:

   ```
   slbuild('sldemo_fuelsys/fuel_rate_control');
   ```

For the different operation modes, the `fuel_rate_control.c` file contains default cases for unreachable switch statements. For example, for the Shutdown operation mode, the generated code contains this default statement:

```
default:
 /* Unreachable state, for coverage only */
 rtDWork.bitsForTID0.is_Fuel_Disabled = IN_NO_ACTIVE_CHILD;
 break;
```

For the Warmup operation mode, the generated code contains this default statement:

```
default:
 /* Unreachable state, for coverage only */
 rtDWork.bitsForTID0.is_Low_Emissions = IN_NO_ACTIVE_CHILD;
 break;
```

Suppress Default Cases for Unreachable Stateflow Switch Statements

1. Open the Configuration Parameters dialog box. On the **Code Generation** > **Code Style** tab, select the **Suppress generation of default cases for Stateflow switch statements if unreachable** parameter.
2. Build the model.

Read through the `fuel_rate_control.c` file. The default cases for unreachable switch statements are not in the generated code.

See Also

Related Examples

- Suppress generation of default cases for Stateflow switch statements if unreachable
- “Model Configuration Parameters: Code Style”

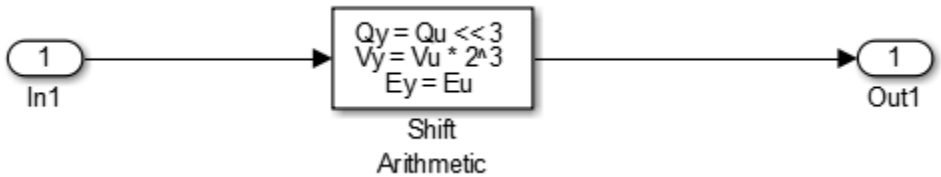
Replace Multiplication by Powers of Two with Signed Bitwise Shifts

This example shows how to generate code that replaces multiplication by powers of two with signed bitwise shifts. Code that contains bitwise shifts executes faster than the code containing multiplication by powers of two.

Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. If you want to increase your chances of producing MISRA C compliant code, do not replace multiplication by powers of two with bitwise shifts.

Example

To replace multiplication by powers of two with bitwise shifts, create the following model. In this model, a signal of **Data type** `int16` feeds into a Shift Arithmetic block. In the Shift Arithmetic Block Parameters dialog box, the **Bits to shift > Direction** parameter is set to `Left`. The **Bits to shift > Number** parameter is set to 3. This parameter corresponds to a value of 8, or raising 2 to the power of 3.



Generate Code with Signed Bitwise Shifts

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. The **Replace multiplications by powers of two with signed bitwise shifts** parameter is on by default.
2. Generate code for the model.

In the `bitwise_multiplication.c` file, the `bitwise_multiplication` step function contains this code:

```
bitwise_multiplication_Y.Out1 = (int16_T)(bitwise_multiplication_U.In1 << 3);
```

The signed integer, `bitwise_multiplication_U.In1`, is shifted three bits to the left.

Generate Code with Multiplication by Powers of Two

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab.
2. Clear the **Replace multiplications by powers of two with signed bitwise shifts** parameter.
3. Generate code for the model.

In the `bitwise_multiplication.c` file, the `bitwise_multiplication` step function contains this code:

```
bitwise_multiplication_Y.Out1 = (int16_T)(bitwise_multiplication_U.In1 * 8);
```

The signed integer `bitwise_multiplication_U.In1` is multiplied by `8`.

See Also

Related Examples

- Replace multiplications by powers of two with signed bitwise shifts
- “Model Configuration Parameters: Code Style”

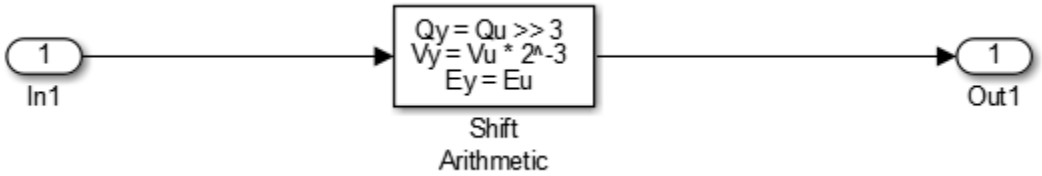
Generate Code Containing Right Shifts on Signed Integers

This example shows how to control whether the generated code contains right shifts on signed integers. Generated code that does not contain right shifts on signed integers first casts the signed integers to unsigned integers, and then right shifts the unsigned integers.

Some coding standards, such as MISRA, do not allow right shifts on signed integers because different hardware can store negative integers differently. For negative integers, you can get different answers depending on the hardware. If you want to increase your chances of producing MISRA C compliant code, do not allow right shifts on signed integers.

Example Model

To generate code with right shifts on signed integers, create this model. A signal of **Data type** `int16` feeds into a Shift Arithmetic block. In the Shift Arithmetic Block Parameters dialog box, the **Bits to shift** > **Direction** parameter is set to `Right`. The **Bits to shift** > **Number** parameter is set to 3.



Generate Code Containing Right Shifts on Signed Integers

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. The **Allow right shifts on signed integers** parameter is on by default.
2. Generate code for the model.

In the `rightshift.c` file, the `rightshift_step` function contains this code:

```
rightshift_Y.Out1 = (int16_T)(rightshift_U.In1 >> 3);
```

The signed integer `rightshift_U.In1` is shifted three bits to the right.

Generate Code That Does Not Allow Right Shifts on Signed Integers

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. Clear the **Allow right shifts on signed integers** parameter.
2. Generate code for the model.

In the `rightshift.c` file, the `rightshift_step` function contains this code:

```
rightshift_Y.Out1 = (int16_T)asr_s32(rightshift_U.In1, 3U);
```

When you clear the **Allow right shifts on signed integers** parameter, the generated code contains a function call instead of a right shift on a signed integer. The function `asr_s32` contains this code:

```
int32_T asr_s32(int32_T u, uint32_T n)
{
```

```
  int32_T y;
  if (u >= 0) {
    y = (int32_T)((uint32_T)u >> n);
  } else {
    y = -(int32_T)((uint32_T)-(u + 1) >> n) - 1;
  }

  return y;
}
```

The `asr_s32` function casts a signed integer to an unsigned integer, and then right shifts the unsigned integer.

See Also

Related Examples

- Allow right shifts on signed integers
- “Model Configuration Parameters: Code Style”

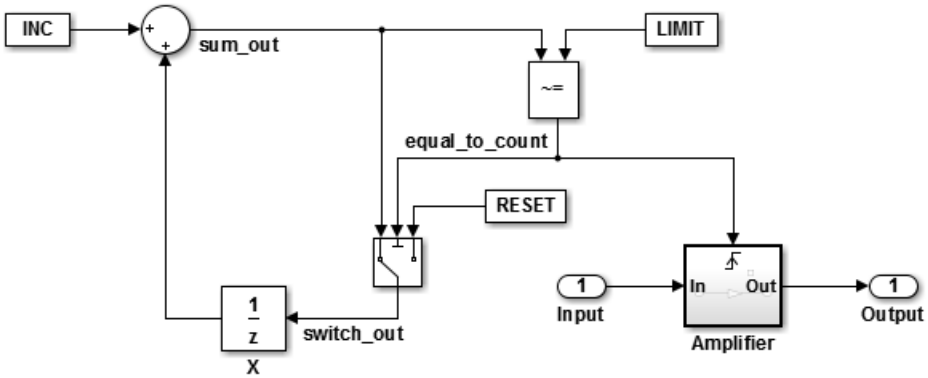
Control Cast Expressions in Generated Code

You can choose how the code generator specifies data type casts in the generated code. In the Configuration Parameters dialog box, select **Code Generation** > **Code Style**. From the **Casting modes** drop-down list, three parameter options control how the code generator casts data types.

- `Nominal` instructs the code generator to generate code that has minimal data type casting. When you do not have special data type information requirements, choose `Nominal`.
- `Standards Compliant` instructs the code generator to cast data types to conform to MISRA standards when it generates code. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations.

  For more information, see “MISRA C” on page 23-14.
- `Explicit` instructs the code generator to cast data type values explicitly when it generates code. You can see how a value is stored, which tells you how much memory space the code uses for the variable. The data type informs you how much precision is possible in calculations involving the variable.

Open the example model `rtwdemo_rtwecintro`.



Enable Nominal Casting Mode and Generate Code

When you choose `Nominal` casting mode, the code generator does not create data type casts for variables in the generated code.

1. On the **Code Generation** > **Code Style** pane, from the **Casting modes** drop-down list, select `Nominal`.
2. On the **Code Generation** > **Report** pane, select **Create code generation report**.
3. On the **Code Generation** pane, select **Generate code only**.
4. Click **Apply**.
5. In the model window, press **Ctrl+B** to generate code.
6. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
```

```
  /* Sum: 'XRootX/Sum' incorporates:
   *  Constant: 'XRootX/INC'
   *  UnitDelay: 'XRootX/X'
   */
  rtDWork.X++;

  /* RelationalOperator: 'XRootX/RelOpt' incorporates:
   *  Constant: 'XRootX/LIMIT'
   */
  rtb_equal_to_count = (rtDWork.X != 16);

  /* Outputs for Triggered SubSystem: 'XRootX/Amplifier' incorporates:
   *  TriggerPort: 'XS1X/Trigger'
   */
  if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
  {
    /* Outport: 'XRootX/Output' incorporates:
     *  Gain: 'XS1X/Gain'
     *  Inport: 'XRootX/Input'
     */
    rtY.Output = rtU.Input << 1;
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
    POS_ZCSIG : (int32_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: 'XRootX/Amplifier' */

  /* Switch: 'XRootX/Switch' */
  if (!rtb_equal_to_count) {
    /* Update for UnitDelay: 'XRootX/X' incorporates:
     *  Constant: 'XRootX/RESET'
     */
    rtDWork.X = 0U;
  }

  /* End of Switch: 'XRootX/Switch' */
}
```

Enable Standards Compliant Casting Mode and Generate Code

When you choose `Standards Compliant` casting mode, the code generator creates MISRA standards compliant data type casts for variables in the generated code.

1. On the **Code Style** pane, from the **Casting modes** drop-down list, select `Standards Compliant`.
2. On the **Code Generation** pane, click **Apply**.
3. In the model window, press **Ctrl+B** to generate code.
4. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/INC'
   *  UnitDelay: '<Root>/X'
   */
  rtDWork.X++;

  /* RelationalOperator: '<Root>/RelOpt' incorporates:
   *  Constant: '<Root>/LIMIT'
   */
  rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);

  /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
   *  TriggerPort: '<S1>/Trigger'
   */
  if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
       POS_ZCSIG)) {
    /* Outport: '<Root>/Output' incorporates:
     *  Gain: '<S1>/Gain'
```

```
     *  Inport: '<Root>/Input'
     */
    rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(int32_T)(rtb_equal_to_count ?
    (int32_T)(uint8_T)POS_ZCSIG : (int32_T)(uint8_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: '<Root>/Amplifier' */

  /* Switch: '<Root>/Switch' */
  if (!rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     */
    rtDWork.X = 0U;
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Enable Explicit Casting Mode and Generate Code

When you choose `Explicit` casting mode, the code generator creates explicit data type casts for variables in the generated code.

1. On the **Code Style** pane, from the **Casting modes** drop-down list, select `Explicit`.
2. On the **Code Generation** pane, click **Apply**.
3. In the model window, press **Ctrl+B** to generate code.
4. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/INC'
   *  UnitDelay: '<Root>/X'
   */
  rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);

  /* RelationalOperator: '<Root>/RelOpt' incorporates:
   *  Constant: '<Root>/LIMIT'
   */
  rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);

  /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
   *  TriggerPort: '<S1>/Trigger'
   */
  if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
    /* Outport: '<Root>/Output' incorporates:
     *  Gain: '<S1>/Gain'
     *  Inport: '<Root>/Input'
     */
    rtY.Output = rtU.Input << 1;
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
        POS_ZCSIG : (int32_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: '<Root>/Amplifier' */

  /* Switch: '<Root>/Switch' */
  if (!(int32_T)rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     */
    rtDWork.X = 0U;
  }

  /* End of Switch: '<Root>/Switch' */
}
```

See Also

Related Examples

- Allow right shifts on signed integers
- “Model Configuration Parameters: Code Style”

Customize Code Organization and Format

In this section...
“Custom File Processing Components” on page 55-46 “Custom File Processing Configuration” on page 55-47

Custom file processing (CFP) tools allow you to customize the organization and formatting of generated code. With these tools, you can:

- Generate a source (`.c` or `.cpp`) or header (`.h`) file. If you configure the **Code interface packaging** for a model to be `Reusable function` or `Nonreuseable function` (not `C++ class`), you can use a *custom file processing template* to control how the code generator:
  - Emits code to standard generated files---*`model`*`.c` or `.cpp` and *`model`*`.h`
  - Produces files that are independent of model code
- Organize generated code into sections (such as includes, `typedef`s, functions, and more). Your CFP template can emit code (for example, functions), directives (such as `#define` or `#include` statements), or comments into each section.
- Generate custom *file banners* (comment sections) at the start and end of generated code files and custom *function banners* that precede functions in the generated code.
- Generate code to call model functions, such as *`model`*`_initialize`, *`model`*`_step`, and so on.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the generated files from the model.

Custom File Processing Components

The custom file processing features are based on the following interrelated components:

- *Code generation template* (CGT) files: a CGT file defines the top-level organization and formatting of generated code. See “Code Generation Template (CGT) Files” on page 55-49.
- The *code template API*: a high-level Target Language Compiler (TLC) API that provides functions with which you can organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls, and perform other functions. See “Code Template API Summary” on page 55-71.
- *Custom file processing (CFP) templates*: a CFP template is a TLC file that manages the process of custom code generation. A CFP template assembles code to be generated into buffers. A CFP template also calls the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections of the generated code. See “Custom File Processing (CFP) Templates” on page 55-55.

To use CFP templates, you must understand TLC programming, for more information, see “Target Language Compiler”.

Custom File Processing Configuration

Customize generated code by specifying code and data templates on the **Code Generation > Templates** pane:

Goal	Action
Specify a template that defines the top-level organization and formatting of generated source code (`.c` or `.cpp`) files	Enter a code generation template (CGT) file for the **Source file (*.c) template** parameter.
Specify a template that defines the top-level organization and formatting of generated header (`.h`) files	Enter a CGT file for the **Header file (*.h) template** parameter. This template file can be the same template file that you specify for **Source file (.c) template**. If you use the same template file, source and header files contain identical banners. The default template is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
Specify a template that organizes generated code into sections (such as includes, `typedefs`, functions, and more)	Enter a custom file processing (CFP) template file for the File customization template parameter. A CFP template can emit code, directives, or comments into each section. For more information, see “Custom File Processing (CFP) Templates” on page 55-55.
Generate a model-specific example main program module	Select **Generate an example main program**. For more information, see “Generate Code for an Application” on page 71-2.

Note Place the template files that you specify on the MATLAB path.

Specify Templates For Code Generation

To use custom file processing features, create CGT files and CFP templates. These files are based on default templates provided by the code generation software. Once you have created your templates, you must integrate them into the code generation process.

Select and edit CGT files and CFP templates, and specify their use in the code generation process in the **Code Generation** > **Templates** pane of a model configuration set. The following figure shows options configured for their defaults.

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (`.c` or `.cpp`) files. You must place this file on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path.

  By default, the template for both source and header files is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP template file to use when generating code files. You must place this file on the MATLAB path. The default CFP template is *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc`.

In each of these fields, click **Browse** to navigate to and select an existing CFP template or CGT file. Click **Edit** to open the specified file into the MATLAB editor where you can customize it.

Code Generation Template (CGT) Files

Code Generation Template (CGT) files define the top-level organization and formatting of generated source code and header files. CGT files have the following applications:

- Generation of custom banners (comments sections) in code files. See “Generate Custom File and Function Banners” on page 55-74.
- Generation of custom code using a CFP template requires a CGT file. To use CFP templates, you must understand the CGT file structure. In many cases, however, you can use the default CGT file without modifying it.

Default CGT file

The code generation software provides a default CGT file, `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`. Base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of one required section and four optional sections:

Code Insertion Section

(Required) This section contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form of:

```
%<SectionName>
```

For example,

```
%<Includes>
```

The code generation software defines a minimal set of required tokens. These tokens generate C or C++ source or header code. They are *built-in* tokens (see “Built-In Tokens and Sections” on page 55-50). You can also define custom tokens on page 55-67 and custom sections on page 55-66.

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding sections appear in the generated code. If you do not include a token, then the corresponding section is not generated. To generate code into a given section, explicitly call the code template API from a CFP template, as described in “Custom File Processing (CFP) Templates” on page 55-55.

The CGT tokens define the high-level organization of generated code. Using the code template API, you can partition each code section into named subsections, as described in “Subsections” on page 55-51.

In the code insertion section, you can also insert C or C++ comments between tokens. Such comments emit directly into the generated code.

File Banner Section

(Optional) This section contains comments and tokens you use in generating a custom file banner.

Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom shared utility function banner.

File Trailer Section

(Optional) This section contains comments for use in generating a custom trailer banner.

For more information on these sections, see “Generate Custom File and Function Banners” on page 55-74.

Built-In Tokens and Sections

The following code extract shows the required code insertion section of the default CGT file with the required built-in tokens.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code insertion section (required)
%%   These are required tokens. You can insert comments and other tokens in
%% between them, but do not change their order or remove them.
%%
%<Includes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Note the following requirements for customizing a CGT file:

- Do not remove required built-in tokens.
- Built-in tokens must appear in the order shown because each successive section has dependencies on previous sections.
- Only one token per line.
- Do not repeat tokens.
- You can add custom tokens and comments to the code insertion section as long as you do not violate the previous requirements.

Note If you modify a CGT file and then rebuild your model, the code generation process does not force a top model build. To regenerate the code, see “Force Regeneration of Top Model Code”.

The following table summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token and Section Name	Description
`Includes`	`#include` directives section
`Defines`	`#define` directives section
`Types`	`typedef` section.`Typedefs` can depend on a previously defined type
`Enums`	Enumerated types section
`Definitions`	Data definitions (for example, `double x = 3.0;`)
`Declarations`	Data declarations (for example, `extern double x;`)
`Functions`	C or C++ functions

Subsections

You can define one or more named subsections for any section. Some of the built-in sections have predefined subsections summarized in table Subsections Defined for Built-In Sections.

Note Sections and subsections emit to the source or header file in the order listed in the CGT file.

Using the custom section feature, you can define additional sections. See “Generate a Custom Section” on page 55-66.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
`Includes`	N/A	
`Defines`	N/A	
`Types`	`IntrinsicTypes`	Intrinsic `typedef` section. Intrinsic types depend only on intrinsic C or C++ types.
`Types`	`PrimitiveTypedefs`	Primitive `typedef` section. Primitive `typedef`s depend only on intrinsic C or C++ types and on `typedef`s previously defined in the `IntrinsicTypes` section.
`Types`	`UserTop`	You can place any type of code in this section, including code that has dependencies on the previous sections.
`Types`	`Typedefs`	`typedef` section. `Typedef`s can depend on previously defined types
`Enums`	N/A	
`Definitions`	N/A	
`Declarations`	N/A	
`Functions`		C or C++ functions
`Functions`	`CompilerErrors`	`#error` directives
`Functions`	`CompilerWarnings`	`#warning` directives
`Functions`	`Documentation`	Documentation (comment) section
`Functions`	`UserBottom`	You can place any code in this section.

Format Generated Code Files by Using Templates

This example shows how to use code generation templates to add custom code banners, rearrange data and functions, and insert additional code segments and documentation into generated code files.

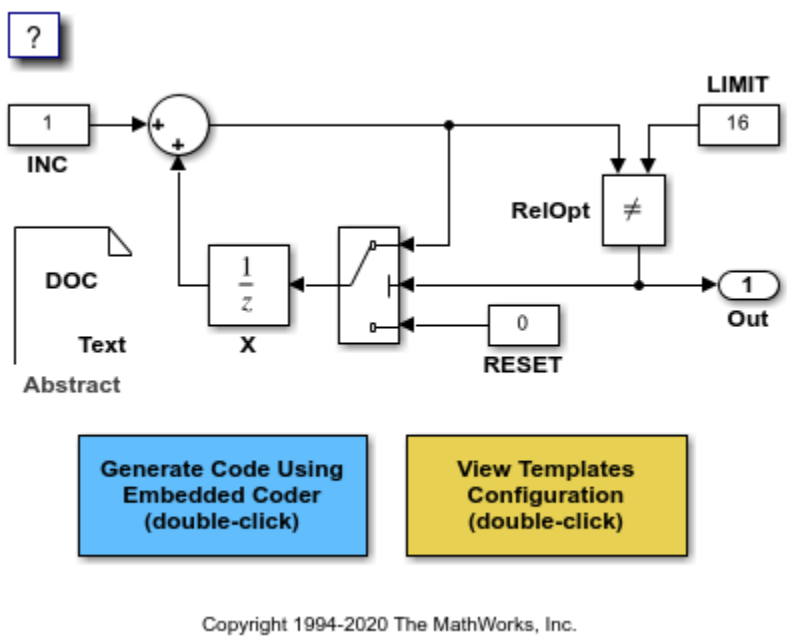
The model extracts text from the Model Description, a DOC block, and a Simulink® annotation. The Simulink annotation maps to the code template `%<Note>` symbol.

The templates provide consistent file layout and elaboration for your organization's coding standards. Embedded Coder® provides extensible code generation templates for formatting generated code. The layout and format of the code template file controls the output of the generated code.

Code generation templates (`.cgt` files) enable you to add custom code banners, rearrange data and functions, and insert additional code segments and documentation into the generated code files. Code generation templates are imported automatically from the MATLAB® path. For this model, the code generation template `rtwdemocodetemplate.cgt` is used.

1. Open model `rtwdemo_codetemplate`.

```
model='rtwdemo_codetemplate';
open_system(model)
```



2. In the Configuration Parameters dialog box, open **Code Generation** > **Templates** pane.

3. In the **Code templates** section, click the **Edit** button beside **Source file template**. The `rtwdemocodetemplate.cgt` template opens in the MATLAB Command Window. Inspect the code template.

4. Build the model and generate code. On the **C Code** tab, click **Build**.

5. Open `rtwdemo_codetemplate.c` to inspect the generated code.

```
rtwdemoclean;
close_system(model,0)
```

Custom File Processing (CFP) Templates

The files provided to support custom file processing are:

- *matlabroot*/rtw/c/tlc/mw/codetemplatelib.tlc: A TLC function library that implements the code template API. codetemplatelib.tlc also provides the comprehensive documentation of the API in the comments headers preceding each function.
- *matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc: An example custom file processing (CFP) template, which you should use as the starting point for creating your own CFP templates. Guidelines and examples for creating a CFP template are provided in "Generate Source and Header Files with a Custom File Processing (CFP) Template" on page 55-62.
- TLC files supporting generation of single-rate and multirate main program modules (see "Customizing Example Main Program Module Generation" on page 55-65).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field. See "Specify Templates For Code Generation" on page 55-48.

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections and Subsections Defined for Built-In Sections.

Code for each section is assembled in buffers and then emitted, in the order listed, to the file being generated.

To generate a file section, your CFP template must first assemble the code to be generated into a buffer. Then, to emit the section, your template calls the TLC function

```
LibSetSourceFileSection(fileH, section, tmpBuf)
```

where

- fileH is a file reference to a file being generated.
- section is the code section or subsection to which code is to be emitted. section must be one of the section or subsection names listed in Subsections Defined for Built-In Sections.

  Determine the section argument as follows:

  - If Subsections Defined for Built-In Sections does not define subsections for a given section, use the section name as the section argument.
  - If Subsections Defined for Built-In Sections defines one or more subsections for a given section, you can use either the section name or a subsection name as the section argument.
  - If you have defined a custom token denoting a custom section, do not call LibSetSourceFileSection. Special API calls are provided for custom sections (see "Generate a Custom Section" on page 55-66).
- tmpBuf is the buffer containing the code to be emitted.

There is no requirement to generate all of the available sections. Your template need only generate the sections you require in a particular file.

Note that legality or syntax checking is not performed on the custom code within each section.

See “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 55-62, for typical usage examples.

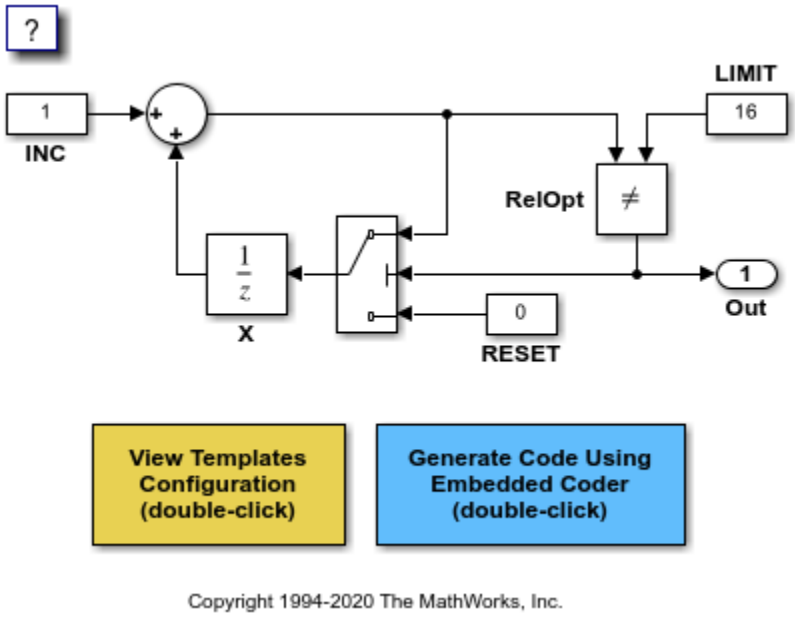
Generate Additional Files Required to Interface with an Application

Embedded Coder® provides an extensible API to augment generated code and to generate additional files. The model configuration parameter **File customization template** specifies a custom Target Language Compiler (TLC) script that executes at the end of the code generation process. Using this script, you can generate additional code to interface with application code.

In this example, the TLC script file `rtwdemofileprocess.tlc` creates an additional entry-point function, `rtwdemo_fileprocess`, which calls the step function and initializes the model, if necessary. The initialization function is transparent to the caller.

1. Open the example model `rtwdemo_fileprocess`.

```
open_system('rtwdemo_fileprocess');
```



2. Verify the model configuration parameter **File customization template** is set to `rtwdemofileprocess.tlc` by double-clicking the yellow button labeled **View Templates Configuration**.

3. Generate code.

4. Inspect the generated code. An additional function, `rtwdemo_fileprocess`, appears at the end of the generated file `rtwdemo_fileprocess.c`. The entry point is declared `extern` in `rtwdemo_fileprocess.h` as the `rtwdemofileprocess.tlc` script specifies.

Change the Organization of a Generated File

The files created during code generation are organized according to the general code generation template. This template has the filename `ert_code_template.cgt`, and is specified by default in **Code Generation** > **Templates** pane of the Configuration Parameters dialog box.

The following fragment shows the `rtwdemo_configinterface.c` file header that is generated using this default template:

```
/*
* File: rtwdemo_configinterface.c
*
* Code generated for Simulink model 'rtwdemo_configinterface'.
*
* Model version                  : 1.387
* Simulink Coder version         : 9.4 (R2020b) 19-May-2020
* C/C++ source code generated on : Thu Jun 11 14:31:07 2020
*
* Target selection: ert.tlc
* Embedded hardware selection: Intel->x86-64 (Windows64)
* Code generation objectives: Unspecified
* Validation result: Not run
*/
```

You can change the organization of generated files using code templates and data templates. Code templates organize the files that contain functions, primarily. Data templates organize the files that contain identifiers. In this procedure, you organize the generated files, using the supplied code and data templates:

1. In the Configuration Parameters dialog box, open the **Code Generation** > **Templates** pane.
2. In the **Code templates** section, in the **Source file template** box, type `code_c_template.cgt`.
3. In the **Header file template** box, type `code_h_template.cgt`.
4. In the **Data templates** section, in the **Source file template** box, type `data_c_template.cgt`.
5. In the **Header file template** box, type `data_h_template.cgt`.
6. Click **Apply**.
7. In the model window, press **Ctrl+B**. Now the files are organized using the templates you specified. For example, the `rtwdemo_configinterface.c` file header now is organized like this:

```
/**
 *****************************************************
 **  FILE INFORMATION:
 **  Filename:           rtwdemo_configinterface.c
 **  File Creation Date: 11-Jun-2020
 **
 **  ABSTRACT:
 **
 **
 **  NOTES:
 **
 **
 **  MODEL INFORMATION:
 **  Model Name:         rtwdemo_configinterface
 **  Model Description:  Data Dictionary Driven Modeling Using
     Simulink(R) Data Objects

   This model shows how to use Simulink data objects to define data
   independently from the diagram which supports a data dictionary
```

```
  driven modeling style.
**  Model Version:      1.387
**  Model Author:       The MathWorks, Inc. - Mon Nov 27 14:36:56 2000
**
**  MODIFICATION HISTORY:
**  Model at Code Generation: user - Thu Jun 11 14:38:31 2020
**
**  Last Saved Modification:  The MathWorks, Inc. - Thu Jun 04 02:05:40 2020
**
**
*****************************************************************************
**/
```

Customize Generated File Names

The code generator creates *`model.*`* files during the code generation and build process. When you generate C/C++ code for ERT-based models, you can customize the names of the generated header, source, and data files. You cannot customize the generated files for referenced models. Custom file names enable you to:

- Comply with naming standards of the company or industry.
- Integrate with external code.

In the Configuration Parameters dialog box, on the **Code Generation** > **Code Placement** pane, use the **Auto-generated file naming rules** group to customize the names of generated files. When you use `Modular` or `Compact(with separate date file)` file packaging, you can specify custom names for generated header, source, and data files. When you use `Compact` file packaging, you can specify custom names for generated header and source files.

1. Open any ERT-based model, for example, `rtwdemo_dynamicio`. In the Configuration Parameters dialog box, change the **System target file** to `ert.tlc`.
2. On the **Code Generation** > **Code Placement** pane, select **File packaging format** as `Compact(with separate date file)`.
3. Specify custom file names by using the naming rules for these parameters:

   - **Header files**: `$R$E_header`
   - **Source files**: `$R$E_source`
   - **Data files**: `$R_data`

   `$E` is mandatory for **Header files** and **Source files**. `$E` represents these instances of file types:

   - `capi`
   - `capi_host`
   - `dt`
   - `testinterface`
   - `private`
   - `types`
4. To build the model, press **Ctrl + B**. In the code generation report, the generated files are listed on the left pane under **Model files** and **Data files** section.

Generated Code

[-] **Main file**

ert_main.c

[-] **Model files**

rtwdemo_dynamicio_header.h

rtwdemo_dynamicio_private_header.h

rtwdemo_dynamicio_source.c

rtwdemo_dynamicio_types_header.h

[-] **Data files**

rtwdemo_dynamicio_data.c

In this example, `$E` resolves to `private` and `types`. The generated header files resolve to the model name with the value for `$E` and custom text `header`. The generated source files resolve to the model name with custom text `source`. The generated data file resolves to the model name with custom text `data`. Here is a summary of naming rules applied and the corresponding generated files:

Type of File	Token Specification	Custom Text	Generated File Name
Header file	`$R$E`	`header`	`rtwdemo_dynamicio_header.h`
Header file	`$R$E`	`header`	`rtwdemo_dynamicio_private_header.h`
Source file	`$R$E`	`source`	`rtwdemo_dynamicio_source.c`
Header file	`$R$E`	`header`	`rtwdemo_dynamicio_types_header.h`
Data file	`$R`	`data`	`rtwdemo_dynamicio_data.c`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement”

Generate Source and Header Files with a Custom File Processing (CFP) Template

In this section...

"Generate Code with a CFP Template" on page 55-62
"Analysis of the Example CFP Template and Generated Code" on page 55-63
"Generate a Custom Section" on page 55-66
"Custom Tokens" on page 55-67

This example shows you the process of generating a simple source (`.c` or `.cpp`) and header (`.h`) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc`, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (`.c` or `.cpp`) and header (`.h`) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, into the standard generated files (for example, *`model`*`.h`)
- Generation of a main program module

Generate Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (`timestwo.c` or `.cpp`) and a header file (`timestwo.h`).

Follow the steps below to become acquainted with the use of CFP templates:

1. Copy the example CFP template, *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc`, to a folder outside of the MATLAB folder structure (that is, not under *`matlabroot`*). If the folder is not on the MATLAB path or the TLC path, then add it to the MATLAB path. It is good practice to locate the CFP template in the same folder as your system target file, which is on the TLC path.
2. Rename the copied `example_file_process.tlc` to `test_example_file_process.tlc`.
3. Open `test_example_file_process.tlc` into the MATLAB editor.
4. Uncomment the following line:

   ```
   %%  %assign ERTCustomFileTest = TLC_TRUE
   ```

   It now reads:

   ```
     %assign ERTCustomFileTest = TLC_TRUE
   ```

   If `ERTCustomFileTest` is not assigned as shown, the CFP template is ignored in code generation.
5. Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.

6. Open the `UserDefinedDataTypes` model.
7. Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Code Generation** pane of the active configuration set.
8. On the **Templates** tab, in the **File customization template** field, specify `test_example_file_process.tlc`. This is the file you previously edited and is now the specified CFP template for your model.
9. On the **General** tab, select the **Generate code only** check box.
10. Click **Apply**.
11. In the model window, press **Ctrl+B**. During code generation, notice the following message in the **Diagnostic Viewer**:

    ```
    Warning:  Overriding example ert_main.c!
    ```

    This message is displayed because `test_example_file_process.tlc` generates the main program module, overriding the default action of the ERT target. This is explained in greater detail below.
12. The `UserDefinedDataTypes` model is configured to generate an HTML code generation report. After code generation is complete, view the report.

    Notice that the **Generated Code** list contains the following files:

    - Under **Main file**, `ert_main.c`.
    - Under **Other files**, `timestwo.c` and `timestwo.h`.

    The files were generated by the CFP template. The next section examines the template to learn how this was done.
13. Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. Refer to the comments in *`matlabroot`*`/rtw/c/tlc/mw/codetemplatelib.tlc` while reading the following discussion.

Generating Code Files

Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers

The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

Subsections Defined for Built-In Sections describes the available file sections and their order in the generated file.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`typesBuf` and `tmpBuf`) to generate two sections (tagged `"Includes"` and `"Functions"`) of the source file `timestwo.c` or `.cpp` (referenced as `cFile`):

```
%openfile typesBuf

#include "rtwtypes.h"

%closefile typesBuf

%<LibSetSourceFileSection(cFile,"Includes",typesBuf)>

 %openfile tmpBuf

 /* Times two function */
 real_T timestwofcn(real_T input) {
   return (input * 2.0);
}

%closefile tmpBuf

%<LibSetSourceFileSection(cFile,"Functions",tmpBuf)>
```

These two sections generate the entire `timestwo.c` or `.cpp` file:

```
#include "rtwtypes.h"

/* Times two function */
FLOAT64 timestwofcn(FLOAT64 input)
{
  return (input * 2.0);
}
```

Adding Code to Standard Generated Files

The `timestwo.c` or `.cpp` file generated in the previous example was independent of the standard code files generated from a model (for example, *`model`*`.c` or `.cpp`, *`model`*`.h`, and so on). You can use similar techniques to generate custom code within the model files. The code template API includes functions to obtain the names of the standard models files and other model-related information. The following excerpt calls `LibGetMdlPubHdrBaseName` to obtain the name for the *`model`*`.h` file. It then obtains a file reference and generates a definition in the `Defines` section of *`model`*`.h`:

```
%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH  = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf

 #define ACCELERATION 9.81

 %closefile tmpBuf

%<LibSetSourceFileSection(modelH,"Defines",tmpBuf)>
```

Examine the generated `UserDefinedDataTypes.h` file to see the generated `#define` directive.

Customizing Example Main Program Module Generation

ERT-based system target files determine whether and how to generate an example main program module (`ert_main.c` or `ert_main.cpp`) based on the settings of the model configuration parameters **Generate an example main program** and **Target operating system**. You can override the default behavior and generate a `main` program customized for your target environment.

Two TLC files support generation of an example `main` program module:

- `bareboard_srmain.tlc`: TLC code for generating a single-rate example `main` program module for a bare-board target environment. TLC function `FcnSingleTaskingMain` generates the code.
- `bareboard_mrmain.tlc`: TLC code for generating a multirate example `main` program module for a bare-board target environment. TLC function `FcnMultiTaskingMain` generates the code.

In the example CFP template file *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc`, this code generates a single- or multitasking example `main` program module. The logic depends on information obtained from the code template calls to `LibIsSingleRateModel` and `LibIsSingleTasking`.

```
%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

 %if LibIsSingleRateModel() || LibIsSingleTasking()
   %include "bareboard_srmain.tlc"
   %<FcnSingleTaskingMain()>
 %else
   %include "bareboard_mrmain.tlc"
   %<FcnMultiTaskingMain()>
 %endif
```

Files `bareboard_srmain.tlc` and `bareboard_mrmain.tlc` use the code template programming interface to generate the example `main` program modules `ert_main.c` or `ert_main.cpp`.

By default, the code generator produces an example `main` program module. To disable generation of that module, set the TLC variable `OverrideSampleERTMain` to `TLC_TRUE`.

For example:

```
%if GenerateSampleERTMain
    %assign CompiledModel.OverrideSampleERTMain = TLC_TRUE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a `SelectCallback` function for your target. A `SelectCallback` function is a MATLAB function that is triggered when you:

- Select a target by setting the **System target file** configuration parameter.
- Build the model.

Your `SelectCallback` clears the model configuration parameter **Generate an example main program**. Clearing the parameter prevents TLC variable `GenerateSampleERTMain` from being set to TLC_TRUE.

For information on how to create a `SelectCallback` function, see “rtwgensettings Structure”.

This code shows how to clear the **Generate an example main program** parameter in the context of a `SelectCallback` function.

```
slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain',0);
hSrc.refreshDialog;
```

Creation of a `main` program for your target environment requires customization. For example, in a bare-board environment, you need to attach `rt_OneStep` to a timer interrupt and customize the generated code, the generating TLC code, or both. For more information, see “Guidelines for Modifying Main Program” on page 71-3 and “Guidelines for Modifying rt_OneStep” on page 71-7.

Generate a Custom Section

You can define custom tokens on page 55-67 in a CGT file and direct generated code into an associated built-in section. This feature gives you additional control over the formatting of code within each built-in section. For example, you could add subsections to built-in sections that do not already define subsections. Custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
  - Assemble code to be generated to the custom section into a buffer.
  - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
  - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

Note If you have not already created custom CGT and CFP files for your model, copy the default template files *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt` and *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc` to a work folder that is outside the MATLAB folder structure but on the MATLAB or TLC path, rename them (for example, add the prefix `test_` to each file), and update the **Templates** pane of the Configuration Parameters dialog box to reference them.

First, add the token `Myincludes` to the code insertion section of your CGT file. For example:

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Next, in the CFP file, add code to generate `include` directives into a buffer. For example, in your copy of the example CFP file, you could insert the following section between the `Includes` section and the `Create a simple main` section:

```
%% Add a custom section to the model's C file model.c

%openfile tmpBuf
#include "moretables1.h"
#include "moretables2.h"
%closefile tmpBuf

%<LibAddSourceFileCustomSection(modelC,"Includes","Myincludes")>
%<LibSetSourceFileCustomSection(modelC,"Myincludes",tmpBuf)>
```

The `LibAddSourceFileCustomSection` function call declares an association between the built-in section `Includes` and the custom section `Myincludes`. `Myincludes` is a subsection of `Includes`. The `LibSetSourceFileCustomSection` function call directs the code in the `tmpBuf` buffer to the `Myincludes` section of the generated file. `LibSetSourceFileCustomSection` is syntactically identical to `LibSetSourceFileSection`.

In the generated code, the include directives generated to the custom section appear after other code directed to `Includes`.

```
#include "UserDefinedDataTypes.h"
#include "UserDefinedDataTypes_private.h"

/* #include "mytables.h" */
#include "moretables1.h"
#include "moretables2.h"
```

Note The placement of the custom token in this example CGT file is arbitrary. By locating `%<Myincludes>` after `%<Includes>`, the CGT file specifies only that the `Myincludes` code appears after `Includes` code.

Custom Tokens

Custom tokens are automatically translated to TLC syntax as a part of the build process. To escape a token, that is to prepare it for normal TLC expansion, use the '!' character. For example, the token `%<!TokenName>` is expanded to `%<TokenName>` by the template conversion program. You can specify valid TLC code, including TLC function calls: `%<!MyTLCFcn()>`.

Custom tokens are not supported when you generate C++ code.

Comparison of a Template and Its Generated File

This figure shows part of a user-modified custom file processing (CFP) template and the resulting generated code. The figure illustrates how you can use a template to:

- Define what code the code generation software should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

Notice `%<Includes>`, for example, on the template. The term `Includes` is a symbol name. A percent sign and brackets (`%< >`) must enclose every symbol name. You can add the desired symbol name (within the `%< >` delimiter) at a particular location in the template. This is how you control where the code generator places an item in the generated file.

Template and Generated File

Mapping Template Specification to Code Generation

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(1)	`/*#INCLUDES*/` `%<Includes>`	26–28	An `/*#INCLUDES*/` comment, followed by `#include` statements	The code generator adds the C/C++ comment as a header, and then interprets the `%<Includes>` template symbol to list the required `#include` statements in the file. This code is first in this section of the file because the template entries are first.
(2)	`/*#DEFINES*/` `%<Defines>`	30	A `/*#DEFINES*/` comment, but no `#define` statements	Next, the code generator places the comment as a header for `#define` statements, but the file does not need `#define`. No code is added.
(3)	`#pragma string1`	31	`#pragma` statements	While the code generator requires `%<>` delimiters for template symbols, it can also interpret C/C++ statements in the template without delimiters. In this case, the generator adds the specified statements to the code, following the order in which the statements appear in the template.
(5)	`#pragma string2`	42		
(4)	`/*DEFINITIONS*/` `%<Definitions>`	32–41	`/*DEFINITIONS*/` comment, followed by definitions	The code generator places the comment and definitions in the file between the `#pragma` statements, according to the order in the template. It also inserts comments (lines 33 and 36) that are preset in the model's Configuration Parameters dialog box.
(6)	`%<Declarations>`	43	No declarations	The file needs no declarations, so the code generator does not generate declarations for this file. The template does not have a comment to provide a header. Line 43 is left blank.
(7)	`%<Functions>`	44–74	Functions	Finally, the code generator adds functions from the model, plus comments that are preset in the Configuration Parameters dialog box. But it adds no comments as a header for the functions, because the template does not have one. This code is last because the template entry is last.

For a list of template symbols and the rules for using them, see “Template Symbol Groups” on page 55-81, “Template Symbols” on page 55-83, and “Rules for Modifying or Creating a Template” on

page 55-85. To set comment options, in the Configuration Parameters dialog box, select the **Code Generation** > **Comments** pane. For details, see “Configure Code Comments”.

Code Template API Summary

Code Template API Functions summarizes the code template API. See the source code in *matlabroot*/rtw/c/tlc/mw/codetemplatelib.tlc for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
`LibClearFileSectionContents`	Clears a file section with custom values before writing file to disk.
`LibGetNumSourceFiles`	Returns the number of created source files (`.c` or `.cpp` and `.h`).
`LibGetSourceFileTag`	Returns `<filename>_h` and `<filename>_c` for header and source files, respectively, where `filename` is the name of the model file.
`LibCreateSourceFile`	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
`LibGetFileRecordName`	Returns a model file name (including the path) without the extension.
`LibGetSourceFileFromIdx`	Returns a model file reference based on its index. This is useful for a common operation on all files, such as to set the leading file banner of all files.
`LibSetSourceFileSection`	Adds to the contents of a specified section within a specified file (see also “Custom File Processing (CFP) Template Structure” on page 55-55).
`LibIndentSourceFile`	Indents a file (from within the TLC environment).
`LibCallModelInitialize`	Returns code for calling the model's *`model`*`_initialize` function (valid for ERT only).
`LibCallModelStep`	Returns code for calling the model's *`model`*`_step` function (valid for ERT only).
`LibCallModelTerminate`	Returns code for calling the model's *`model`*`_terminate` function (valid for ERT only).
`LibCallSetEventForThisBaseStep`	Returns code for calling the model's set events function (valid for ERT only).
`LibWriteModelData`	Returns data for the model (valid for ERT only).
`LibSetRTModelErrorStatus`	Returns the code to set the model error status.
`LibGetRTModelErrorStatus`	Returns the code to get the model error status.
`LibIsSingleRateModel`	Returns true if model is single rate and false otherwise.
`LibGetModelName`	Returns name of the model (without an extension).
`LibGetMdlSrcBaseName`	Returns the name of model's main source file (for example, *`model`*`.c` or `.cpp`).
`LibGetMdlPubHdrBaseName`	Returns the name of model's public header file (for example, *`model`*`.h`).
`LibGetMdlPrvHdrBaseName`	Returns the name of the model's private header file (for example, *`model`*`_private.h`).
`LibIsSingleTasking`	Returns `true` if the model is configured for single-tasking execution.

Function	Description
`LibWriteModelInput`	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
`LibWriteModelOutput`	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).
`LibWriteModelInputs`	Returns the code to write to root inputs (that is, all model inport blocks). (valid for ERT only)
`LibWriteModelOutputs`	Returns the code to write to root outputs (that is, all model outport blocks). (valid for ERT only).
`LibNumDiscreteSampleTimes`	Returns the number of discrete sample times in the model.
`LibSetSourceFileCodeTemplate`	Set the code template to be used for generating a specified source file.
`LibSetSourceFileOutputDirectory`	Set the folder into which a specified source file is to be generated.
`LibAddSourceFileCustomSection`	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`.
`LibSetSourceFileCustomSection`	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with `LibAddSourceFileCustomSection`.

Generate Custom File and Function Banners

Using code generation template (CGT) files, you can specify custom file banners and function banners for the generated code files. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. Use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain characters, which propagate to the generated code.

To specify banners, create a custom CGT file with customized banner sections. The build process creates an executable TLC file from the CGT file. The code generation process then invokes the TLC file.

You do not need to be familiar with TLC programming to generate custom banners. You can modify example files that are supplied with the ERT target.

Note Prior releases supported direct use of customized TLC files as banner templates. You specified these with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. You can still use a custom TLC file banner template, however, you can now use CGT files instead.

ERT template options on the **Code Generation** > **Templates** pane of a configuration set, in the **Code templates** section, support banner generation.

The options for function and file banner generation are:

- Code templates: Source file (*.c) template: CGT file to use when generating source (`.c` or `.cpp`) files. Place this file on the MATLAB path.
- Code templates: Header file (*.h) template: CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path. This file can be the same template specified in the **Code templates: Source file (*.c) template** field, in which case identical banners are generated in source and header files.

  By default, the template for both source and header files is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- In each of these fields, click **Browse** to navigate to and select an existing CGT file for use as a template. Click **Edit** to open the specified file into the MATLAB editor, where you can customize it.

Create a Custom File and Function Banner Template

To customize a CGT file for custom banner generation, make a local copy of the default code template and edit it, as follows:

1. Activate the configuration set that you want to work with.
2. Open the **Code Generation** pane of the active configuration set.
3. Click the **Templates** tab.
4. By default, the code template specified in the **Code templates: Source file (*.c) template** and **Code templates: Header file (*.h) template** fields is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

5. If you want to use a different template as your starting point, click **Browse** to locate and select a CGT file.
6. Click **Edit** button to open the CGT file into the MATLAB editor.
7. Save a local copy of the CGT file. Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If required, add the folder to the MATLAB path.
8. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder.
9. Rename your local copy of the CGT file. When you rename the CGT file, update the associated **Code templates: Source file (*.c) template** or **Code templates: Header file (*.h) template** field to match the new file name.
10. Edit and customize the local copy of the CGT file for banner generation, using the information provided in “Customize a Code Generation Template (CGT) File for File and Function Banner Generation” on page 55-75.
11. Save your changes to the CGT file.
12. Click **Apply** to update the configuration set.
13. Save your model.
14. Generate code. Examine the generated source and header files to confirm that they contain the banners specified by the template or templates.

Customize a Code Generation Template (CGT) File for File and Function Banner Generation

This section describes how to edit a CGT file for custom file and function banner generation. For a description of CGT files, see “Code Generation Template (CGT) Files” on page 55-49.

Components of the File and Function Banner Sections in the CGT file

In a CGT file, you can modify the following sections: file banner, function banner, shared utility function banner, and file trailer. Each section is defined by open and close tags. The tags specific to each section are shown in the following table.

CGT File Section	Open Tag	Close Tag
File Banner on page 55-77	`<FileBanner>`	`</FileBanner>`
Function Banner on page 55-78	`<FunctionBanner>`	`</FunctionBanner>`
Shared-utility Banner on page 55-79	`<SharedUtilityBanner>`	`</SharedUtilityBanner>`
File Trailer on page 55-79	`<FileTrailer>`	`</FileTrailer>`

You can customize your banners by including tokens and comments between the open and close tag for each section. Tokens are typically TLC variables, for example `<ModelVersion>`, which are replaced with values in the generated code.

Note Including C comment indicators, '/*' or a '*/', in the contents of your banner might introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- `width`: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- `style`: specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is as follows:

```
<OpenTag style = "style_value" width = "num_width">
```

Note If the **Configuration Parameters > Code Generation > Language** parameter is set to `C++`, to select a comment style that uses C comment notation (`/*...*/`), you must also set the **Configuration Parameters > Comments > Comment style** parameter to `Multi-line`.

The built-in style options for the `style` attribute are:

- `classic`

```
/* single line comments */
/*
 * multiple line comments
 * second line
 */
```

- `classic_cpp`

```
// single line comments
//
// multiple line comments
// second line
//
```

- `box`

```
/*********************************************************/
/* banner contents                                       */
/*********************************************************/
```

- `box_cpp`

```
///////////////////////////////////////////////////////////
// banner contents                                       //
///////////////////////////////////////////////////////////
```

- `open_box`

```
/*********************************************************
 * banner contents
 *********************************************************/
```

- `open_box_cpp`

```
///////////////////////////////////////////////////////////
// banner contents
///////////////////////////////////////////////////////////
```

- `doxygen`

```
/** single line comments */

/**
 * multiple line comments
 * second line
 */
```

- `doxygen_cpp`

```
/// single line comments

///
/// multiple line comments
/// second line
///
```

- `doxygen_qt`

```
/*! single line comments */

/*!
 * multiple line comments
 * second line
 */
```

- `doxygen_qt_cpp`

```
//! single line comments

//!
//! multiple line comments
//! second line
//!
```

File Banner

This section contains comments and tokens for use in generating a custom file banner. The file banner precedes C or C++ code generated by the model. If you omit the file banner section from the CGT file, then no file banner emits to the generated code.

Note If you customize your file banner, the software does not emit the customized banner for the file `const_params.c`.

The following section is the file banner section provided with the default CGT file, *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file banner section (optional)
%%
<FileBanner style="classic">
File: %<FileName>

Code generated for Simulink model %<ModelName>.

Model version                        : %<ModelVersion>
Simulink Coder version               : %<RTWFileVersion>
C/C++ source code generated on       : %<SourceGeneratedOn>
%<CodeGenSettings>
</FileBanner>
```

Summary of Tokens for File Banner Generation

`FileName`	Name of the generated file (for example, `"UserDefinedDataTypes.c"`).
`FileType`	Either `"source"` or `"header"`. Designates whether generated file is a `.c` or `.cpp` file or an `.h` file.
`FileTag`	Given file names `file.c` or `.cpp` and `file.h`; the file tags are `"file_c"` and `"file_h"`, respectively.
`ModelName`	Name of generating model.
`ModelVersion`	Version number of model.
`Description`	Description of the model specified in the CGT file or the text specified in the **Descripton** tab in **Model Properties** (**Modeling** > **Model Settings**). This token is not supported for referenced models in a model reference hierarchy.
`RTWFileVersion`	Version number of *`model`*`.rtw` file.
`RTWFileGeneratedOn`	Timestamp of *`model`*`.rtw` file.
`TLCVersion`	Version of Target Language Compiler.
`SourceGeneratedOn`	Timestamp of generated file.
`CodeGenSettings`	Code generation settings for model: target language, target selection, production hardware selection, test hardware selection, code generation objectives (in priority order), and Code Generation Advisor validation result.

Function Banner

This section contains comments and tokens for use in generating a custom function banner. The function banner precedes C or C++ function generated during the build process. If you omit the function banner section from the CGT file, the default function banner emits to the generated code. The following section is the default function banner section provided with the default CGT file, *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%%   Customize function banners by using the following predefined tokens:
%% %<ModelName>, %<FunctionName>, %<FunctionDescription>, %<Arguments>,
%% %<ReturnType>, %<GeneratedFor>, %<BlockDescription>.
%%
<FunctionBanner style="classic">
%<FunctionDescription>
%<BlockDescription>
</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

`FunctionName`	Name of function
`Arguments`	List of function arguments
`ReturnType`	Return type of function
`ModelName`	Name of generating model
`FunctionDescription`	Short abstract about the function
`GeneratedFor`	Full block path for the generated function
`BlockDescription`	User input from the **Block Description** parameter of the block properties dialog box. `BlockDescription` contains an optional token attribute, `style`. The only valid value for`style` is `content_only`, which is case-sensitive and enclosed in double quotes. Use the `content_only` style when you want to include only the block description content that you entered in the block parameter dialog. The syntax for the token attribute `style` is: `%<BlockDescription style = "content_only">`

Shared Utility Function Banner

The shared utility function banner section contains comments and tokens for use in generating a custom shared utility function banner. The shared utility function banner precedes C or C++ shared utility function generated during the build process. If you omit the shared utility function banner section from the CGT file, the default shared utility function banner emits to the generated code. The following section is the default shared utility function banner section provided with the default CGT file, *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

```
%%%
```

```
File trailer for generated code.

[EOF]
</FileTrailer>
```

Tokens available for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation.

Template Symbols and Rules

In this section...
“Introduction” on page 55-81 “Template Symbol Groups” on page 55-81 “Template Symbols” on page 55-83 “Rules for Modifying or Creating a Template” on page 55-85

Introduction

“Template Symbol Groups” on page 55-81 and “Template Symbols” on page 55-83 describe custom file processing (CFP) template symbols and rules for using them. The location of a symbol in one of the supplied template files (`code_c_template.cgt`, `code_h_template.cgt`, `data_c_template.cgt`, or `data_h_template.cgt`) determines where the items associated with that symbol are located in the corresponding generated file. “Template Symbol Groups” on page 55-81 identifies the symbol groups, starting with the parent (“Base”) group, followed by the children of each parent. “Template Symbols” on page 55-83 lists the symbols alphabetically.

Note If you are using custom CGT sections, for files generated to the `_sharedutils` folder, you can only use symbol names in the Base symbol group.

Template Symbol Groups

Symbol Group	Symbol Names in This Group

Base (Parents)	`Declarations`
	`Defines`
	`Definitions`
	`Documentation`
	`Enums`
	`Functions`
	`Includes`
	`Types`

Declarations	`ExternalCalibrationLookup1D`
	`ExternalCalibrationLookup2D`
	`ExternalCalibrationScalar`
	`ExternalVariableScalar`

Symbol Group	Symbol Names in This Group
Defines	`LocalDefines` `LocalMacros`

Definitions	`FilescopeCalibrationLookup1D`
	`FilescopeCalibrationLookup2D`
	`FilescopeCalibrationScalar`
	`FilescopeVariableScalar`
	`GlobalCalibrationLookup1D`
	`GlobalCalibrationLookup2D`
	`GlobalCalibrationScalar`
	`GlobalVariableScalar`

Documentation	`Abstract`
	`Banner`
	`Created`
	`Creator`
	`Date`
	`Description`
	`FileName`
	`History`
	`LastModifiedDate`
	`LastModifiedBy`
	`ModelName`
	`ModelVersion`
	`ModifiedBy`

	`Notes` `ToolVersion`
Functions	`CFunctionCode`
Types	This parent has no children.

Template Symbols

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Abstract`	Documentation	N/A	User-supplied description of the model or file. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Banner`	Documentation	N/A	Comments located near top of the file. Contains information that includes model and software versions, and date file was generated.
`CFunctionCode`	Functions	File	C/C++ functions. Must be at the bottom of the template.
`Created`	Documentation	N/A	Date when model was created. From **Created on** field on Model Properties dialog box.
`Creator`	Documentation	N/A	User who created model. From **Created by** field on Model Properties dialog box.
`Date`	Documentation	N/A	Date file was generated. Taken from computer clock.
`Declarations`	Base		Data declaration of a signal or parameter. For example, `extern real_T globalvar;`.
`Defines`	Base	File	Required `#defines` of `.h` files.
`Definitions`	Base	File	Data definitions of signals or parameters.
`Description`	Documentation	N/A	Description of model. From **Model description** field on Model Properties dialog box.**
`Documentation`	Base	N/A	Comments about how to interpret the generated files.
`Enums`	Base	File	Enumerated data type definitions.
`ExternalCalibrationLookup1D`	Declarations	External	***
`ExternalCalibrationLookup2D`	Declarations	External	***
`ExternalCalibrationScalar`	Declarations	External	***
`ExternalVariableScalar`	Declarations	External	***
`FileName`	Documentation	N/A	Name of the generated file.
`FilescopeCalibrationLookup1D`	Definitions	File	***

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`FilescopeCalibrationLookup2D`	Definitions	File	***
`FilescopeCalibrationScalar`	Definitions	File	***
`FilescopeVariableScalar`	Definitions	File	***
`Functions`	Base	File	Generated function code.
`GlobalCalibrationLookup1D`	Definitions	Global	***
`GlobalCalibrationLookup2D`	Definitions	Global	***
`GlobalCalibrationScalar`	Definitions	Global	***
`GlobalVariableScalar`	Definitions	Global	***
`History`	Documentation	N/A	User-supplied revision history of the generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Includes`	Base	File	`#include` preprocessor directives.
`LastModifiedDate`	Documentation	N/A	Date when model was last saved. From **Last saved on** field on Model Properties dialog box.
`LastModifiedBy`	Documentation	N/A	User who last saved model. From **Last saved by** field on Model Properties dialog box.
`LocalDefines`	Defines	File	`#define` preprocessor directives from code-generation data objects.
`LocalMacros`	Defines	File	C/C++ macros local to the file.
`ModelName`	Documentation	N/A	Name of the model.
`ModelVersion`	Documentation	N/A	Number incremented by 1 each time you save the model.
`ModifiedBy`	Documentation	N/A	Name of user who last modified the model.
`Notes`	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`ToolVersion`	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
`Types`	Base		Data types of generated code.

* Symbol names must be enclosed between `%<  >`. For example, `%<Functions>`.

** This symbol can be used to add a comment to the generated files. See “Add Global Comments in the Generated Code” on page 55-5. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

*** The description can be deduced from the symbol name. For example, `GlobalCalibrationScalar` is a symbol that identifies a scalar. It contains data of global scope that you can calibrate.

Rules for Modifying or Creating a Template

The following are the rules for creating an MPF template. “Comparison of a Template and Its Generated File” on page 55-68 illustrates several of these rules.

1. Place a symbol on a template within the `%< >` delimiter. For example, the symbol named `Includes` should look like this on a template: `%<Includes>`. *Note that symbol names are case-sensitive.*
2. Place a symbol on a template where desired. Its location on the template determines where the item associated with this symbol is located in the generated file. If no item is associated with it, the symbol is ignored.
3. Place a C/C++ statement outside of the `%< >` delimiter, and on a different line than a `%< >` delimiter, for that statement to appear in the generated file. For example, `#pragma message ("my text")` in the template results in `#pragma message ("my text")` at the corresponding location in the generated file. Note that the statement must be compatible with your C/C++ compiler.
4. Use the `.cgt` extension for every template filename. ("`cgt`" stands for *c*ode *g*eneration *t*emplate.)
5. Note that `%% $Revision: 1.1.4.10.4.1 $` appears at the top of the MathWorks supplied templates. This is for internal MathWorks use only. It does not need to be placed on a user-defined template and does not show in a generated file.
6. Place a comment on the template between `/* */` as in standard ANSI C[6]. This results in `/*comment*/` on the generated file.
7. Each MPF template must have all of the Base group symbols, in predefined order. They are listed in “Template Symbol Groups” on page 55-81. Each symbol in the Base group is a parent. For example, `Declarations` is a parent symbol.
8. Each symbol in a non-Base group is a child. For example, `LocalMacros` is a child.
9. Except for Documentation children, children must be placed after their parent, before the next parent, and before the `Functions` symbol.
10. Documentation children can be located before or after their parent in any order anywhere in the template.
11. If a non-Documentation child is missing from the template, the code generator places the information associated with this child at its parent location in the generated file.
12. If a Documentation child is missing from the template, the code generator omits the information associated with that child from the generated file.

6 ANSI is a registered trademark of the American National Standards Institute, Inc.

Annotate Code for Justifying Polyspace Checks

This example shows how to turn on comments in the generated code that suppress error detection through Polyspace®.

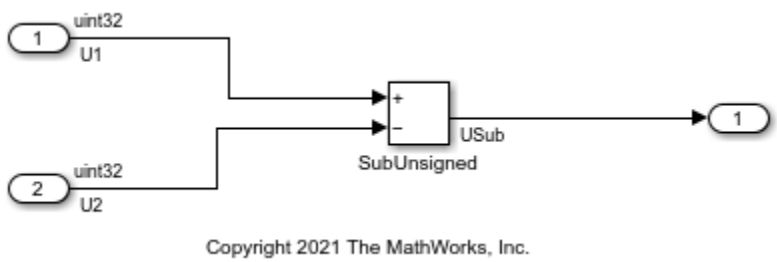
Polyspace Checks

With the Polyspace Code Prover™ software you can apply Polyspace verification to Embedded Coder® generated code. The software detects run-time errors in the generated code and helps you to locate and fix model faults. Polyspace might highlight overflows for certain operations that are legitimate because of the way the code generator implements these operations. These legitimate overflows should not be flagged by Polyspace. Generate comments in the code that suppress this error detection.

Code Generation

1. Open the example Simulink® model `mSatAddSub`.

```
model='mSatAddSub';
open_system(model);
```



2. In the Embedded Coder app on the **Configuration Parameters** dialog box, turn off **Operator annotations**.

```
set_param('mSatAddSub','OperatorAnnotations','off');
```

3. To build the model and generate code, press **Ctrl+B**.

```
evalc('slbuild(''mSatAddSub'')');
```

```
%The generated code is:
file = fullfile('mSatAddSub_ert_rtw','mSatAddSub.c');
rtwdemodbtype(file,'/* Model step function */',...
    'End of Sum',1,1);
```

```
/* Model step function */
void mSatAddSub_step(void)
{
  uint32_T qY;

  /* Sum: '<Root>/SubUnsigned' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
```

```
  qY = U1 - U2;
  if (qY > U1) {
    qY = 0U;
  }

  /* Sum: '<Root>/SubUnsigned' */
  USub = qY;

  /* Outport: '<Root>/Out3' */
  mSatAddSub_Y.Out3 = USub;

  /* Matfile logging */
  rt_UpdateTXYLogVars(mSatAddSub_M->rtwLogInfo, (&mSatAddSub_M->Timing.taskTime0));

  /* signal main to stop simulation */
  {                                   /* Sample time: [1.0s, 0.0s] */
    if ((rtmGetTFinal(mSatAddSub_M)!=-1) &&
        !((rtmGetTFinal(mSatAddSub_M)-mSatAddSub_M->Timing.taskTime0) >
          mSatAddSub_M->Timing.taskTime0 * (DBL_EPSILON))) {
      rtmSetErrorStatus(mSatAddSub_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   */
  mSatAddSub_M->Timing.taskTime0 =
    ((time_T)(++mSatAddSub_M->Timing.clockTick0)) *
    mSatAddSub_M->Timing.stepSize0;
}

/* Model initialize function */
void mSatAddSub_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)mSatAddSub_M, 0,
                sizeof(RT_MODEL_mSatAddSub));
  rtmSetTFinal(mSatAddSub_M, 1.0);
  mSatAddSub_M->Timing.stepSize0 = 1.0;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    mSatAddSub_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(mSatAddSub_M->rtwLogInfo, (NULL));
```

```
rtliSetLogXSignalPtrs(mSatAddSub_M->rtwLogInfo, (NULL));
rtliSetLogT(mSatAddSub_M->rtwLogInfo, "tout");
rtliSetLogX(mSatAddSub_M->rtwLogInfo, "");
rtliSetLogXFinal(mSatAddSub_M->rtwLogInfo, "");
rtliSetLogVarNameModifier(mSatAddSub_M->rtwLogInfo, "rt_");
rtliSetLogFormat(mSatAddSub_M->rtwLogInfo, 1);
rtliSetLogMaxRows(mSatAddSub_M->rtwLogInfo, 1000);
rtliSetLogDecimation(mSatAddSub_M->rtwLogInfo, 1);

/*
 * Set pointers to the data and signal info for each output
 */
{
  static void * rt_LoggedOutputSignalPtrs[] = {
    &mSatAddSub_Y.Out3
  };

  rtliSetLogYSignalPtrs(mSatAddSub_M->rtwLogInfo, ((LogSignalPtrsType)
    rt_LoggedOutputSignalPtrs));
}

{
  static int_T rt_LoggedOutputWidths[] = {
    1
  };

  static int_T rt_LoggedOutputNumDimensions[] = {
    1
  };

  static int_T rt_LoggedOutputDimensions[] = {
    1
  };

  static boolean_T rt_LoggedOutputIsVarDims[] = {
    0
  };

  static void* rt_LoggedCurrentSignalDimensions[] = {
    (NULL)
  };

  static int_T rt_LoggedCurrentSignalDimensionsSize[] = {
    4
  };

  static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
    SS_UINT32
  };

  static int_T rt_LoggedOutputComplexSignals[] = {
    0
  };

  static RTWPreprocessingFcnPtr rt_LoggingPreprocessingFcnPtrs[] = {
    (NULL)
  };
```

```
    static const char_T *rt_LoggedOutputLabels[] = {
      "USub" };

    static const char_T *rt_LoggedOutputBlockNames[] = {
      "mSatAddSub/Out3" };

    static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
      { 0, SS_UINT32, SS_UINT32, 0, 0, 0, 1.0, 0, 0.0 }
    };

    static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
      {
        1,
        rt_LoggedOutputWidths,
        rt_LoggedOutputNumDimensions,
        rt_LoggedOutputDimensions,
        rt_LoggedOutputIsVarDims,
        rt_LoggedCurrentSignalDimensions,
        rt_LoggedCurrentSignalDimensionsSize,
        rt_LoggedOutputDataTypeIds,
        rt_LoggedOutputComplexSignals,
        (NULL),
        rt_LoggingPreprocessingFcnPtrs,

        { rt_LoggedOutputLabels },
        (NULL),
        (NULL),
        (NULL),

        { rt_LoggedOutputBlockNames },

        { (NULL) },
        (NULL),
        rt_RTWLogDataTypeConvert
      }
    };

    rtliSetLogYSignalInfo(mSatAddSub_M->rtwLogInfo, rt_LoggedOutputSignalInfo);

    /* set currSigDims field */
    rt_LoggedCurrentSignalDimensions[0] = &rt_LoggedOutputWidths[0];
  }

  rtliSetLogY(mSatAddSub_M->rtwLogInfo, "yout");
}

/* block I/O */

/* exported global signals */
USub = 0U;

/* external inputs */
U1 = 0U;
U2 = 0U;

/* external outputs */
mSatAddSub_Y.Out3 = 0U;
```

```
  /* Matfile logging */
  rt_StartDataLoggingWithStartTime(mSatAddSub_M->rtwLogInfo, 0.0, rtmGetTFinal
    (mSatAddSub_M), mSatAddSub_M->Timing.stepSize0, (&rtmGetErrorStatus
    (mSatAddSub_M)));
}

/* Model terminate function */
void mSatAddSub_terminate(void)
{
  /* (no terminate code required) */
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

4. Run the Polyspace check. The code generator recognizes that the largest built-in data type is 32-bit. It is not possible to saturate all the results of the subtractions using `0U` and a bigger single-word integer data type. Instead the software detects the results overflow and the direction of the overflow, and saturates the result. The operation is flagged by Polyspace because it fails to recognize the saturation.

5. In the Embedded Coder app on the **Configuration Parameters** dialog box, turn off **Operator annotations**.

```
set_param('mSatAddSub','OperatorAnnotations',"on");
evalc('slbuild(''mSatAddSub'')');
```

6. The generated code is:

```
file = fullfile('mSatAddSub_ert_rtw','mSatAddSub.c');
rtwdemodbtype(file,'/* Model step function */',...
    'End of Sum',1,1);
```

```
/* Model step function */
void mSatAddSub_step(void)
{
  uint32_T qY;

  /* Sum: '<Root>/SubUnsigned' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  qY = U1 - /*MW:OvSatOk*/ U2;
  if (qY > U1) {
    qY = 0U;
  }

  /* Sum: '<Root>/SubUnsigned' */
  USub = qY;

  /* Outport: '<Root>/Out3' */
  mSatAddSub_Y.Out3 = USub;

  /* Matfile logging */
```

```
  rt_UpdateTXYLogVars(mSatAddSub_M->rtwLogInfo, (&mSatAddSub_M->Timing.taskTime0));

  /* signal main to stop simulation */
  {                                   /* Sample time: [1.0s, 0.0s] */
    if ((rtmGetTFinal(mSatAddSub_M)!=-1) &&
        !((rtmGetTFinal(mSatAddSub_M)-mSatAddSub_M->Timing.taskTime0) >
          mSatAddSub_M->Timing.taskTime0 * (DBL_EPSILON))) {
      rtmSetErrorStatus(mSatAddSub_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   */
  mSatAddSub_M->Timing.taskTime0 =
    ((time_T)(++mSatAddSub_M->Timing.clockTick0)) *
    mSatAddSub_M->Timing.stepSize0;
}

/* Model initialize function */
void mSatAddSub_initialize(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)mSatAddSub_M, 0,
                sizeof(RT_MODEL_mSatAddSub));
  rtmSetTFinal(mSatAddSub_M, 1.0);
  mSatAddSub_M->Timing.stepSize0 = 1.0;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    mSatAddSub_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(mSatAddSub_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(mSatAddSub_M->rtwLogInfo, (NULL));
    rtliSetLogT(mSatAddSub_M->rtwLogInfo, "tout");
    rtliSetLogX(mSatAddSub_M->rtwLogInfo, "");
    rtliSetLogXFinal(mSatAddSub_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(mSatAddSub_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(mSatAddSub_M->rtwLogInfo, 1);
    rtliSetLogMaxRows(mSatAddSub_M->rtwLogInfo, 1000);
    rtliSetLogDecimation(mSatAddSub_M->rtwLogInfo, 1);

    /*
     * Set pointers to the data and signal info for each output
     */
```

```
{
  static void * rt_LoggedOutputSignalPtrs[] = {
    &mSatAddSub_Y.Out3
  };

  rtliSetLogYSignalPtrs(mSatAddSub_M->rtwLogInfo, ((LogSignalPtrsType)
    rt_LoggedOutputSignalPtrs));
}

{
  static int_T rt_LoggedOutputWidths[] = {
    1
  };

  static int_T rt_LoggedOutputNumDimensions[] = {
    1
  };

  static int_T rt_LoggedOutputDimensions[] = {
    1
  };

  static boolean_T rt_LoggedOutputIsVarDims[] = {
    0
  };

  static void* rt_LoggedCurrentSignalDimensions[] = {
    (NULL)
  };

  static int_T rt_LoggedCurrentSignalDimensionsSize[] = {
    4
  };

  static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
    SS_UINT32
  };

  static int_T rt_LoggedOutputComplexSignals[] = {
    0
  };

  static RTWPreprocessingFcnPtr rt_LoggingPreprocessingFcnPtrs[] = {
    (NULL)
  };

  static const char_T *rt_LoggedOutputLabels[] = {
    "USub" };

  static const char_T *rt_LoggedOutputBlockNames[] = {
    "mSatAddSub/Out3" };

  static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
    { 0, SS_UINT32, SS_UINT32, 0, 0, 0, 1.0, 0, 0.0 }
  };

  static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
    {
```

```
            1,
            rt_LoggedOutputWidths,
            rt_LoggedOutputNumDimensions,
            rt_LoggedOutputDimensions,
            rt_LoggedOutputIsVarDims,
            rt_LoggedCurrentSignalDimensions,
            rt_LoggedCurrentSignalDimensionsSize,
            rt_LoggedOutputDataTypeIds,
            rt_LoggedOutputComplexSignals,
            (NULL),
            rt_LoggingPreprocessingFcnPtrs,

            { rt_LoggedOutputLabels },
            (NULL),
            (NULL),
            (NULL),

            { rt_LoggedOutputBlockNames },

            { (NULL) },
            (NULL),
            rt_RTWLogDataTypeConvert
          }
        };

        rtliSetLogYSignalInfo(mSatAddSub_M->rtwLogInfo, rt_LoggedOutputSignalInfo);

        /* set currSigDims field */
        rt_LoggedCurrentSignalDimensions[0] = &rt_LoggedOutputWidths[0];
      }

      rtliSetLogY(mSatAddSub_M->rtwLogInfo, "yout");
    }

    /* block I/O */

    /* exported global signals */
    USub = 0U;

    /* external inputs */
    U1 = 0U;
    U2 = 0U;

    /* external outputs */
    mSatAddSub_Y.Out3 = 0U;

    /* Matfile logging */
    rt_StartDataLoggingWithStartTime(mSatAddSub_M->rtwLogInfo, 0.0, rtmGetTFinal
      (mSatAddSub_M), mSatAddSub_M->Timing.stepSize0, (&rtmGetErrorStatus
      (mSatAddSub_M)));
  }

  /* Model terminate function */
  void mSatAddSub_terminate(void)
  {
    /* (no terminate code required) */
  }
```

```
/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

The code generator inserts the `/*MW:OvSatOk*/` comment that suppresses the error detected by Polyspace.

7. Close the model.

```
bdclose(model)
rtwdemoclean;
```

See Also

Related Examples

- “Model Configuration Parameters: Comments”
- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)

Enhance Readability of Code for Flow Charts

In this section...

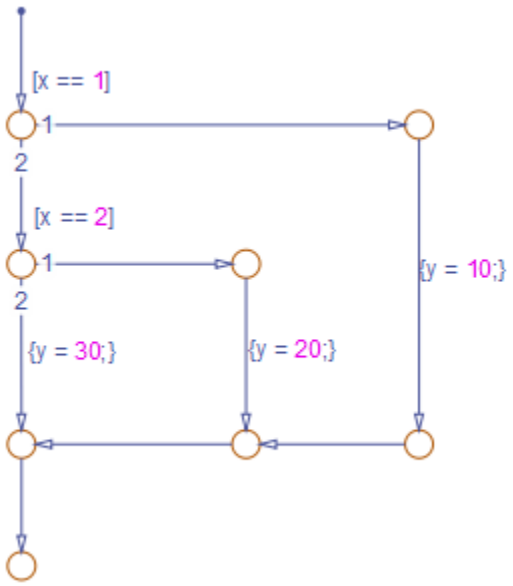
"Appearance of Generated Code for Flow Charts" on page 55-95
"Convert If-Elseif-Else Code to Switch-Case Statements" on page 55-97
"Example of Converting Code to Switch-Case Statements" on page 55-99

Appearance of Generated Code for Flow Charts

If you have Embedded Coder and you generate code for models that include Stateflow objects, the code from a flow chart resembles the samples that follow.

The following characteristics apply:

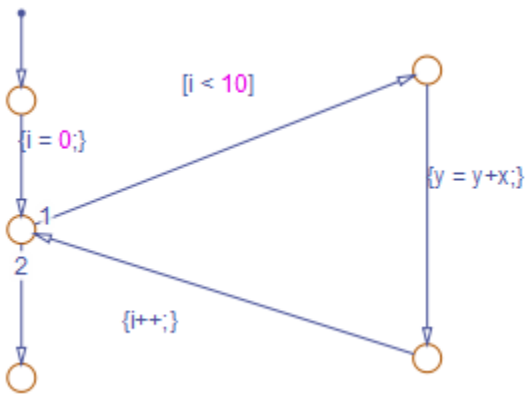
- By default, the generated code uses `if-elseif-else` statements to represent `switch` patterns. To convert the code to use `switch-case` statements, see "Convert If-Elseif-Else Code to Switch-Case Statements" on page 55-97.
- By default, variables that appear in the flow chart do not retain their names in the generated code. Modified identifiers make sure that no naming conflicts occur.
- Traceability comments for the transitions appear between each set of `/*` and `*/` markers. To learn more about traceability, see "Trace Stateflow Elements in Generated Code" on page 85-12.




```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  modelname_Y.Out1 = 10.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
```

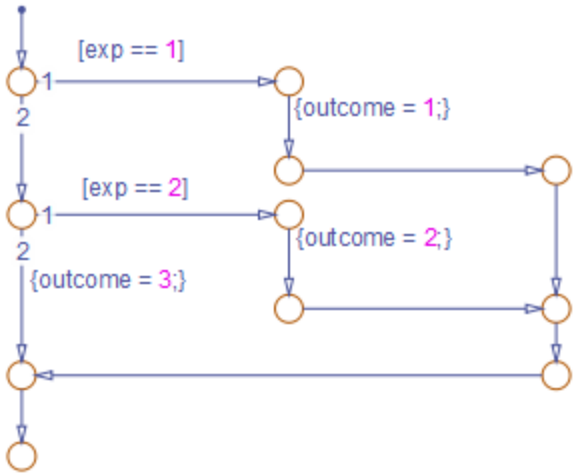
```
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
    modelname_Y.Out1 = 20.0;

    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    modelname_Y.Out1 = 30.0;
  }
}
```

Sample Code for a Decision Logic Pattern




```
for (sf_i = 0; sf_i < 10; sf_i++) {
  /* Transition: '<S1>:40' */
  /* Transition: '<S1>:41' */
  modelname_B.y = modelname_B.y +
    modelname_U.In1;

  /* Transition: '<S1>:39' */
}
```

Sample Code for an Iterative Loop Pattern



```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:149' */
  /* Transition: '<S1>:150' */
  modelname_Y.Out1 = 1.0;

  /* Transition: '<S1>:151' */
  /* Transition: '<S1>:152' */
  /* Transition: '<S1>:158' */
  /* Transition: '<S1>:159' */
} else {
  /* Transition: '<S1>:156' */
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:153' */
    /* Transition: '<S1>:154' */
    modelname_Y.Out1 = 2.0;

    /* Transition: '<S1>:155' */
    /* Transition: '<S1>:158' */
    /* Transition: '<S1>:159' */
  } else {
    /* Transition: '<S1>:161' */
    modelname_Y.Out1 = 3.0;
  }
}
```

Sample Code for a Switch Pattern

Convert If-Elseif-Else Code to Switch-Case Statements

When you generate code for embedded real-time targets, you can choose to convert `if-elseif-else` code to `switch-case` statements. This conversion can enhance readability of the code. For example, when a flow chart contains a long list of conditions, the `switch-case` structure:

- Reduces the use of parentheses and braces
- Minimizes repetition in the generated code

How to Convert If-Elseif-Else Code to Switch-Case Statements

Step	Task	Reference
1	Verify that your flow chart follows the rules for conversion.	“Verify the Contents of the Flow Chart” on page 55-100
2	Enable the conversion and generate code for your model.	“Enable the Conversion” on page 55-101
3	Troubleshoot the generated code. • If you see `switch-case` statements for your flow chart, you can stop. • If you see `if-elseif-else` statements for your flow chart, update the chart and repeat the previous step.	“Troubleshoot the Generated Code” on page 55-101

Rules of Conversion

For the conversion to occur, following these guidelines. LHS and RHS refer to the left-hand side and right-hand side of a condition, respectively.

Construct	Rules to Follow
Flow chart	Must have a default and two or more *unique* conditions. For more information, see “Duplicate Conditions” on page 55-98.
Each condition	Must test equality only.
	Must use the same variable or expression for the LHS. You can reverse the LHS and RHS.
Each LHS	Must be a single variable or expression.
	Cannot be a constant.
	Must have an integer or enumerated data type.
	Cannot have any side effects on simulation. For example, the LHS can read from but not write to global variables.
Each RHS	Must be a constant or a parameter.
	Must have an integer or enumerated data type.

Duplicate Conditions

If a flow chart has duplicate conditions, the conversion preserves only the first condition. The code discards the other instances of duplicate conditions.

After removal of duplicates, two or more unique conditions must exist. If not, the conversion does not occur and the code contains all duplicate conditions.

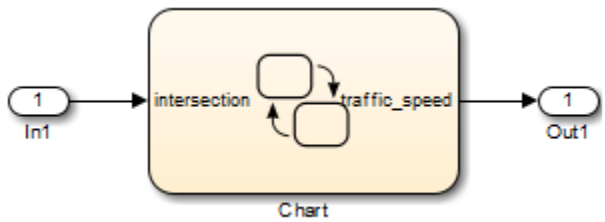
Example of Generated Code	Code After Conversion

`if (x == 1) {`
`  block1`
`} else if (x == 2) {`
`  block2`
`} else if (x == 1) {  // duplicate`
`  block3`
`} else if (x == 3) {`
`  block4`
`} else if (x == 1) {  // duplicate`
`  block5`
`} else {`
`  block6`
`}`
	`switch (x) {`
	`  case 1:`
	`  block1; break;`
	`  case 2:`
	`  block2; break;`
	`  case 3:`
	`  block4; break;`
	`  default:`
	`  block6; break;`
	`}`

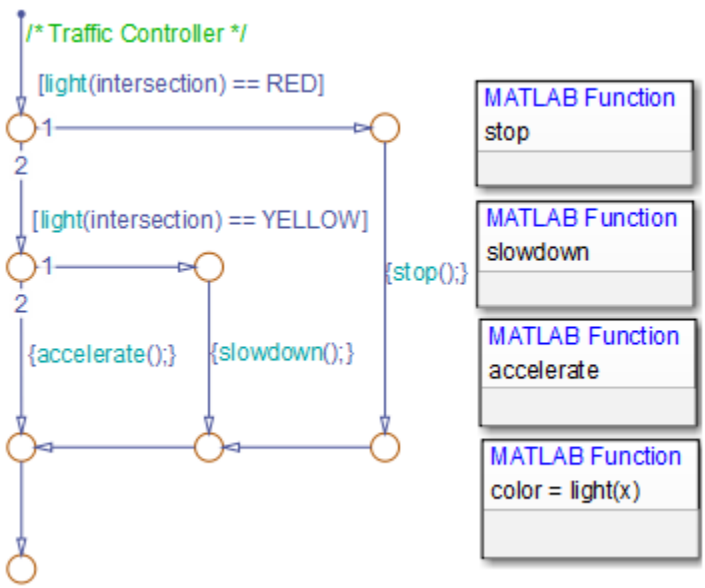
`if (x == 1) {`
`  block1`
`} else if (x == 1) {  // duplicate`
`  block2`
`} else {`
`  block3`
`}`
	No change, because only one unique condition exists

Example of Converting Code to Switch-Case Statements

Suppose that you have this model containing a single chart.



The chart contains a flow chart and four MATLAB functions.



The MATLAB functions in the chart contain the code listed in the table. In each case, the **Function Inline Option** is `Auto`. For more information about function inlining, see “Specify Properties of Graphical Functions” (Stateflow).

MATLAB Function	Code

`stop`	`function stop`
	`%#codegen`
	`coder.extrinsic('disp');`
	`disp('Not moving.')`
	
	`traffic_speed = 0;`

`slowdown`	`function slowdown`
	`%#codegen`
	`coder.extrinsic('disp')`
	`disp('Slowing down.')`
	
	`traffic_speed = 1;`

MATLAB Function	Code

accelerate	`function accelerate`
	`%#codegen`
	`coder.extrinsic('disp');`
	`disp('Moving along.')`
	
	`traffic_speed = 2;`

light	`function color = light(x)`
	`%#codegen`
	`if (x < 20)`
	`  color = TrafficLights.GREEN;`
	`elseif (x >= 20 && x < 25)`
	`  color = TrafficLights.YELLOW;`
	`else`
	`  color = TrafficLights.RED;`
	`end`

The output `color` of the function `light` uses the enumerated type `TrafficLights`. The enumerated type definition in `TrafficLights.m` is:

```
classdef TrafficLights < Simulink.IntEnumType
  enumeration
    RED(0)
    YELLOW(5)
    GREEN(10)
  end
end
```

For more information, see “Define Enumerated Data Types” (Stateflow).

Verify the Contents of the Flow Chart

Check that the flow chart in your Stateflow chart follows the rules in “Rules of Conversion” on page 55-98.

Construct	How the Construct Follows the Rules
Flow chart	A default condition and two unique conditions exist: • `[light(intersection) == RED]` • `[light(intersection) == YELLOW]`
Each condition	Each condition: • Tests equality • Uses the same function call `light(intersection)` for the LHS
Each LHS	Each LHS: • Contains a single expression • Is the output of a function call, not a constant • Is of enumerated type `TrafficLights`, which you define in `TrafficLights.m` on the MATLAB path. See “Define Enumerated Data Types” (Stateflow). • Uses a function call that does not have side effects

Construct	How the Construct Follows the Rules
Each RHS	Each RHS: • Is an enumerated value and therefore a constant • Is of enumerated type `TrafficLights`

Enable the Conversion

1. Open the Model Configuration Parameters dialog box.
2. In the **Code Generation** pane, select `ert.tlc` for the **System target file** for an ERT-based target for your model.
3. In the **Code Generation** > **Code Style** pane, select the **Convert if-elseif-else patterns to switch-case statements** check box.

   If you select this check box, the conversion applies to:

   - Flow charts in all charts of a model
   - MATLAB functions in all charts of a model
   - All MATLAB Function blocks in that model
4. In the model, on the **C Code** tab, click **Build** to generate source code from the model.

Troubleshoot the Generated Code

The generated code for the flow chart appears like this code:

```
if (sf_color == RED) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  rtb_traffic_speed = 0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
  /* MATLAB Function 'light': '<S1>:19' */
  if (ifelse_using_enums_U.In1 < 20.0) {
    /* '<S1>:19:3' */
    /* '<S1>:19:4' */
    sf_color = GREEN;
  } else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
    /* '<S1>:19:5' */
    /* '<S1>:19:6' */
    sf_color = YELLOW;
  } else {
    /* '<S1>:19:8' */
    sf_color = RED;
  }

  if (sf_color == YELLOW) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
```

```
    /* MATLAB Function 'slowdown': '<S1>:24' */
    /* '<S1>:24:6' */
    rtb_traffic_speed = 1;

    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    /* MATLAB Function 'accelerate': '<S1>:25' */
    /* '<S1>:25:6' */
    rtb_traffic_speed = 2;
  }
}
```

Because the MATLAB function `light` appears inlined, inequality comparisons appear in these lines of code:

```
if (ifelse_using_enums_U.In1 < 20.0) {
....
} else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
....
```

Because inequalities appear in the body of the `if-elseif-else` code for the flow chart, the conversion to `switch-case` statements does not occur. Do one of the following:

- Specify that the function `light` does not appear inlined. See “Change the Inlining Property for the Function” on page 55-102.
- Modify the flow chart. See “Modify the Flow Chart to Ensure Switch-Case Statements” on page 55-103.

Change the Inlining Property for the Function

If you do not want to modify your flow chart, change the inlining property for the function `light`:

1. Right-click the function box for `light` and select **Properties**.
2. In the properties dialog box, for **Function Inline Option**, select `Function`.
3. Click **OK** to close the dialog box.

Note You do not have to change the inlining property for the other three MATLAB functions in the chart. Because the flow chart does not call those functions during evaluation of conditions, the inlining property for those functions can remain `Auto`.

When you regenerate code for your model, the code for the flow chart now appears like this code:

```
switch (ifelse_using_enums_light(ifelse_using_enums_U.In1)) {
 case RED:
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  ifelse_using_enums_Y.Out1 = 0.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
```

```
  break;

 case YELLOW:
  /* Transition: '<S1>:10' */
  /* Transition: '<S1>:13' */
  /* Transition: '<S1>:14' */
  /* MATLAB Function 'slowdown': '<S1>:24' */
  /* '<S1>:24:6' */
  ifelse_using_enums_Y.Out1 = 1.0;

  /* Transition: '<S1>:16' */
  break;

 default:
  /* Transition: '<S1>:17' */
  /* MATLAB Function 'accelerate': '<S1>:25' */
  /* '<S1>:25:6' */
  ifelse_using_enums_Y.Out1 = 2.0;
  break;
}
```

Because the MATLAB function `light` no longer appears inlined, the conversion to `switch-case` statements occurs. To enhance readability, the `switch-case` statements provide these benefits:

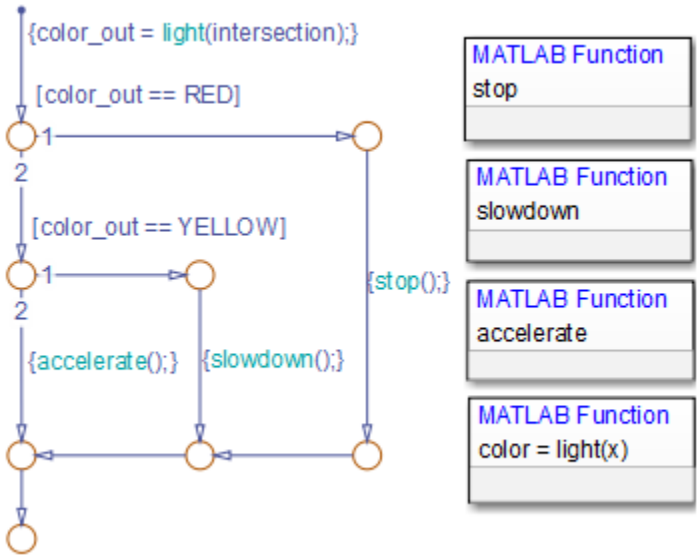
- The code reduces the use of parentheses and braces.
- The LHS expression `ifelse_using_enums_light(ifelse_using_enums_U.In1)` appears only once, minimizing repetition in the code.

Modify the Flow Chart to Ensure Switch-Case Statements

If you do not want to change the inlining property for the function `light`, modify your flow chart:

1. Add chart local data `color_out` with the enumerated type `TrafficLights`.
2. Replace each instance of `light(intersection)` with `color_out`.
3. Add the action `{color_out = light(intersection)}` to the default transition of the flow chart.

The chart now looks similar to this chart:



When you regenerate code for your model, the code for the flow chart uses `switch-case` statements.

See Also

More About

- “Model Configuration Parameters: Code Style”

Enhance Code Readability for MATLAB Function Blocks

In this section...

"Requirements for Using Readability Optimizations" on page 55-105
"Converting If-Elseif-Else Code to Switch-Case Statements" on page 55-105
"Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements" on page 55-107

Requirements for Using Readability Optimizations

To use readability optimizations in your code, you must have an Embedded Coder license. These optimizations appear only in code that you generate for an embedded real-time (`ert`) target.

Note These optimizations do not apply to MATLAB files that you call from the MATLAB Function block.

For more information, see "Configure a System Target File" on page 50-2.

Converting If-Elseif-Else Code to Switch-Case Statements

When you generate code for embedded real-time targets, you can choose to convert `if-elseif-else` decision logic to `switch-case` statements. This conversion can enhance readability of the code.

For example, when a MATLAB Function block contains a long list of conditions, the `switch-case` structure:

- Reduces the use of parentheses and braces
- Minimizes repetition in the generated code

How to Convert If-Elseif-Else Code to Switch-Case Statements

The following procedure describes how to convert generated code for the MATLAB Function block from `if-elseif-else` to `switch-case` statements.

Step	Task	Reference
1	Verify that your block follows the rules for conversion.	"Verifying the Contents of the Block" on page 55-108
2	Enable the conversion.	"Enabling the Conversion" on page 55-109
3	Generate code for your model.	"Generating Code for Your Model" on page 55-109

Rules for Conversion

For the conversion to occur, the following rules must hold. LHS and RHS refer to the left-hand side and right-hand side of a condition, respectively.

Construct	Rules to Follow
MATLAB Function block	Must have two or more *unique* conditions, in addition to a default. For more information, see “How the Conversion Handles Duplicate Conditions” on page 55-106.
Each condition	Must test equality only.
	Must use the same variable or expression for the LHS. **Note** You can reverse the LHS and RHS.
Each LHS	Must be a single variable or expression, not a compound statement.
	Cannot be a constant.
	Must have an integer or enumerated data type.
	Must not have side effects on simulation. For example, the LHS can read from but not write to global variables.
Each RHS	Must be a constant.
	Must have an integer or enumerated data type.

How the Conversion Handles Duplicate Conditions

If a MATLAB Function block has duplicate conditions, the conversion preserves only the first condition. The generated code discards other instances of duplicate conditions.

After removal of duplicates, two or more unique conditions must exist. Otherwise, a conversion does not occur and the generated code contains instances of duplicate conditions.

The following examples show how the conversion handles duplicate conditions.

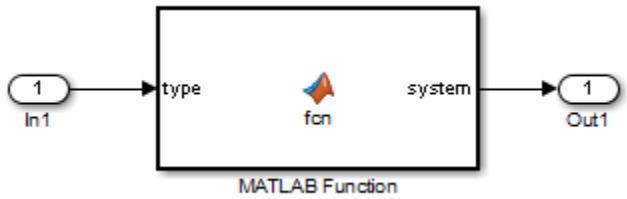
Example of Generated Code	Code After Conversion

`if (x == 1) {`
`  block1`
`} else if (x == 2) {`
`  block2`
`} else if (x == 1) {  // duplicate`
`  block3`
`} else if (x == 3) {`
`  block4`
`} else if (x == 1) {  // duplicate`
`  block5`
`} else {`
`  block6`
`}`
	`switch (x) {`
	`  case 1:`
	`  block1; break;`
	`  case 2:`
	`  block2; break;`
	`  case 3:`
	`  block4; break;`
	`  default:`
	`  block6; break;`
	`}`

`if (x == 1) {`
`  block1`
`} else if (x == 1) {  // duplicate`
`  block2`
`} else {`
`  block3`
`}`
	Does not change, because only one unique condition exists

Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements

Suppose that you have the following model with a MATLAB Function block. Assume that the output data type is `double` and the input data type is `Controller`, an enumerated type that you define.



The block contains the following code:

```
function system = fcn(type)
%#codegen

if (type == Controller.P)
    system = 0;
elseif (type == Controller.I)
    system = 1;
elseif (type == Controller.PD)
    system = 2;
elseif (type == Controller.PI)
    system = 3;
elseif (type == Controller.PID)
    system = 4;
else
    system = 10;
end
```

The enumerated type definition in `Controller.m` is:

```
classdef Controller < Simulink.IntEnumType
  enumeration
    P(0)
    I(1)
    PD(2)
    PI(3)
    PID(4)
    UNKNOWN(10)
  end
end
```

If you generate code for an embedded real-time target using default settings, you see something like this:

```
if (if_to_switch_eml_blocks_U.In1 == P) {
  /* '<S1>:1:4' */
  /* '<S1>:1:5' */
  if_to_switch_eml_blocks_Y.Out1 = 0.0;
} else if (if_to_switch_eml_blocks_U.In1 == I) {
  /* '<S1>:1:6' */
  /* '<S1>:1:7' */
```

```
  if_to_switch_eml_blocks_Y.Out1 = 1.0;
} else if (if_to_switch_eml_blocks_U.In1 == PD) {
  /* '<S1>:1:8' */
  /* '<S1>:1:9' */
  if_to_switch_eml_blocks_Y.Out1 = 2.0;
} else if (if_to_switch_eml_blocks_U.In1 == PI) {
  /* '<S1>:1:10' */
  /* '<S1>:1:11' */
  if_to_switch_eml_blocks_Y.Out1 = 3.0;
} else if (if_to_switch_eml_blocks_U.In1 == PID) {
  /* '<S1>:1:12' */
  /* '<S1>:1:13' */
  if_to_switch_eml_blocks_Y.Out1 = 4.0;
} else {
  /* '<S1>:1:15' */
  if_to_switch_eml_blocks_Y.Out1 = 10.0;
}
```

The LHS variable `if_to_switch_eml_blocks_U.In1` appears multiple times in the generated code.

Note By default, variables that appear in the block do not retain their names in the generated code. Modified identifiers prevent naming conflicts.

Traceability comments appear between each set of `/*` and `*/` markers. To learn more about traceability, see “Use Traceability in MATLAB Function Blocks” on page 85-34.

Verifying the Contents of the Block

Check that the block follows the rules in “Rules for Conversion” on page 55-105.

Construct	How the Construct Follows the Rules
MATLAB Function block	Five unique conditions exist, in addition to the default: • `(type == Controller.P)` • `(type == Controller.I)` • `(type == Controller.PD)` • `(type == Controller.PI)` • `(type == Controller.PID)`
Each condition	Each condition: • Tests equality • Uses the same input for the LHS

Construct	How the Construct Follows the Rules
Each LHS	Each LHS: • Contains a single variable • Is the input to the block and therefore not a constant • Is of enumerated type `Controller`, which you define in `Controller.m` on the MATLAB path • Does not impact simulation
Each RHS	Each RHS: • Is an enumerated value and therefore a constant • Is of enumerated type `Controller`

Enabling the Conversion

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** pane, select `ert.tlc` for the **System target file**.

   This step specifies an embedded real-time target for your model.
3. In the **Code Generation** > **Code Style** pane, select the **Convert if-elseif-else patterns to switch-case statements** check box.

   Tip This conversion works on a per-model basis. If you select this check box, the conversion applies to:

   - MATLAB Function blocks in a model
   - MATLAB functions in Stateflow charts of that model
   - Flow charts in Stateflow charts of that model

   For more information, see “Enhance Readability of Code for Flow Charts” on page 55-95.

Generating Code for Your Model

In the model window, press **Ctrl+B**.

The code for the MATLAB Function block uses `switch-case` statements instead of `if-elseif-else` code:

```
switch (if_to_switch_eml_blocks_U.In1) {
 case P:
  /* '<S1>:1:4' */
  /* '<S1>:1:5' */
  if_to_switch_eml_blocks_Y.Out1 = 0.0;
  break;

 case I:
  /* '<S1>:1:6' */
  /* '<S1>:1:7' */
  if_to_switch_eml_blocks_Y.Out1 = 1.0;
  break;
```

```
 case PD:
  /* '<S1>:1:8' */
  /* '<S1>:1:9' */
  if_to_switch_eml_blocks_Y.Out1 = 2.0;
  break;

 case PI:
  /* '<S1>:1:10' */
  /* '<S1>:1:11' */
  if_to_switch_eml_blocks_Y.Out1 = 3.0;
  break;

 case PID:
  /* '<S1>:1:12' */
  /* '<S1>:1:13' */
  if_to_switch_eml_blocks_Y.Out1 = 4.0;
  break;

 default:
  /* '<S1>:1:15' */
  if_to_switch_eml_blocks_Y.Out1 = 10.0;
  break;
}
```

The `switch-case` statements provide the following benefits to enhance readability:

- The code reduces the use of parentheses and braces.
- The LHS variable `if_to_switch_eml_blocks_U.In1` appears only once, minimizing repetition in the code.

See Also

More About

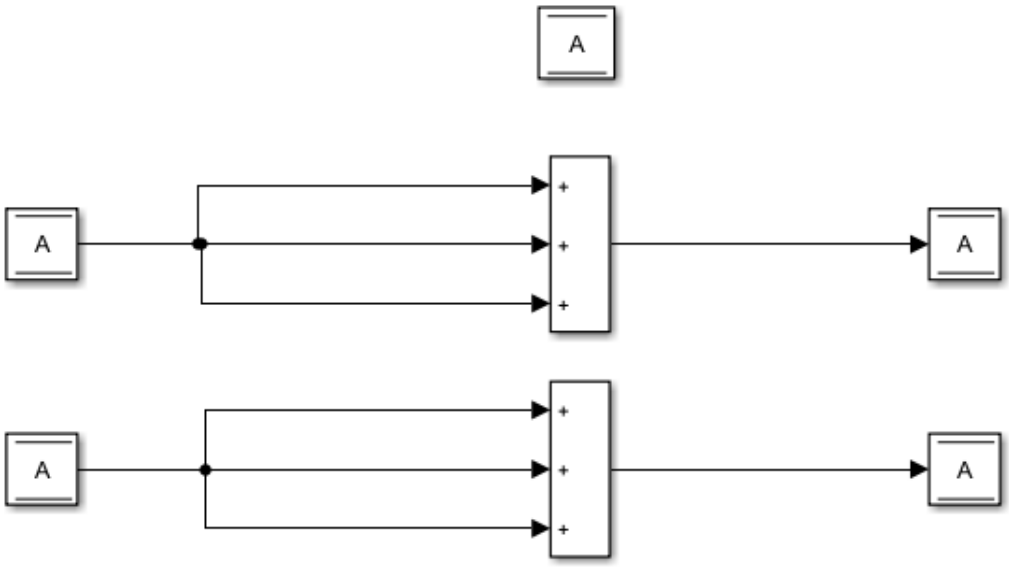
- “Use Traceability in MATLAB Function Blocks” on page 85-34
- “Code Generation for Enumerations”
- “Model Configuration Parameters: Code Style”

Improve Data Coherency in Generated Code

For models containing read and write operations for Data Store Memory blocks, you can generate code that contains a single variable to hold the value for each Data Store Read and Write operation. Generating one variable for each operation improves data access coherency.

Example Model

The `data_store_latching` model contains a single Data Store Memory block that is accessed by two different Data Store Memory Read and two Data Store Memory Write blocks. Two Sum blocks perform addition on the input generated by the Data Store Read blocks and output the data to the Data Store Write blocks .



Generate Code Without Parameter Enabled

Building the model with the **Implement each data store block as a unique access point** parameter turned off by default, generates this code:

```
void data_store_latching_step(void)
    {
      /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
       *  DataStoreRead: '<Root>/Data Store Read'
       *  Sum: '<Root>/Add'
       */
      data_store_latching_DW.A += data_store_latching_DW.A +
        data_store_latching_DW.A;

      /* DataStoreWrite: '<Root>/Data Store Write1' incorporates:
       *  DataStoreRead: '<Root>/Data Store Read1'
       *  Sum: '<Root>/Add1'
```

```
       */
      data_store_latching_DW.A += data_store_latching_DW.A +
        data_store_latching_DW.A;
    }
```

For both data read and write operations, the code contains the global variable `data_store_latching_DW.A`.

Generate Code With Parameter Enabled

In the Configuration Parameters dialog box, on the **Interface** pane, select the **Implement each data store block as a unique access point** parameter. Click **Apply**.

The generated code is now:

```
void data_store_latching_step(void)
    {
      real_T rtb_DataStoreRead;
      real_T rtb_DataStoreRead1;

      /* DataStoreRead: '<Root>/Data Store Read' */
      rtb_DataStoreRead = data_store_latching_DW.A;

      /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
       *  Sum: '<Root>/Add'
       */
      data_store_latching_DW.A = (rtb_DataStoreRead + rtb_DataStoreRead) +
        rtb_DataStoreRead;

      /* DataStoreRead: '<Root>/Data Store Read1' */
      rtb_DataStoreRead1 = data_store_latching_DW.A;

      /* DataStoreWrite: '<Root>/Data Store Write1' incorporates:
       *  Sum: '<Root>/Add1'
       */
      data_store_latching_DW.A = (rtb_DataStoreRead1 + rtb_DataStoreRead1) +
        rtb_DataStoreRead1;
    }
```

For each data store read operation, the code contains a variable. The variables are `rtb_DataStoreRead` and `rtb_DataStoreRead1`. These separate variables improve data access coherency.

See Also

More About

- “Data Stores in Generated Code”
- “Data Stores”
- Implement each data store block as a unique access point
- “Model Configuration Parameters: Code Style”

Code Replacement in Simulink Coder

- “What Is Code Replacement?” on page 56-2
- “Choose a Code Replacement Library” on page 56-6
- “Replace Code Generated from Simulink Models” on page 56-8

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from libraries that MathWorks provides and that you create and register by using the Embedded Coder product. The list of available libraries depends on:

- Installed support packages.
- System target file, language, language standard, and device vendor configuration.
- Whether you have created and registered libraries, using the Embedded Coder product.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Code Replacement Libraries

A code replacement library consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A code replacement table contains one or more code replacement entries, with each entry representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

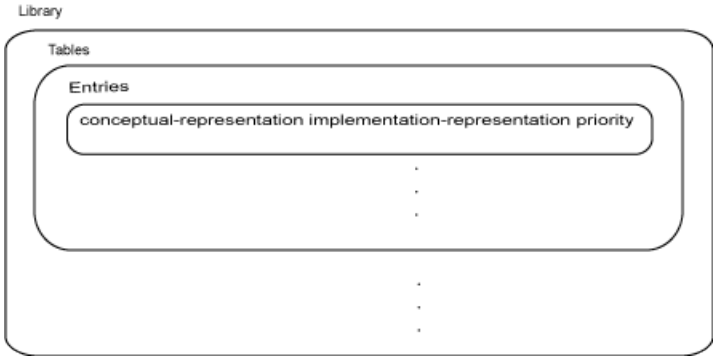


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a call site object with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

See Also

Related Examples

- “Choose a Code Replacement Library”
- “Replace Code Generated from Simulink Models”

Choose a Code Replacement Library

In this section...

"About Choosing a Code Replacement Library" on page 56-6
"Explore Available Code Replacement Libraries" on page 56-6
"Explore Code Replacement Library Contents" on page 56-6

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?" on page 57-2.
   - See "Explore Available Code Replacement Libraries" on page 56-6.
2. Explore the contents of the library. See "Explore Code Replacement Library Contents" on page 56-6.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

Select the Code replacement library to use for code generation from the **Configuration Parameters** > **Code Generation** > **Interface** pane . To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the **Code Replacement Viewer** to explore the content of a code replacement library.

1. At the command prompt, type `crviewer`.

   ```
   >> crviewer
   ```

   The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

   ```
   >> crviewer('GNU C99 extensions')
   ```
2. In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
3. In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
4. In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See **Code Replacement Viewer** for details on what the viewer displays.

See Also

Related Examples

- “What Is Code Replacement?”
- “Replace Code Generated from Simulink Models”

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as elimination of `math.h`, system header files, or calls to `memcpy` or `memset`, or use of BLAS.

Prepare for Code Replacement

1. Make sure that MATLAB®, Simulink®, Simulink Coder™, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can require Embedded Coder®.

To install MathWorks® products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver`.

2. Identify an existing Simulink model or create a model for which you want the code generator to replace code.

Choose a Code Replacement Library

By default, the code generator does not apply a code replacement library. You can choose from libraries that MathWorks® provides and that you create and register by using the Embedded Coder® product. The list of available libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you have created and registered libraries, using the Embedded Coder® product.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available. If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Configure Code Generator to Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the model. Do one of the following:

- In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, select a library for the Code replacement library parameter.

- Set the `CodeReplacementLibrary` parameter at the command line or programmatically.

2. Configure the code generator to produce code only (not build an executable program) so you can verify your code replacements before building an executable program. Do one of the following:

- In the Configuration Parameters dialog box, on the **Code Generation** pane, select Generate code only.
- Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information in Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

1. Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select Create code generation report. Consider having the report open automatically. Select Open report automatically.

2. Include the code replacement section in the report. Select Summarize which blocks triggered code replacements.

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, in the model window, press **Ctrl+B**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

More About

- “What Is Code Replacement?”
- “Choose a Code Replacement Library”
- “Code Generation Configuration”

Code Replacement for Simulink Models in Embedded Coder

- “What Is Code Replacement?” on page 57-2
- “Replace Code Generated from Simulink Models” on page 57-6

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from libraries that MathWorks provides and that you create and register by using the Embedded Coder product. The list of available libraries depends on:

- Installed support packages.
- System target file, language, language standard, and device vendor configuration.
- Whether you have created and registered libraries, using the Embedded Coder product.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Code Replacement Libraries

A code replacement library consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A code replacement table contains one or more code replacement entries, with each entry representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

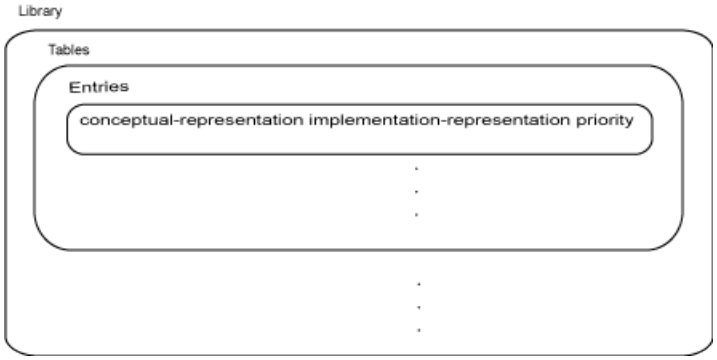


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a call site object with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

See Also

Related Examples

- “Choose a Code Replacement Library” on page 56-6
- “Replace Code Generated from Simulink Models” on page 57-6

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as elimination of `math.h`, system header files, or calls to `memcpy` or `memset`, or use of BLAS.

Prepare for Code Replacement

1. Make sure that MATLAB®, Simulink®, Simulink Coder™, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can require Embedded Coder®.

To install MathWorks® products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver`.

2. Identify an existing Simulink model or create a model for which you want the code generator to replace code.

Choose a Code Replacement Library

By default, the code generator does not apply a code replacement library. You can choose from libraries that MathWorks® provides and that you create and register by using the Embedded Coder® product. The list of available libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you have created and registered libraries, using the Embedded Coder® product.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available. If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Configure Code Generator to Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the model. Do one of the following:

- In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, select a library for the Code replacement library parameter.

- Set the `CodeReplacementLibrary` parameter at the command line or programmatically.

2. Configure the code generator to produce code only (not build an executable program) so you can verify your code replacements before building an executable program. Do one of the following:

- In the Configuration Parameters dialog box, on the **Code Generation** pane, select Generate code only.
- Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information in Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

1. Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select Create code generation report. Consider having the report open automatically. Select Open report automatically.

2. Include the code replacement section in the report. Select Summarize which blocks triggered code replacements.

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, in the model window, press **Ctrl+B**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

More About

- “What Is Code Replacement?”
- “Choose a Code Replacement Library”
- “Code Generation Configuration”

Deployment

External Code Integration in Simulink Coder

The code generator includes multiple approaches for integrating legacy or custom code with generated code. Legacy code is existing handwritten code or code for environments that you integrate with code produced by the code generator. Custom code is legacy code or other user-specified lines of code that you include in the code generator build process. Collectively, legacy and custom code are called external code.

You integrate external code by importing existing external code into code produced by the code generator, exporting generated code into an existing external code base, or you can do both. For example, you can import code by calling an external function, by using the Legacy Code Tool, or place external code at specific locations in generated code by including Custom Code blocks in a model. When you import external code, the resulting generated code interfaces with generated scheduling code.

You can export generated code as a plug-in function for use in an external development environment. When you export generated code, you intend to interface that code manually with a scheduling mechanism in your application run-time environment.

For guidance on choosing an approach based on your application, see “Choose an External Code Integration Workflow” .

test

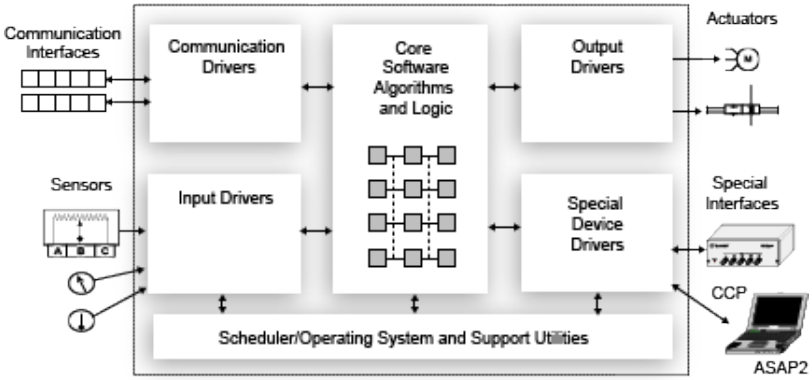
- “Choose an External Code Integration Workflow” on page 58-2
- “Configure Target Hardware Characteristics” on page 58-9
- “Check Code Generation Assumptions” on page 58-12
- “Call Reusable External Algorithm Code for Simulation and Code Generation” on page 58-15
- “Place External C/C++ Code in Generated Code” on page 58-23
- “Call External Device Drivers” on page 58-31
- “Apply Function and Operator Code Replacements” on page 58-32
- “Build Integrated Code Within the Simulink Environment” on page 58-33
- “Generate Component Source Code for Export to External Code Base” on page 58-40
- “Generate Shared Library for Export to External Code Base” on page 58-54
- “Interface to a Development Computer Simulator by Using a Shared Library” on page 58-59
- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
- “Exchange Data Between External Calling Code and Generated Code” on page 58-69
- “Generate Code That Matches Appearance of External Code” on page 58-72

Choose an External Code Integration Workflow

In this section...
“Choose a Software Execution Framework for Scheduling Code Execution” on page 58-3 “Evaluate Characteristics of External Code” on page 58-4 “Identify Integration Requirements” on page 58-6 “Choose a Workflow” on page 58-7

Software projects typically involve combining code from multiple sources. A typical system structure for a code generation application consists of a framework that combines code from multiple sources, including external code and code generated from Simulink models.

This figure shows an application that requires integration of existing driver code for hardware devices. The core software algorithms and logic can be a combination of code modules for external reusable algorithms integrated into the Simulink environment and code generated as part of an overall model design.



Completing these tasks helps you choose external code integration workflows and tooling that align with your project.

Task	Action	More Information
1	Partition your application, map algorithms to components, and identify integration points.	“Software Deployment Objectives and Deployment Types” on page 1-2 “Divide Software Problem” on page 2-2 ,“Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3
2	Determine whether you can rely on scheduling code that the code generator produces, or whether you must integrate generated code with scheduling mechanisms that are specific to your run-time environment.	“Choose a Software Execution Framework for Scheduling Code Execution” on page 58-3
3	Evaluate the characteristics of the external code that you are importing.	“Evaluate Characteristics of External Code” on page 58-4

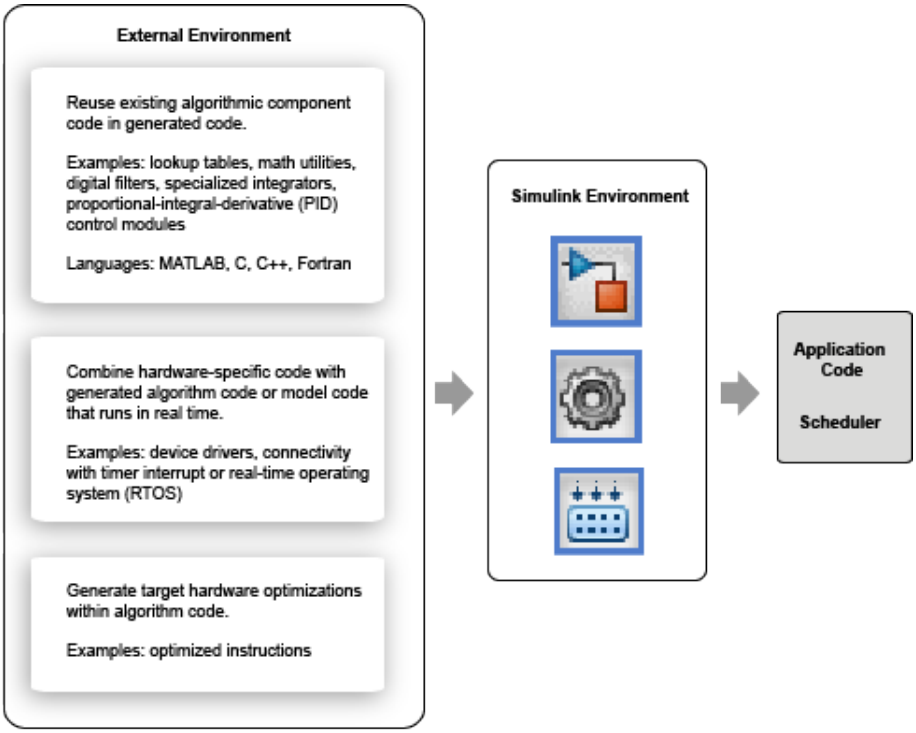
Task	Action	More Information
4	Identify integration requirements, which assists with choosing optimal tooling for your integration.	“Identify Integration Requirements” on page 58-6
5	Based on the results of tasks 1–4, choose a workflow.	“Choose a Workflow” on page 58-7

Choose a Software Execution Framework for Scheduling Code Execution

The code generator supports two types of software execution frameworks—single top model and multiple top-level. The first question to answer concerns which of the two frameworks meets the scheduling and other needs of your project. For example, you can import external code into a single, rate-based top model. You can export code from a single top model or multiple top-level models for integration with custom (external) scheduling mechanisms.

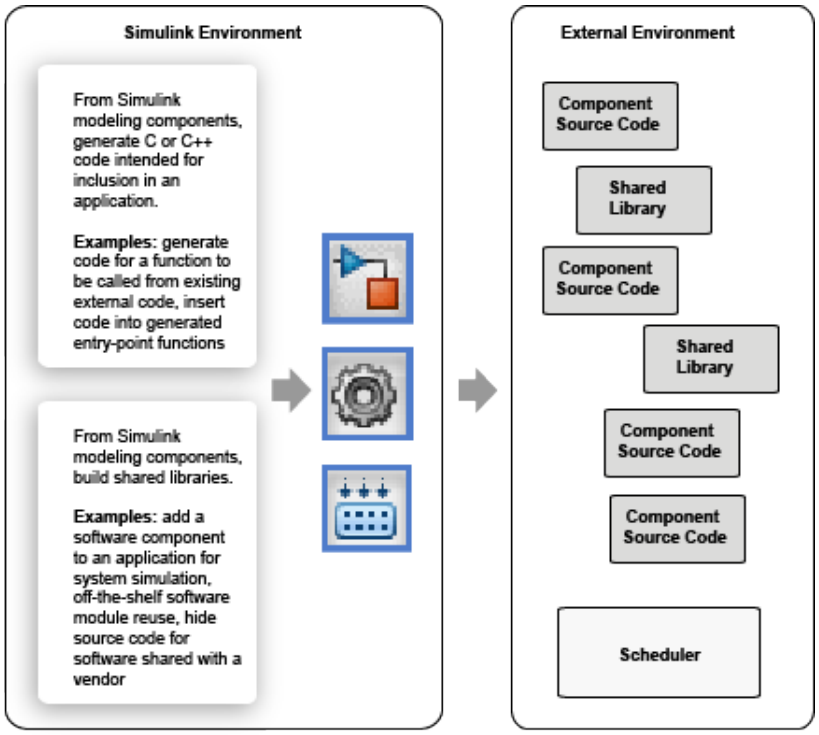
- Single top model

  Generate one set of application code files from external code and code that the Simulink C/C++ code generator produces. The generated code includes a scheduler. In this case, you *import* code into the Simulink code generation environment.



- Single top model or multiple top-level models

  Integrate C or C++ code that the code generator produces from model components with external application code and an external scheduler. You *export* generated code from the Simulink code generation environment.



Importing calls to external device driver code into a model and generating code for that model for export involves importing and exporting code.

Based on goals and requirements, external code integration is characterized in several ways, requiring different workflows and integration tooling:

- Import existing external code into generated code.
  - Call reusable external algorithm code for simulation and code generation.
  - Place external C/C++ code in generated code.
  - Call external device drivers.
  - Apply function and operator code replacements.
  - Interface with external timer interrupt or scheduler.
  - Generate replacement code for specific run-time environment.
- Export generated code for inclusion in external code base. Requires Embedded Coder .
  - Generate component source code for export.
  - Generate shared library for export.

  For more information on exporting generated code, see “Callable Function Integration”.

Next, see “Evaluate Characteristics of External Code” on page 58-4.

Evaluate Characteristics of External Code

Before choosing an external integration workflow, evaluate these characteristics of the external code. To interface with external code, generated C or C++ code handles one or more of the external code

characteristics. An understanding of these characteristics and your requirements for modeling, simulation, and code generation helps you choose the optimal workflow for your integration scenario. (See “Identify Integration Requirements” on page 58-6.)

Characteristic	What to Consider
Hardware dependency	Is the external code hardware-dependent? Utility functions, lookup tables, and filters are examples of hardware-independent code. Device drivers interact directly with hardware. They depend on characteristics of the hardware. For example, a device driver for an analog-to-digital converter initializes, reads data from, and writes data to hardware registers. Hardware differences and dependencies concern data type size, endianess, shift operations, compiler directives, and optimized function and operator support. Other code interfaces with device drivers by using an API and data mapped to specific memory addresses. Typically, simulation on a development computer is not possible. Reading from and writing to a register during simulation on a development computer produces unexpected and unwanted results.
Reusable	Is the external code a reusable software module? Examples include utility functions, lookup tables, filters, specialized integrators, and proportional-integral-derivative (PID) control modules.
Dependency on data persistence between function calls	Does the external code require persistent data? For example, a call to a first order filter function uses the output of the previous call to the function to calculate a new output value. You have the option of defining the data as global or using shared memory outside the context of the function.
Data typing and interface	How complex is the data that the external code uses? What does the data interface look like? It consists of arguments, a return value, global variables, and access functions. What data types does the code use? Are the types limited to basic ANSI C integers, floating-point types, arrays of integers or floating-point types, and pointers to these types? Does the interface include structures or pointers to structures?
Fixed-point code	Is the external code designed to run on integer-only processors? If yes, the code exchanges and uses data represented as integers only. Data can be associated with fixed-point scaling or offsets.
External resource dependencies	Does the external code use data, functions, or macros defined outside the scope of the code? For example, the function can use a standard ANSI function, a shared library, or predefined constants. In these cases, you must inform the compiler and linker of the paths and file names of the external resources.
External solver required	Are you using the external function for advanced development or rapid prototyping to describe a system with a continuous transfer function or a set of differential equations? If yes, the external code relies on an external solver.

Next, see “Identify Integration Requirements” on page 58-6.

Identify Integration Requirements

Before choosing an external integration workflow, review these integration requirements. An understanding of these requirements and the characteristics of your external code helps you choose the optimal workflow for your integration scenario. (See “Evaluate Characteristics of External Code” on page 58-4.)

Requirement	What to Consider
Effort	What level of effort is planned for the integration project—low, medium, or high?
Learning effort	What is the programming experience of assigned project resources? How much experience do assigned resources have with Simulink and MathWorks C/C++ code generation products?
Simulation and code generation behaviors	Do you want to take advantage of Model-Based Design? To take full advantage of Model-Based Design, convert code to modeling elements, which you can then use in the Simulink and Stateflow simulation environment. Then, simulate and generate code for the integrated component. Use software-in-the-loop (SIL) or processor-in-the-loop (PIL) testing to verify whether algorithm behavior is the same in both environments.
Data interface and typing	• Does your model or generated code need to exchange data with the external function? If so, map inputs, outputs, and parameters to the external function interface. Typical function interfaces involve function arguments and return values, global variables, and access functions, such as `getRPM`. • Do you want to represent arrays, structures, or enumerated types? In the Simulink environment, you can represent these types as vectors, buses, and `IntEnum`, respectively. • Is fixed-point support required? If you use the Simulink fixed-point interface, you can scale and specify offsets. • Does the external code use company-specific data types? If yes and you have Embedded Coder software, create alias types to represent those external types. The code generator uses the alias types in the code that it produces. For example, once defined, you can specify an alias type in a function prototype, for a temporary variable, or for block output. • Does the code exchange data with shared memory? If yes, define and use memory sections.
Direct function call	Do you want to call C external code directly from a model? You can choose from mechanisms, such as the C Function block, the C Caller block, Legacy Code Tool, Stateflow external code interface and chart action language, and the MATLAB Function block.
Insertion of external code into generated code	Do you want to control the placement of external code within generated code? Do you want to insert code into generated entry-point functions? You can place code within generated code by using model configuration parameters or custom code blocks.

Requirement	What to Consider
Code generation optimization support	Do you want to optimize the code that the code generator produces? If so, you can configure the model for the code generator to optimize the code it produces based on application objectives, such as execution, ROM, and RAM efficiency. You also have the option of using code replacement libraries.
Files required	Do you want to minimize the number of files that you maintain? Some external code integration tools require that you maintain separate files for defining simulation and code generation.

Next, see “Choose a Workflow” on page 58-7.

Choose a Workflow

To choose a workflow for integrating external code into the Simulink environment, use the following table with links to more information.

Goal	Action	More Information
Call hardware-independent reusable external algorithmic code from generated code. For example: utility functions, lookup tables, and digital filters.	Choose an approach for integrating external code into the Simulink environment for code generation based on your programming language and other integration requirements.	“Call Reusable External Algorithm Code for Simulation and Code Generation” on page 58-15
Call hardware-dependent external reusable external algorithmic code from generated code. For example: device drivers.	Call external driver code from the Simulink environment.	“Call External Device Drivers” on page 58-31
Place external C/C++ code in generated code.	Augment functions with external code using model configuration parameters or custom code blocks.	“Place External C/C++ Code in Generated Code” on page 58-23
Change the code that the code generator produces for functions and operators to meet application code requirements.	Use code replacements to specify application-specific implementations of functions and operators.	“Apply Function and Operator Code Replacements” on page 58-32

See Also

More About

- “Call Reusable External Algorithm Code for Simulation and Code Generation” on page 58-15
- “Place External C/C++ Code in Generated Code” on page 58-23
- “Call External Device Drivers” on page 58-31
- “Deploy Generated Component Software to Application Target Platforms” on page 71-34

- “Code Replacement”
- “Generate Component Source Code for Export to External Code Base” on page 58-40
- “Generate Shared Library for Export to External Code Base” on page 58-54

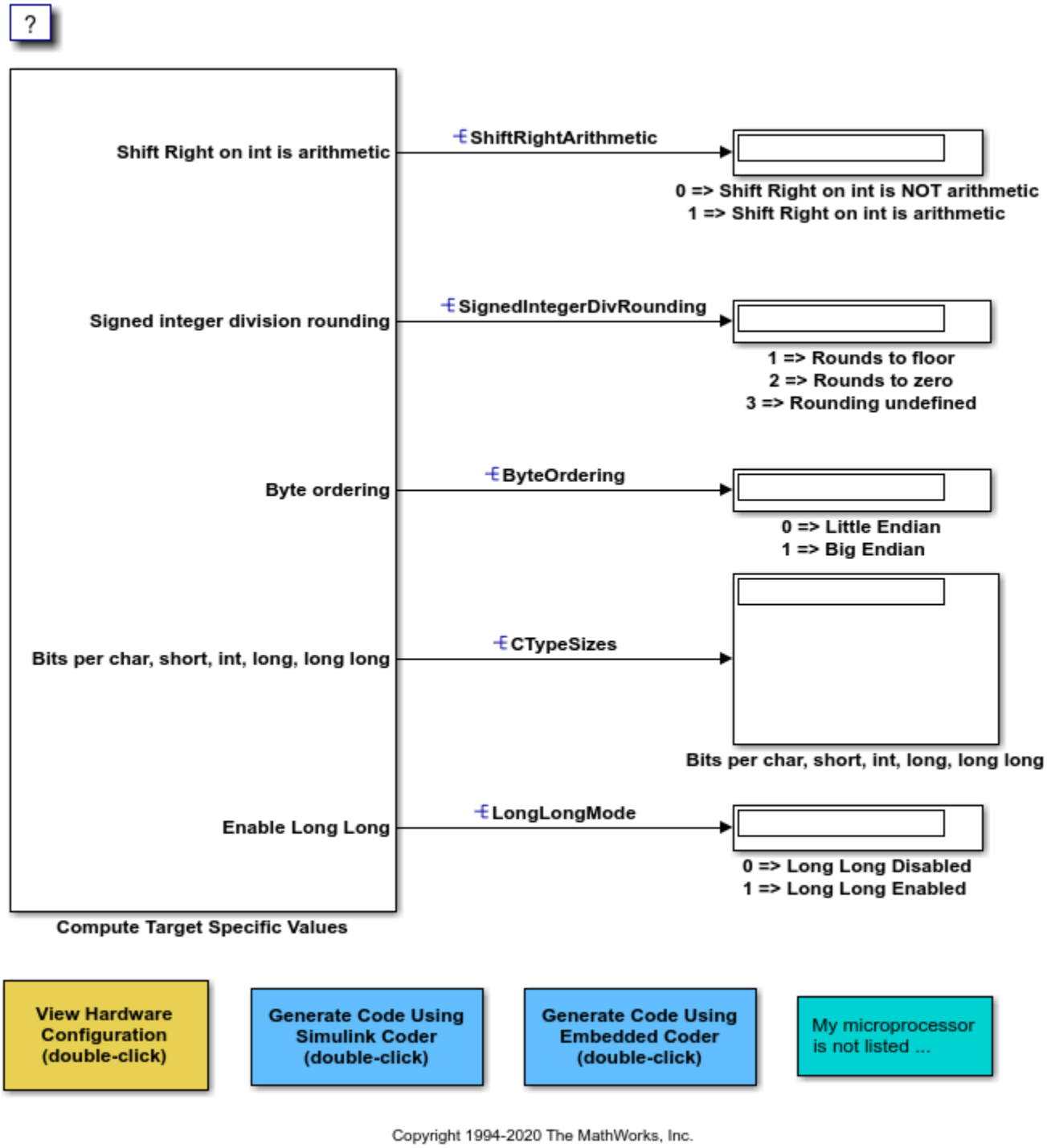
Configure Target Hardware Characteristics

Simulink® allows you to configure the hardware implementation characteristics of your target system. Failure to do so can result in code that is incorrect or inefficient.

Open Example Model

Open the example model `rtwdemo_targetsettings`.

```
open_system('rtwdemo_targetsettings');
```



Copyright 1994-2020 The MathWorks, Inc.

Open the **Hardware Implementation** pane of the Configuration Parameters dialog box by double-clicking the yellow button in the model.

Hardware Board Configuration

By default, the model is configured for the code generator to determine the hardware board configuration based on the specified system target file. The default system target file setting is `grt.tlc`.

If you have installed support for a hardware board, that board should appear in the **Hardware board** menu. If you select it, the dialog box displays parameters that are relevant to your hardware board.

My Hardware Board Is Not Listed

If you are using a hardware board that is not listed, add it to the menu by installing the corresponding target support package. Start the Support Package Installer by setting **Hardware board** to `Get hardware Support Packages`, or by entering `supportPackageInstaller` in the MATLAB® Command Window.

After installing support for a hardware board, reopen the Configuration Parameters dialog box and select the hardware board.

Cross-Development Systems That Include Source Level Debuggers

If you use a cross-development system that includes a source level debugger:

1. Configure the model for your hardware board.

2. Generate code and build an executable using your target development system. For details about retrieving file and path information, see the documentation on the Simulink Coder® Build Information API.

3. Download the executable to the target hardware and set breakpoints at the model output code for each outport of the model.

4. Run a debugging session and observe the behavior of the code as it executes on the target hardware. As you step through the code, it computes the values for target hardware device characteristics.

If your target development system does not support source level debugging, generate code and add `printf` statements or other code that gets the results.

5. On the **Hardware Implementation** pane of the Configuration Parameters dialog box, set **Device vendor** to `Custom processor`, click the **Device details** arrow, and adjust the microprocessor device settings manually.

Check Code Generation Assumptions

The code generator implements assumptions that depend, for example, on the hardware implementation settings for your model. It is important to check that the assumptions are valid for your target hardware. Use the `buildStandaloneCoderAssumptions` function to create an application that performs assumption checks on your target hardware.

When you configure your model, try to specify hardware implementation and build configuration settings that match your target hardware. Then, to check code generator assumptions for your target hardware, use this workflow:

1. With `GenerateReport` set to `'on'`, build (**Ctrl+B**) your model.
2. To view the list of code generator assumptions that you can check, open the code generation report and click the **Coder Assumptions** link.

   For more information, see “Coder Assumptions List” on page 58-13.
3. Run `buildStandaloneCoderAssumptions`, which uses the generated code in the build folder to create an application that runs code generation assumption checks.
4. Download the application onto the target hardware or target environment, and then run the application.
5. While the application runs, use a debug tool to view check results that are in a data structure.

Check Code Generator Assumptions for Development Computer

If the target hardware is, for example, your Windows development computer, you can use Microsoft Visual Studio® to run and debug the application:

1. To open a model, in the Command Window, enter:

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILTopModel.slx')
   ```
2. In the Model Configuration Parameters dialog box, in the **Hardware Implementation** pane, specify settings to match the target hardware. For example, for a 64-bit Windows computer specify these parameter settings:

   - **Device vendor** –– `Intel`
   - **Device type** –– `x86-64 (Windows 64)`
3. Clear model configuration parameter **Enable portable word sizes**.
4. Build the model (**Ctrl+B**).
5. To view the list of code generator assumptions, open the code generation report and click the **Coder Assumptions** link.
6. From the Command Window, in the working folder, run:

   ```
   buildStandaloneCoderAssumptions('SILTopModel_ert_rtw')
   ```

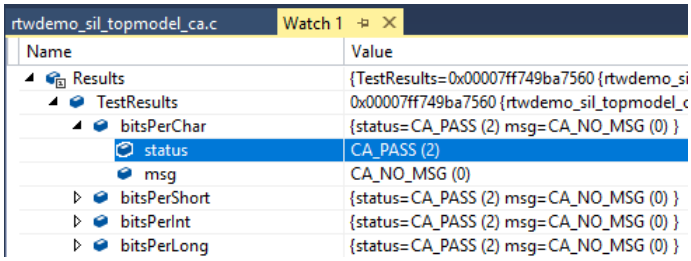
   In the build folder, the function creates the `coderassumptions\standalone` subfolder, which contains the target application, `SILTopModel_ca`.
7. Open Microsoft Visual Studio and select **File** > **Open** > **Project/Solution**

8. Using the Open Project dialog box, navigate to the `coderassumptions\standalone` subfolder and select `SILTopModel_ca`. Then, click **Open**.
9. Select **File** > **Open** > **File**. Using the Open File dialog box, select `coderassumptions\standalone\SILTopModel_ca.c`
10. At the `return` statement, insert a break point.
11. Select **Debug** > **Start Debugging**.
12. To verify code generator assumptions, use a Watch window to inspect the `Results` data structure:
    a. In the `SILTopModel_ca.c` code, right-click `Results`.
    b. From the context menu, select **Add Watch**.

The variables in the data structure contain:

- Check outcomes (`TestResults`)
- Target hardware truths (`ActualValues`)
- Code generator assumptions (`ExpectedValues`)

For example, if the code generator assumption for `bitsPerChar` is accurate, you see the `CA_PASS` value in the `status` variable.



Coder Assumptions List

The contents section of the code generation report has a link to the Coder Assumptions page. The page provides a list of:

- Code generation assumptions that you can check
- Expected results for the assumption checks

This table describes the labels that you see in the list.

Category	Label	Assumption You Can Check
C/C++ Language Configuration for Target Hardware or Development Computer	`BitPerChar`	Number of bits per `char` (`ProdBitPerChar`)
	`BitPerShort`	Number of bits per `short` (`ProdBitPerShort`)
	`BitPerInt`	Number of bits per `int` (`ProdBitPerInt`)

Category	Label	Assumption You Can Check
	`BitPerLong`	Number of bits per `long` (`ProdBitPerLong`)
	`BitPerFloat`	Size of `float` (Only if `PurelyIntegerCode` is `'off'`)
	`BitPerDouble`	Size of `double` (Only if `PurelyIntegerCode` is ‘`off`’)
	`BitPerPointer`	Number of bits per `pointer` (`ProdBitPerPointer`)
	`BitPerSizeT`	Number of bits per `size_t` (`ProdBitPerSizeT`)
	`BitPerPtrDiffT`	Number of bits per `ptrdiff_t` (`ProdBitPerPtrDiffT`)
	`Endianess`	Byte ordering (`ProdEndianess`)
	`Shift right for signed integer is arithmetic shift`	Sign bit behavior (`ProdShiftRightIntArith`)
	`Signed integer division rounds to`	Rounding behavior for integer division (`ProdIntDivRoundTo`)
C/C++ Language Standard	`Initial value of a global integer variable and dynamically allocated memory is zero`	Zero initialization of memory (`ZeroExternalMemoryAtStartup`, `ZeroInternalMemoryAtStartup`, and `GenerateAllocFcn`)
Floating-Point Numbers	`Flush-to-zero computed subnormal values (FTZ)`	Flush-to-zero subnormal number processing
	`Flush-to-zero incoming subnormal values (DAZ)`	Denormals-are-zero subnormal number processing

See Also

`buildStandaloneCoderAssumptions`

More About

- “Verification of Code Generation Assumptions” on page 88-71

Call Reusable External Algorithm Code for Simulation and Code Generation

Code reuse offers business and technological advantages. From a business perspective, code reuse saves time and resources. From a technological perspective, code reuse promotes consistency and reduces memory requirements. Other considerations include:

- Modularizing an application
- Reusing an optimized algorithm
- Interfacing with a predefined dataset
- Developing application variants

Examples of reusable hardware-independent algorithmic code to consider importing into the Simulink environment for simulation and code generation include:

- Utility functions
- Lookup tables
- Digital filters
- Specialized integrators
- Proportional-integral-derivative (PID) control modules

Workflow

To call reusable external algorithm code for simulation and code generation, iterate through the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	“Choose an External Code Integration Workflow” on page 58-2
2	Based on the programming language of the external code, choose an integration approach to add the external code to a Simulink model.	“Choose an Integration Approach” on page 58-16
3	Verify algorithm behavior and performance by simulating the model.	“Simulation”
4	Define the representation of model data for code generation.	“Code Interface Configuration” and “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
5	Configure the model for code generation.	“Model Configuration Set Customization”
6	Generate code and a code generation report.	“Code Generation”

Task	Action	More Information
7	Review the generated code interface and static code metrics.	“Analyze Generated Data Code Interface” on page 54-13, “Analyze Generated Service Code Interface Report” on page 54-21, and “Static Code Metrics” on page 54-26
8	Build an executable program from the model.	“Build Integrated Code Within the Simulink Environment” on page 58-33
9	Verify that executable program behaves as expected.	“Verification and Testing”“Verification, Testing, and Certification”

Choose an Integration Approach

Several approaches are available for integrating reusable algorithmic code into the Simulink environment for code generation. Some approaches integrate external code directly. Other approaches convert the external code to Simulink or Stateflow modeling elements for simulation, and later for code generation from the modeled design. The integration approach that you choose depends on:

- Programming language of the external code — MATLAB, C, C++, or Fortran
- Your programming language experience and preference
- Performance requirements
- Whether the algorithm must model continuous time dynamics or you are integrating the algorithm into an application that uses discrete and continuous time
- Whether you want to take advantage of Model-Based Design
- Level of control required over the code that the code generator produces

To choose an approach for a reusable algorithm, see the subsection that matches the programming language of your external algorithm code.

- “Integration Approaches for External MATLAB Code” on page 58-16
- “Integration Approaches for External C or C++ Code” on page 58-18
- “Integration Approaches for External Fortran Code” on page 58-21

Integration Approaches for External MATLAB Code

Multiple approaches are available for integrating external MATLAB code into the Simulink environment. The following table helps you choose the best integration approach for your application based on integration requirements.

Condition or Requirement	Action	More Information
• External code complies with the MATLAB code generation subset • You want to call MATLAB code from a Simulink model	Add a MATLAB Function block to the model. Embed the MATLAB code in that block.	• “Integrate C Code by Using the MATLAB Function Block” • MATLAB Function

Condition or Requirement	Action	More Information
• External code complies with the MATLAB code generation subset • You want to call MATLAB code from a Simulink model • Your algorithm includes iterative computations that process large streams of data	Add a MATLAB System block to the model. Embed the MATLAB code in that block as a System object™.	• “Integrate C Code by Using the MATLAB Function Block” • MATLAB System
• External code complies with the MATLAB code generation subset • You want to call MATLAB code from a Simulink model • Your algorithm includes design logic that is based on state machines and flow charts	Add a Stateflow chart to the model. Call the external code from the chart, using MATLAB as the action language.	• “Chart Programming Basics” (Stateflow) • “Insert External Code into Stateflow Charts” on page 58-21
You want to use the `parfor` function for parallel computing or interface data types that are available to MATLAB Coder, Simulink Coder, and Embedded Coder. To use `parfor`, Parallel Computing Toolbox must be installed.	Use software to generate C code. Then, call that generated code as external C code.	• “Generate C Code by Using the MATLAB Coder App” • “Get Started with MATLAB Coder”
You have C or C++ programming experience and the external MATLAB code is compact and primarily uses C or C++ constructs.	Manually convert the MATLAB code to C or C++ code. Choose an integration approach for C or C++ code.	“Integration Approaches for External C or C++ Code” on page 58-18
Sections of the external MATLAB code map to built-in blocks.	Develop the algorithm in the context of a model, using the applicable built-in blocks.	• “Interactive Model Editing” and “Project Management” • “Use Products and Blocks Supported for Code Generation” on page 25-2
The algorithm must model continuous state dynamics.	Write a MATLAB S-function and, for generating code, a corresponding TLC file for the algorithm. Add the S-function to your model.	• “Create and Configure MATLAB S-Functions” • “Target Language Compiler”

To embed external MATLAB code in a MATLAB Function block or generate C or C++ code from MATLAB code with the MATLAB Coder software, the MATLAB code must use functions and classes supported for C/C++ code generation.

- “Functions and Objects Supported for C/C++ Code Generation”

Integration Approaches for External C or C++ Code

Multiple approaches are available for integrating external C or C++ code into the Simulink environment. The following table helps you choose the best integration approach for your application based on integration requirements.

Condition or Requirement	Action	More Information
You want to integrate external C code with generated C++ code or conversely	Match the language choice for the generated code by modifying the language of the external code.	“Modify Programming Language of External Code to Match Generated Code” on page 58-20
• You have existing C code that models custom functionality. You want to integrate the code into Simulink blocks. • Your functions do not read or write global/static variables. • You want to easily integrate with Simulink Coverage, Simulink Test, and Simulink Design Verifier	For simple algorithms written in C, consider using the Simulink C Caller block.	“Integrate C Code Using C Caller Blocks”
• You have existing C code that models custom functionality. You want to call multiple C functions from a Simulink block, or call C functions conditionally. • You want to include C code to preprocess and postprocess data before and after calling C functions. • You want to specify different code for simulation and code generation.	Add a C Function block to your model.	“Integrate C Code by Using the MATLAB Function Block”

Condition or Requirement	Action	More Information
• The algorithm must model discrete and continuous state dynamics for simulation and rapid prototyping. • The external code requires a fixed-point interface. • Programming experience is limited. You want ease of use with basic flexibility for controlling what code the code generator produces for rapid prototyping.	Generate S-function and TLC files by using the S-Function Builder. If necessary, refine the generated code manually to meet application requirements. (If you change the generated code, you lose the changes if you regenerate the S-function and TLC files.)	• "Use a Bus with S-Function Builder to Create an S-Function"
• You want to simulate and generate external code for a discrete time application. Optimizing generated code is essential. • You want ease of use with moderate flexibility for controlling what code the code generator produces. • You have C or C++ programming experience, but you prefer to generate the files for adding the code to a model.	Generate S-function and TLC files by using the Legacy Code Tool. If necessary, refine the generated code manually to meet application requirements. (If you change the generated code, you lose the changes if you regenerate the S-function and TLC files.)	• "Integrate C Functions Using Legacy Code Tool" • "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 13-6 • "Integrate C Code Using C Caller Blocks"
• The application requires more entry-point functions than the code generator typically produces—for example, more than `model_step`, `model_initialize`, and `model_terminate`. You want maximum flexibility for controlling what code the code generator produces.	Manually write an S-function and TLC file.	• "C/C++ S-Function Basics" • "S-Functions and Code Generation" on page 13-2 • "C S-Function Examples" and "C++ S-Function Examples"

Condition or Requirement	Action	More Information
Your algorithm includes design logic that is based on state machines and flow charts. Or, a function that you want to integrate must exchange data with a model by using global variables. The function defines the global variables and uses them to write output rather than returning a value (return) or writing output to an argument.	Add a Stateflow chart to the model. Call the external code from the chart, using C as the action language. In the chart, write code that calls the external function and reads from and writes to the global variables. To perform calculations with output of the external code, the model must read from the global variable during execution.	"Insert External Code into Stateflow Charts" on page 58-21
You want to include external C or C++ code in a Stateflow chart for simulation and code generation.	Configure the model that contains the chart to apply the external C or C++ code.	• "Custom Code" (Stateflow) • "Call C Library Functions in C Charts" (Stateflow) • "Insert External Code into Stateflow Charts" on page 58-21
You quickly want to embed a call to external C or C++ code in a model. Performance is not an issue.	Call the C or C++ code with the `coder.ceval` function from within a MATLAB Function block.	• `coder.ceval` function • "Integrate C Code by Using the MATLAB Function Block" • MATLAB Function block

Modify Programming Language of External Code to Match Generated Code

To integrate external C code with generated C++ code or conversely, modify the language of the external code to match the programming language choice for the generated code. Options for making the programming language match include:

- Writing or rewriting the external code in the language choice for the generated code.
- If you are generating C++ code and the external code is C code, for each C function, create a header file that prototypes the function. Use this format:

  ```
  #ifdef __cplusplus
  extern "C" {
  #endif
  int my_c_function_wrapper();
  #ifdef __cplusplus
  }
  #endif
  ```

  The prototype serves as a function wrapper. If your compiler supports C++ code, the value `__cplusplus` is defined. The linkage specification `extern "C"` specifies C linkage without name mangling.
- If you are generating C code and the external code is C++ code, include an `extern "C"` linkage specification in each `.cpp` file. For example, the following example shows C++ code in the file `my_func.cpp`:

  ```
  extern "C" {

  int my_cpp_function()
  {
  ```

```
  ...
}
}
```

Integration Approaches for External Fortran Code

To integrate external Fortran code:

1. Write an S-function and corresponding TLC file.
2. Add the S-function to your model.
3. If necessary, add support files and control model code generation and builds within the Simulink environment.

See “C/C++ S-Function Basics”, “Implement Algorithms Using Fortran Code”, “S-Functions and Code Generation” on page 13-2, and “Fortran S-Function Examples”.

Insert External Code into Stateflow Charts

- “Integrate External Code for Library Charts” on page 58-21
- “Integrate External Code for All Charts” on page 58-21

Integrate External Code for Library Charts

To integrate external code that applies only to Stateflow library charts for code generation, for each library model that contributes a chart to your main model, complete these steps. Then, generate code.

1. In the Stateflow Editor, open the Model Configuration Parameters dialog box. Select parameter **Use local custom code settings (do not inherit from main model)**,

   The library model retains its own custom code settings during code generation.
2. Specify your custom code in the subpanes.

   Follow the guidelines in “Specify Relative Paths to Your Custom Code” (Stateflow).

   If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.
3. Click **OK**.

After completing these steps for each library model, generate code.

Integrate External Code for All Charts

To integrate external code that applies to all charts for code generation:

1. Specify custom code options for code generation of your main model.

   a. In the Model Configuration Parameters dialog box, select **Code Generation** > **Custom Code**.

   b. In the custom code text fields, specify your custom code.

      Follow the guidelines in “Specify Relative Paths to Your Custom Code” (Stateflow).

If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

2. Configure code generation for each library model that contributes a chart to your main model. In the Stateflow Editor, open the Model Configuration Parameters dialog box. Clear parameter **Use local custom code settings (do not inherit from main model)**. The library charts inherit the custom code settings of your main model. Click **OK**.
3. Generate code.

See Also

More About

- "Integrate C Functions Using Legacy Code Tool"
- "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 13-6
- "Custom Code" (Stateflow)
- "Integrate C Code by Using the MATLAB Function Block"
- "C/C++ S-Function Basics"
- "S-Functions and Code Generation" on page 13-2

Place External C/C++ Code in Generated Code

In this section...
“Workflow” on page 58-23 “Choose an Integration Approach” on page 58-24 “Integrate External Code by Using Custom Code Blocks” on page 58-25 “Integrate External Code by Using Model Configuration Parameters” on page 58-27 “Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters” on page 58-28

You can customize code that the code generator produces for a model by specifying external code with custom code blocks or model configuration parameters.

- Place code at the start and end of the generated code for the root model.
- Place declaration, body, and exit code in generated function code for blocks in the root model or nonvirtual subsystems.

The functions that you can augment with external code depends on the functions that the code generator produces for blocks that are in the model. For example, if a model or atomic subsystem includes blocks that have states, you can specify code for a disable function. Likewise, if you need the code for a block to save data, free memory, or reset target hardware, specify code for a terminate function. For more information, see “Block Target File Methods”.

Workflow

To place external C or C++ code at specific locations in code that the code generator produces for root models and subsystems, iterate through the tasks listed in this table.

Task	Action	More Information
1	If you want to integrate external C code with generated C++ code or conversely, modify the language of the external code to match the language choice for the generated code.	“Modify Programming Language of External Code to Match Generated Code” on page 58-20
2	Review your assessment of external code characteristics and integration requirements.	“Choose an External Code Integration Workflow” on page 58-2
3	If necessary, rewrite code in C or C++.	
4	Choose an integration approach to add the external code to a Simulink model.	“Choose an Integration Approach” on page 58-24
5	Define the representation of model data for code generation.	“Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
6	Configure the model for code generation.	“Model Configuration Set Customization”

Task	Action	More Information
7	Generate code and a code generation report.	“Code Generation”
8	Review the generated code interface and static code metrics.	“Analyze Generated Data Code Interface” on page 54-13, “Analyze Generated Service Code Interface Report” on page 54-21, and “Static Code Metrics” on page 54-26
9	Build an executable program from the model.	“Build Integrated Code Within the Simulink Environment” on page 58-33
10	Verify that executable program performs as expected.	“Verification, Testing, and Certification”

Choose an Integration Approach

Within the Simulink modeling environment, two approaches are available for placing external C or C ++ code into sections of code that the code generator produces:

- Add blocks from the Custom Code library to a root model or atomic subsystem.
- Set model configuration parameters on the **Code Generation** > **Custom Code** pane.

The following table compares the two approaches. Choose the approach that aligns best with your integration requirements. For more information about how to apply each approach, see “Integrate External Code by Using Custom Code Blocks” on page 58-25 and “Integrate External Code by Using Model Configuration Parameters” on page 58-27.

Requirement	Blocks	Model Configuration Parameters
Include a representation of your external code in the modeling canvas	✓	
Place code in functions generated for root models	✓	✓
Place code in functions generated for atomic subsystems	✓	
Save code placement in a model configuration set		✓
Place code at the top and bottom of the header and source files generated for a model	✓	✓
Place code within declaration, execution, and exit sections of the `SystemInitialize` and`SystemTerminate` functions that the code generator creates	✓	✓
Place code within declaration, execution, and exit sections of the `SystemStart`, `SystemEnable`, `SystemDisable`, `SystemOutputs`, `SystemUpdate`, or`SystemDerivatives` functions that the code generator creates	✓	
Add preprocessor macro definitions to generated code		✓

Requirement	Blocks	Model Configuration Parameters
Use the custom code settings that are specified for the simulation target		✓
Configure a library model to use custom code settings of the parent model to which the library is linked		✓

Integrate External Code by Using Custom Code Blocks

- “Custom Code Block Library” on page 58-25
- “Add Custom Code Blocks to the Modeling Canvas” on page 58-26
- “Add External Code to Generated Start Function” on page 58-26

Custom Code Block Library

The Custom Code block library contains blocks that you can use to place external C or C++ code into specific locations and functions within code that the code generator produces. The library consists of 10 blocks that add your code to the model header (*model*.h) and source (*model*.c or *model*.cpp) files that the code generator produces.

The Model Header and Model Source blocks add external code at the top and bottom of header and source files that the code generator produces for a root model. These blocks display two text fields into which you can type or paste code. One field specifies code that you want to place at the top of the generated header or source file. The second field specifies code that you want to place at the bottom of the file.

The remaining blocks add external code to functions that the code generator produces for the root model or atomic subsystem that contains the block. The blocks display text fields into which you can type or paste code that customizes functions that the code generator produces. The text fields correspond to the declaration, execution, and exit sections of code for a given function.

To Customize Code That	Use This Block
Computes continuous states	System Derivatives
Disables state	System Disable
Enables state	System Enable
Resets state	System Initialize
Produces output	System Outputs
Executes once	System Start
Saves data, free memory, reset target hardware	System Terminate
Requires updates at each major time step	System Update

The block and its location within a model determines where the code generator places the external code. For example, if the System Outputs block is at the root model level, the code generator places the code in the model `Outputs` function. If the block resides in a triggered or enabled subsystem, the code generator places the code in the subsystem `Outputs` function.

If the code generator does not need to generate a function that corresponds to a Custom Code block that you include in a model, the code generator does one of the following:

- Omits the external code that you specify in the Custom Code block.
- Returns an error, indicating that the model does not include a relevant block. In this case, remove the Custom Code block from the model.

For more information, see “Block Target File Methods”.

Note To avoid a potential mismatch between simulation and code generation results, do not use custom code blocks to read from and write to global Simulink data (signals, states, and block parameters). Instead, use the proper modeling pattern (for example, Data Store Read, Data Store Write, State Reader, and State Writer blocks.)

Add Custom Code Blocks to the Modeling Canvas

To add the Custom Code library blocks to a model:

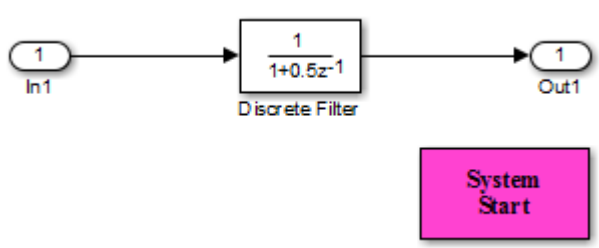
1. In the Simulink Library Browser, expand the `Simulink Coder` node then select the Custom Code block library.
2. Drag the blocks that you want into your model or subsystem. Drag Model Header and Model Source blocks into root models only. Drag function-based Custom Code blocks into root models or atomic subsystems.

You can use models that contain Custom Code blocks as referenced models. The code generator ignores the blocks when producing code for a simulation target. When producing code for a code generation target, the code generator includes and compiles the custom code.

Add External Code to Generated Start Function

This example shows how to use the System Start block to place external C code in the code that the code generator produces for a model that includes a discrete filter.

1. Create the following model.



2. Configure the model for code generation.
3. Double-click the System Start block.
4. In the block parameters dialog box, in the **System Start Function Declaration Code** field, enter this code:

   ```
   unsigned int *ptr = 0xFFEE;
   ```
5. In the **System Start Function Execution Code** field, enter this code:

```
/* Initialize hardware */
*ptr = 0;
```

6. Click **OK**.
7. Generate code and a code generation report.
8. View the generated *model*.c file. Search for the string `start function`. You should find the following code, which includes the external code that you entered in steps 4 and 5.

```
{
  {
    /* user code (Start function Header) */
    /* System '<Root>' */
    unsigned int *ptr = 0xFFEE;

    /* user code (Start function Body) */
    /* System '<Root>' */
    /* Initialize hardware */
    *ptr = 0;
  }
}
```

For another example, see “Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters” on page 58-28.

Integrate External Code by Using Model Configuration Parameters

Model configuration parameters provide a way to place external C or C++ code into specific locations and functions within code that the code generator produces.

To	Select
Insert external code near the top of the generated *model*.c or *model*.cpp file	**Additional code**, and enter the external code to insert. If you generate subsystem code into separate files, that code does not have access to external code that you specify with the **Additional code** parameter. For example, if you specify an include file as a **Additional code** setting, the code generator inserts the `#include` near the top of the *model*.c or *model*.cpp file. The subsystem code that the code generator places in a separate file does not have access to declarations inside your included file. In this case, consider specifying your external code with the **Include headers** parameter.
Insert external code near the top of the generated *model*.h file	**Include headers**, and enter the external code to insert.
Insert external code inside the model initialize function in the *model*.c or *model*.cpp file	**Initialize code**, and enter the external code to insert.
Insert external code inside the model terminate function in the *model*.c or *model*.cpp file	**Terminate code**, and enter the external code to insert. Also select the **Terminate function required** parameter on the **Interface** pane.

To	Select
Add preprocessor macro definitions	**Defines**, and enter a space-separated list of preprocessor macro definitions to add to the generated code. The list can include simple definitions (for example, `-DEF1`) and definitions with a value (for example, `-DDEF2=1`). Definitions can omit the `-D` (for example, `-DFOO=1` and `FOO=1` are equivalent). If a definition includes `-D`, the toolchain can override the flag if the toolchain uses a different flag for defines.
Use the same custom code parameter settings as the settings specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	**Use the same custom code settings as Simulation Target** This parameter refers to the **Simulation Target** pane in the Configuration Parameters dialog box.
Enable a library model to use custom code settings unique from the parent model to which the library is linked	**Use local custom code settings (do not inherit from main model)** This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

To include a header file in an external header file, add `#ifndef` code. Using this code avoids multiple inclusions. For example, in `rtwtypes.h`, the following `#include` guards are added:

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

For more information on how to add files names and locations of header, source, and shared library files to the build process, see “Build Integrated Code Within the Simulink Environment” on page 58-33.

Note The code generator includes external code that you include in a configuration set when generating code for software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. However, the code generator ignores external code that you include in a configuration set when producing code with the S-function, rapid simulation, or simulation system target file.

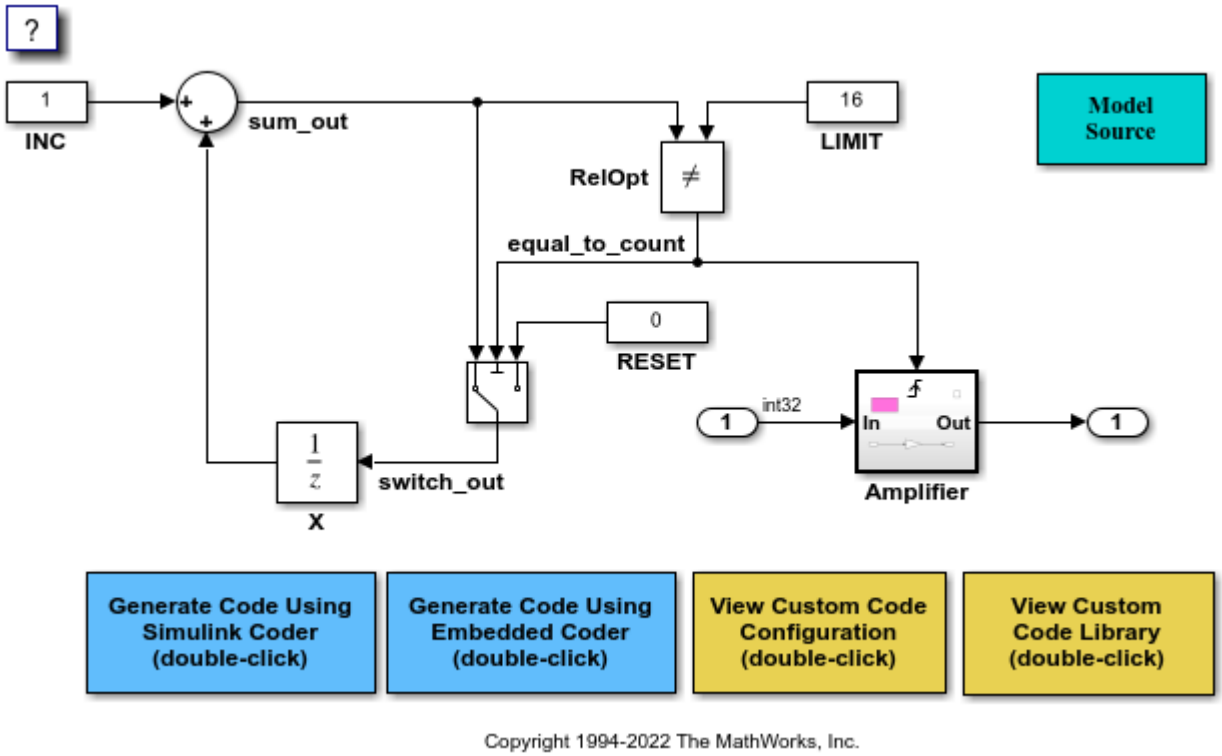
For more information about **Custom Code** parameters, see “Model Configuration Parameters: Code Generation Custom Code”. For an example, see “Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters” on page 58-28.

Integrate External C Code into Generated Code by Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



2. Open the **Simulink Coder** or **Embedded Coder** app.

3. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

4. On the **Additional source code** tab, examine the settings for model configuration parameters **Additional code** and **Initialize code**.

- **Additional code** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize code** initializes the variable `GLOBAL_INT2` to 1.

5. Close the dialog box.

6. Double-click the Model Source block. Block parameter **Top of Model Source** specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

7. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and initializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

8. Generate code and a code generation report.

9. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Additional code** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;

int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Outputs block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;

*intPtr = -1;

rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;

*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize code** configuration parameter.

```
GLOBAL_INT2 = 1;
```

See Also

More About

- “Configure Model and Generate Code”

Call External Device Drivers

Device drivers for protocols and target hardware are essential to many real-time development projects. For example, you can have a working device driver that you want to integrate with algorithmic code that has to read data from and write data to the I/O device that the driver supports. The code generator can produce a single set of application source files from an algorithm model and integrated driver code written in C or C++.

To call external device driver code from the Simulink environment, iterate through the tasks in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	“Choose an External Code Integration Workflow” on page 58-2
2	Define the representation of model data for code generation.	“Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
3	Generate S-function and TLC files by using the Legacy Code Tool. If necessary, refine the generated code manually to meet application requirements.	• “Integrate C Functions Using Legacy Code Tool” • “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6
4	Verify algorithm behavior and performance by simulating the model.	“Simulation”
5	Configure the model for code generation.	“Model Configuration Set Customization”
6	Generate code and a code generation report.	“Code Generation”
7	Review the generated code interface and static code metrics.	“Analyze Generated Data Code Interface” on page 54-13 and “Static Code Metrics” on page 54-26
8	Build an executable program from the model.	“Build Integrated Code Within the Simulink Environment” on page 58-33
9	Verify that executable program behaves and performs as expected.	“Verification, Testing, and Certification”

See Also

More About

- “Integrate C Functions Using Legacy Code Tool”
- “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6
- “Custom Targets” on page 94-2

Apply Function and Operator Code Replacements

If your generated code must use functions and operators that are consistent with external code, configure the code generator to use a code replacement library (CRL). By default, the code generator does not apply a code replacement library. You can choose from several libraries that MathWorks provides, including GNU C99 extensions, AUTOSAR 4.0, and several Intel platform-specific IPP and IPP/SSE libraries. Depending on the products that you have, other libraries might be available. If you have Embedded Coder software, you can view and choose from additional libraries and you can create custom code replacement libraries.

See Also

More About

- “What Is Code Replacement?” on page 56-2
- “What Is Code Replacement Customization?” on page 72-3
- **Code Replacement Viewer**
- **Code Replacement Tool**

Build Integrated Code Within the Simulink Environment

Workflow

To build executable programs that integrate generated code and external C or C++ code, iterate through the tasks in this table.

Task	Action	More Information
1	Choose a build approach.	“Approaches for Building Code Generated from Simulink Models” on page 59-11 For an example, see “Build Process Workflow for Real-Time Systems” on page 59-39.
2	Configure build process support for your external code.	“Configure Parameters for Integrated Code Build Process” on page 58-33
3	Configure S-Function build support for your external code.	“Build Support for S-Functions” on page 58-35 “Use makecfg to Customize Generated Makefiles for S-Functions” on page 93-15 For examples, see “Call External C Code from Model and Generated Code” on page 53-48 and “Call Reusable External Algorithm Code for Simulation and Code Generation” on page 58-15.
4	Configure build process to find the external code source, library, and header files.	“Manage Build Process File Dependencies” on page 52-43 “Control Library Location and Naming During Build” on page 93-3
5	Set up custom build processing required for your external code integration.	For the build process customization workflow, see “Customize Post-Code-Generation Build Processing” on page 93-7. To automate applying build customizations to a toolchain approach build, see “Customize Build Process with sl_customization.m” on page 93-29. To automate applying build customizations to a template makefile approach build, see “Customize Build Process with STF_make_rtw_hook File” on page 93-25.

Configure Parameters for Integrated Code Build Process

The table provides a guide to model configuration parameters on **Code Generation** > **Custom Code** pane of the Model Configuration Parameters dialog box that support the build process for external code integration. For information about folders for your external code, see “Manage Build Process Folders” on page 52-30. If you choose to place your external code in the “Code generation folder”, see “Preserve External Code Files in Build Folder” on page 58-34.

To	Select
Add include folders, which contain header files, to the build process	**Code information** > **Include directories**, and enter the absolute or relative paths to the folders. If you specify relative paths, the paths must be relative to the folder containing your model files, not relative to the build folder. The order in which you specify the folders is the order in which they are searched for header, source, and library files.

Add source files to be compiled and linked	**Code information** > **Source files**, and enter the full paths or just the file names for the files.
	
	Enter just the file name if the file is in the current MATLAB folder or in one of the include folders. For each additional source that you specify, the build process expands a generic rule in the template makefile for the folder in which the source file is located. For example, if a source file is located in folder `inc`, the build process adds a rule similar to the following:
	
	`%.obj: buildir\inc\%.c`
	`$(CC) -c -Fo$(@F) $(CFLAGS) $<`
	
	The build process adds the rules in the order that you list the source files.

Add libraries to be linked	**Code information** > **Libraries**, and enter the full paths or just the file names for the libraries. Enter just the file name if the library is located in the current MATLAB folder or in one of the include folders.
Use the same custom code settings as those specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	**Use the same custom code settings as Simulation Target** This parameter refers to the **Simulation Target** pane in the Configuration Parameters dialog box.
Enable a library model to use custom code settings unique from the parent model to which the library is linked	**Use local custom code settings (do not inherit from main model)** This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

Preserve External Code Files in Build Folder

By default, the build process deletes foreign source files. You can preserve foreign source files by following these guidelines.

If you put a `.c/.cpp` or `.h` source file in a build folder, and you want to prevent the code generator from deleting it during the TLC code generation process, insert the text `target specific file` in the first line of the `.c/.cpp` or `.h` file. For example:

```
/*  COMPANY-NAME target specific file
 *
 *   This file is created for use with the
```

```
 *   COMPANY-NAME target.
 *   It is used for ...
 */
...
```

Make sure that you spell the text “target specific file” as shown in the preceding example, and that the text is in the first line of the source file. Other text can appear before or after this text.

Flagging user files in this manner prevents postprocessing these files to indent them with generated source files. Auto-indenting occurred in previous releases to build folder files with names having the pattern *model_*.c/.cpp (where * was text). The indenting is harmless, but can cause differences detected by source control software that can potentially trigger unnecessary updates.

Build Support for S-Functions

User-written S-Function blocks provide a powerful way to incorporate external code into the Simulink development environment. In most cases, you use S-functions to integrate existing external code with generated code. Several approaches to writing S-functions are available:

- “Write Noninlined S-Function” on page 13-91
- “Write Wrapper S-Function and TLC Files” on page 13-93
- “Write Fully Inlined S-Functions” on page 13-100
- “Write Fully Inlined S-Functions with mdlRTW Routine” on page 13-102
- “S-Functions for Code Reuse” on page 13-81
- “S-Functions for Multirate Multitasking Environments” on page 13-86

S-functions also provide the most flexible and capable way of including build information for legacy and custom code files in the build process.

There are different ways of adding S-functions to the build process.

Implicit Build Support

When building models with S-functions, the build process adds rules, include paths, and source file names to the generated makefile. The source files (`.h`, `.c`, and `.cpp`) for the S-function must be in the same folder as the S-function MEX-file. Whether using the toolchain approach or template makefile approach for builds, the build process propagates this information through the toolchain or template makefile.

- If the file *sfcnname*`.h` exists in the same folder as the S-function MEX-file (for example, *sfcnname.mexext*), the folder is added to the include path.
- If the file *sfcnname*`.c` or *sfcnname*`.cpp` exists in the same folder as the S-function MEX-file, the build process adds a makefile rule for compiling files from that folder.
- When an S-function is not inlined with a TLC file, the build process must compile the S-function source file. To determine the name of the source file to add to the list of files to compile, the build process searches for *sfcnname*`.cpp` on the MATLAB path. If the source file is found, the build process adds the source file name to the makefile. If *sfcnname*`.cpp` is not found on the path, the build process adds the file name *sfcnname*`.c` to the makefile, whether or not it is on the MATLAB path.

Note For the Simulink engine to find the MEX-file for simulation and code generation, it must exist on the MATLAB path or exist in our current MATLAB working folder.

Specify Additional Source Files for an S-Function

If your S-function has additional source file dependencies, you must add the names of the additional modules to the build process. Specify the file names:

- In the **S-function modules** field in the S-Function block parameter dialog box
- With the `SFunctionModules` parameter in a call to the `set_param` function

For example, suppose you build your S-function with multiple modules.

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

You can then add the modules to the build process by doing one of the following:

- In the S-function block dialog box, specify `sfun_main`, `sfun_module1`, and `sfun_module2` in the **S-function modules** field.
- At the MATLAB command prompt, enter:

  ```
  set_param(sfun_block,'SFunctionModules','sfun_module1 sfun_module2')
  ```

  Alternatively, you can define a variable to represent the parameter value.

  ```
  modules = 'sfun_module1 sfun_module2'
            set_param(sfun_block,'SFunctionModules', modules)
  ```

The **S-function modules** field and `SFunctionModules` parameter do not support complete source file path specifications. To use the parameter, the code generator must find the additional source files when executing the makefile. For the code generator to locate the additional files, place them in the same folder as the S-function MEX-file. You can then leverage the implicit build support described in “Implicit Build Support” on page 58-35.

When you are ready to generate code, force the code generator to rebuild the top model, as described in “Control Regeneration of Top Model Code”.

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17.

Use TLC Library Functions

If you inline your S-function by writing a TLC file, you can add source file names to the build process by using the TLC library function `LibAddToModelSources`. For details, see “LibAddSourceFileCustomSection(file, builtInSection, newSection)”.

Note This function does not support complete source file path specifications. The function assumes that the code generator can find the additional source files when executing the makefile.

Another useful TLC library function is `LibAddToCommonIncludes`. Use this function in a `#include` statement to include S-function header files in the generated *`model`*`.h` header file. For details, see “LibAddToCommonIncludes(incFileName)”.

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17.

Precompile S-Function Libraries

You can precompile new or updated S-function libraries (MEX-files) for a model by using the MATLAB language function `rtw_precompile_libs`. Using a specified model and a library build specification, this function builds and places the libraries in a precompiled library folder.

By precompiling S-function libraries, you can optimize system builds. Once your precompiled libraries exist, the build process can omit library compilation from subsequent builds. For models that use numerous libraries, the time savings for build processing can be significant.

To use `rtw_precompile_libs`:

1. Set the library file suffix, including the file type extension, based on your system platform.

   Consider determining the type of platform, and then use the `TargetLibSuffix` parameter to set the library suffix accordingly. For example, when applying a suffix for a GRT target, you can set the suffix to `_std.a` for a UNIX platform and `_vcx64.lib` for a Windows platform.

   ```
   if isunix
     suffix = '_std.a';
   else
     suffix = '_vcx64.lib';
   end

   set_param(my_model,'TargetLibSuffix', suffix);
   ```

   There are a number of factors that influence the precompiled library suffix and extension. The following table provides examples for typical selections of system target file, the compiler toolchain, and other options that affect your choice of suffix and extension. For more information, examine the template make files in the `matlab/rtw/c/grt` folder or `matlab/rtw/c/ert` folder.

TMF File	COMPILER _TOOL_CHAIN Value	Precompiler Libraries (PRECOMP_LIBRARIES)			
		Library Suffix S-Function (EXPAND _LIBRARY _NAME Value)	Library Suffix Integer- Only Code (EXPAND _LIBRARY _NAME Value)	Library Suffix Optimize for Speed (EXPAND _LIBRARY _NAME Value)	Library Extension (EXPAND _LIBRARY _NAME Value)
ert_lcc64.tmf	lcc	_rtwsfcn_lcc	_int_ert_lcc	_ert_lcc	.lib
ert_vcx64.tmf	vcx64	_rtwsfcn_vcx64	_int_ert_vcx64	_ert_vcx64	.lib
ert_unix.tmf	unix	_rtwsfcn	_int_ert	_ert	.a
grt_lcc64.tmf	lcc	n/a	n/a	_lcc	.lib
grt_vcx64.tmf	vcx64	n/a	n/a	_vcx64	.lib
grt_unix.tmf	unix	n/a	n/a	_std	.a

2. Set the precompiled library folder.

   Use one of the following methods to set the precompiled library folder:

   - Set the `TargetPreCompLibLocation` parameter, as described in “Specify the Location of Precompiled Libraries” on page 93-4.
   - Set the `makeInfo.precompile` field in an `rtwmakecfg.m` function file. (For more information, see “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17.)

If you set `TargetPreCompLibLocation` and `makeInfo.precompile`, the setting for `TargetPreCompLibLocation` takes precedence.

The following command sets the precompiled library folder for model `my_model` to folder `lib` under the current working folder.

```
set_param(my_model,'TargetPreCompLibLocation', fullfile(pwd,'lib'));
```

Note If you set both the target folder for the precompiled library files and a target library file suffix, the build process detects whether any precompiled library files are missing while processing builds.

3 Define a build specification.

Set up a structure that defines a build specification. The following table describes fields that you can define in the structure. These fields are optional, except for `rtwmakecfgDirs`.

Field	Description
`rtwmakecfgDirs`	A cell array of character vectors that name the folders containing `rtwmakecfg` files for libraries to be precompiled. The function uses the `Name` and `Location` elements of `makeInfo.library`, as returned by `rtwmakecfg`, to specify the name and location of the precompiled libraries. If you set the `TargetPreCompLibLocation` parameter to specify the library folder, that setting overrides the `makeInfo.library.Location` setting. **Note:** The specified model must contain blocks that use precompiled libraries specified by the `rtwmakecfg` files because the TMF-to-makefile conversion generates the library rules only if the build process uses the libraries.
`libSuffix`	A character vector that specifies the suffix, including the file type extension, to be appended to the name of each library (for example, `.a` or `_vc.lib`). The character vector must include a period (.). You must set the suffix with either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.
`intOnlyBuild`	A Boolean flag. When set to true, the flag indicates the libraries are to be optimized such that they are compiled from integer code only. This field applies to ERT targets only.
`makeOpts`	A character vector that specifies an option to be included in the `rtwMake` command line.
`addLibs`	A cell array of structures that specify libraries to be built that are not specified by an `rtwmakecfg` function. Each structure must be defined with two fields that are character arrays: • `libName` — the name of the library without a suffix • `libLoc` — the location for the precompiled library The target makefile (TMF) can specify other libraries and how those libraries are built. Use this field to precompile those libraries.

The following commands set up build specification `build_spec`, which indicates that the files to be compiled are in folder `src` under the current working folder.

```
build_spec = [];
build_spec.rtwmakecfgDirs = {fullfile(pwd,'src')};
```

4. Issue a call to `rtw_precompile_libs`.

   The call must specify the model for which you want to build the precompiled libraries and the build specification. For example:

   ```
   rtw_precompile_libs(my_model,build_spec);
   ```

See Also

More About

- “Call Reusable External Algorithm Code for Simulation and Code Generation” on page 58-15
- “Place External C/C++ Code in Generated Code” on page 58-23
- “Call External Device Drivers” on page 58-31
- “Deploy Applications to Target Hardware” on page 71-2
- “Deploy Generated Component Software to Application Target Platforms” on page 71-34

Generate Component Source Code for Export to External Code Base

In this section...

"Modeling Options" on page 58-40
"Requirements" on page 58-40
"Limitations for Export-Function Subsystems" on page 58-42
"Workflow" on page 58-42
"Choose an Integration Approach" on page 58-43
"Generate C Function Code for Export-Function Model" on page 58-44
"Generate C++ Function and Class Code for Export-Function Model" on page 58-48
"Generate Code for Export-Function Subsystems" on page 58-52

If you have Embedded Coder software, you can generate function source code from modeling components to use in an external code base. The generated code does not include supporting scheduling code (for example, a step function). Controlling logic outside of the Simulink environment invokes the generated function code.

Modeling Options

You can generate function code to export for these modeling components:

- Export-function models (model containing functional blocks that consist exclusively of function-call subsystems, function-call model blocks, or other export-function models, as described in "Export-Function Models Overview")
- Export-function subsystems (virtual subsystem that contains function-call subsystems)

To export code that the code generator produces for these modeling components, the modeling components must meet specific requirements on page 58-40.

For models designed in earlier releases, the code generator can export functions from triggered subsystems. The requirements stated for export-function subsystems also apply to exporting functions from triggered subsystems, with the following exceptions:

- Encapsulate triggered subsystems from which you intend to export functions in a top-level virtual subsystem.
- Triggered subsystems do not have to meet requirements and limitations identified for virtual subsystems that contain function-call subsystem.
- "Export Functions That Use Absolute or Elapsed Time" on page 58-41 does not apply to exporting functions from triggered subsystems.

Requirements

- Model solver must be a fixed-step discrete solver.
- You must configure each root-level Inport block that triggers a function-call subsystem to output a function-call trigger. These Inport blocks cannot connect to an Asynchronous Task Specification block.

- Model or subsystem, must contain only the following blocks at the root level:
  - Function-call blocks (such as Function-Call Subsystem, Simulink Function, S-Functions, and Function-Call Model blocks at the root level if the solver model configuration parameter **Tasking and sample time options** > **Periodic sample time constraint** is set to `Ensure sample time independent`)
  - Inport and Outport blocks (ports)
  - Constant blocks (including blocks that resolve to constants, such as Add)
  - Blocks with a sample time of Inf
  - Merge and data store memory blocks
  - Virtual connection blocks (such as, Function-Call Split, Mux, Demux, Bus Creator, Bus Selector, Signal Specification, and virtual subsystems that contain these blocks)
  - Signal-viewer blocks, such as Scope blocks (export-function subsystems only)
- When a constant block appears at the top level of the model or subsystem, you must set the model configuration parameter **Optimization** > **Default parameter behavior** for the model or containing model to `Inlined`.
- Blocks inside the model or subsystem must support code generation.
- Blocks that use absolute or elapsed time must be inside a periodic function-call subsystem with a discrete sample time specified on the corresponding function-call root-level Inport block. See “Export Functions That Use Absolute or Elapsed Time” on page 58-41.
- Data signals that cross the boundary of an exported system cannot be a virtual bus and cannot be implemented as a Goto-From connection. Data signals that cross the export boundary must be scalar, muxed, or a nonvirtual bus.

In addition, for export-function models, you cannot generate code for a rate-based model that includes multiple instances of an export-function model. For example, you cannot generate code for a test harness model that you use for scheduling reusable export-function models during simulation.

For export-function subsystems, the following additional requirements apply:

- A trigger signal that crosses the boundary of an export-function subsystem must be scalar. Input and output data signals that do not act as triggers do not have to be scalar.
- When a constant signal drives an output port of an export-function subsystem, the signal must specify a storage class.

Export Functions That Use Absolute or Elapsed Time

If you want to export function code for a modeling component with blocks that use absolute or elapsed time, those blocks must be inside a function-call subsystem that:

- You configure for periodic execution
- You configure the root-level Inport block with a discrete sample time

To configure a function-call subsystem for periodic execution:

1. In the function-call subsystem, right-click the Trigger block and choose **Block Parameters** from the context menu.
2. Set parameter **Sample time type** to `periodic`.
3. Set **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.

4 Click **OK** or **Apply**.

For more information, see “Absolute and Elapsed Time Computation” on page 28-2.

Limitations for Export-Function Subsystems

- Subsystem block parameters do not control the names of the files containing the generated code. The file names begin with the name of the exported subsystem.
- Subsystem block parameters do not control the names of top-level functions in the generated code. Each function name reflects the name of the signal that triggers the function or (for an unnamed signal) reflects the block from which the signal originates.
- You can export function-call systems for the C++ class code interface packaging only when its function specification is set to **Default step method**. See “Interactively Configure C++ Interface” on page 44-2. The exported function is compatible with single-threaded execution. To avoid potential data race conditions for shared signals, invoke all members for the class from the same execution thread.
- The code generator supports a SIL or PIL block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include Stateflow charts.
- A Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-Call Generator block, must drive a SIL block.
- You can export an asynchronous (sample-time) function-call system, but the software does not support the SIL or PIL block for an asynchronous system.
- The use of the TLC function `LibIsFirstInit` has been removed for export-function subsystems.

Workflow

To generate code for an exported function, iterate through the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	“Choose an External Code Integration Workflow” on page 58-2
2	Verify that the model or subsystem that you are exporting satisfies function exporting requirements.	“Requirements” on page 58-40
3	Address data interface requirements by modifying the model or subsystem.	“Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” on page 58-61
4	If necessary, configure function prototype.	“Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks” on page 42-2 and, for fixed-step rate-based models, “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114 or “Interactively Configure C++ Interface” on page 44-2

Task	Action	More Information
5	If necessary , update the model to place external application-specific code in generated system functions.	“Place External C/C++ Code in Generated Code” on page 58-23
6	Verify that the functions behave and perform as expected during simulation by creating and using a test harness model. The test harness model schedules execution of the functions during simulation.	“Configure Model, Generate Code, and Simulate” on page 52-2 and, if you have Simulink Test software, “Test Authoring” (Simulink Test)
7	Configure the model or subsystem for code generation.	“Model Configuration Set Customization”
8	Generate code and a code generation report.	“Code Generation”
9	Review the generated code interface and static code metrics.	“Analyze Generated Data Code Interface” on page 54-13 and “Static Code Metrics” on page 54-26
10	Build an executable program that includes the exported function code.	“Build Integrated Code Outside the Simulink Environment” on page 53-53
11	Verify that executable program behaves and performs as expected.	“Verification, Testing, and Certification”

Choose an Integration Approach

Multiple approaches are available for generating function code for export to an external development environment. The following table compares approaches. Choose the approach that aligns best with your integration requirements. For more information on how to create export-function models, see “Export-Function Models Overview”. For more information on generating code for function call subsystems, see “Generate Component Source Code for Export to External Code Base” on page 58-40.

Condition or Requirement	Use	More Information
• Traceability between modeling elements and generated code • Local inputs (Inport block) and outputs (Outport block)	Function-call subsystem	• “Generate C Function Code for Export-Function Model” on page 58-44 • “Generate C++ Function and Class Code for Export-Function Model” on page 58-48 • “Using Function-Call Subsystems” • Function-Call Subsystem

Condition or Requirement	Use	More Information
• Control over generated function prototype • Formal input arguments (Argument Inport blocks) and output arguments (Argument Outport blocks) • Local inputs (Inport block) and outputs (Outport block)	Simulink Function block	• "Configure Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks" on page 42-2 • "Simulink Function Blocks and Code Generation" on page 8-6 • "Generate Code for Simulink Function and Function Caller" on page 8-9 • "Simulink Functions" • Simulink Function
Code responds to an initialization event	Initialize Function block	• "Generate C Function Code for Export-Function Model" on page 58-44 • "Generate C++ Function and Class Code for Export-Function Model" on page 58-48 • "Startup, Reset, and Shutdown Function Interfaces" • "Using Initialize, Reinitialize, Reset, and Terminate Functions"
Code responds to a reset event	Reset Function block	• "Generate C Function Code for Export-Function Model" on page 58-44 • "Generate C++ Function and Class Code for Export-Function Model" on page 58-48 • "Startup, Reset, and Shutdown Function Interfaces" • "Using Initialize, Reinitialize, Reset, and Terminate Functions"
Code includes entry-point functions beyond what the code generator produces by default (`model_initialize`, `model_step`, and `model_terminate`)	S-function	"S-Functions and Code Generation" on page 13-2
Single-model execution framework to use as test harness and to export code generated for portions of a model	Export-function subsystem	• "Simulink Modeling Components" • "Using Function-Call Subsystems" • Function-Call Subsystem

Generate C Function Code for Export-Function Model

This example shows how to generate function code for individual function-call subsystems in a model without generating scheduling code.

To generate function code for export:

1. Create a model that contains the functions for export.
2. Configure the code interface.

3. Create a test harness model that schedules execution of the functions during simulation.
4. Simulate the model that contains the functions by using the test harness model.
5. Generate code for the model that contains the functions.

Create Model That Contains Functions for Export

The model with functions for export must satisfy architectural constraints at the model root level. At the root level, valid blocks are:

- Inport
- Outport
- Function-Call Subsystem
- Simulink Function
- Goto
- From
- Merge

For this example, the code generator produces function code for Initialize Function, Terminate Function, and Function-Call Subsystem blocks. An Initialize Function block is executed on a model initialize event and a Terminate Function block is executed on a terminate event. For a Function-call Subsystem block, you connect the block input ports to root Inport blocks that assert function-call signals. The subsystem is executed based on the function-call signal that it receives.

For exporting functions, model `ComponentDeploymentFcn` contains two function-call subsystems (`Integrator` and `Accumulator`). Function-call subsystem `Integrator` represents an aperiodic integrator function. The function applies a forward integration method and a gain value of 1.25 to the value of state variable `x` each time the function executes. Function-call subsystem `Accumulator` represents a periodic function that increments the value of state variable `x` by 1 and applies a tunable gain value `k`. The function executes every second.

```
open_system('ComponentDeploymentFcn')
```

Configure Model Code Interface

Model `ComponentDeploymentFcn` is configured to use the service code interface that is defined in `ComponentDeploymentCoderDictionary.sldd`. The code generator uses the linked dictionary to establish default code mappings that you can view and change by using the Code Mappings Editor or code mappings programming interface.

For elements for which the default interface aligns with the requirements, you do not need to make changes. For this example, to meet code interface requirements, these adjustments were made:

- On the **Functions** tab, the default function customization template for the aperiodic integrator and periodic accumulator functions produces function names `CD_Aperiodic` and `CD_Periodic`. To align with target platform requirements for these functions, the code mappings override the default functions names, by specifying the function names `CD_integrator` and `CD_accumulator`.
- An alternate timer service interface was associated with the integrator function. The coder dictionary defines two timer service interfaces: the default interface and an alternate interface `get_CD_tick`, which aligns with the target platform timer service requirements for function names. To see the timer service setting, on the **Functions** tab, select the row for `Exported`

`Function:Aperiodic` and click the pencil icon. In the dialog box that appears, see that the **Timer Service** property is set to `get_CD_tick`. Alternatively, you can set the property in the Property Inspector.

- On the **Parameters** tab, a code identifier was specified for the Gain block gain parameter `k`. To see the identifier setting, under **Model Parameters**, select the row for the parameter and click the pencil icon. In the dialog box that appears, see the value specified for the **Identifier** property, `k`. The identifier informs the code generator how to represent the variable in the generated code. Alternatively, you can set the property in the Property Inspector.
- On the **Signals/States** tab, code identifiers were specified for the states associated with the Discrete-Time Integrator and Unit Delay blocks. To see the identifier settings (`dti` and `delay`), select the row for the state of interest and click the pencil icon. In the dialog box that appears, see the value specified for the **Identifier** property. The identifiers inform the code generator how represent corresponding variables in the generated code. Alternatively, you can set the property in the Property Inspector.

To review the code mappings, open the Embedded Coder app. Then, select **Code Interface** > **Component Interface**. The Code Mappings editor appears at the bottom of the app display.

In the Code Mappings editor, you can change the interfaces for the sender, receiver, data transfer, and timer services. To change the sender, receiver, and data transfer service, click the corresponding tab and, in the service column, select an alternative service. To change the timer service, click the **Functions** tab, select the row for the integrator function, click the pencil icon, and in the dialog that appears, select an alternative interface. The default interfaces use the outside-execution data communication method. The alternative interfaces use during-execution and direct-access data communication. Consider changing the interfaces to view differences in the generated code. If you choose to change the interfaces, for each service called along the data path of a signal, set the interfaces such that they apply the same data communication method.

For more information about model code mappings for service interfaces, see “C Service Interfaces” and Code Mappings Editor – C.

Create Test Harness Model for Simulation

When you export functions, the generated code does not include a scheduler. Create a test harness model to handle scheduling during simulation. Do not use the test harness model to generate code that you deploy.

Model `ComponentDeploymentFcnHarness` is a test harness. The model provides function-call signals to other models in this example to schedule the model contents, including the model initialize and terminate events.

```
open_system('ComponentDeploymentFcnHarness')
```

For this model, in the Embedded Coder app, **Output** is set to `Simulation Only`. This setting allows you to switch focus between the test harness model and models from which you intend to generate code without opening a separate Simulink Editor window.

Simulate the Test Harness Model

Verify that the model containing the functions that you want to export is executed as you expect by simulating the test harness model. For example, simulate `ComponentDeploymentFcnHarness`.

```
sim('ComponentDeploymentFcnHarness')
```

Generate Function Code

In the **Embedded Coder** app, switch focus to component model by double clicking `ComponentDeploymentFcn`. Then, generate code for the functions that you want to export.

```
slbuild('ComponentDeploymentFcn')
```

```
### Starting build procedure for: ComponentDeploymentFcn
### Successful completion of code generation for: ComponentDeploymentFcn

Build Summary

Top model targets built:

Model                    Action           Rebuild Reason
========================================================================================
ComponentDeploymentFcn  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 27.562s
```

Review Generated Code

Review the generated code. The code generator creates the folder `ComponentDeploymentFcn_ert_rtw` in the current working folder and places source code files in that folder. The generated code is in two primary files: header file `ComponentDeploymentFcn.h` and source code file `ComponentDeploymentFcn.c`. `ComponentDeploymentFcn.c` calls the initialize function, exported functions `Integrator` and `Accumulator`, and terminate function. Model data and entry-point functions are accessible to a caller by including the header file.

The code generator also produces `rtwtypes.h` and a services header file, which by default the coder generator names services.h. Header file `rtwtypes.h` defines data types, structures, and macros that the generated code requires. Header file `services.h` is an interface header file that resides in subfolder `services`.

To inspect the generated code, use the generated Code Interface Report or, in the Embedded Coder app, use the Code view. The report is helpful for verifying that the generated interface code aligns with the target platform requirements. For more information, see “Analyze Generated Service Code Interface Report” on page 54-21.

Write Interface Code

Open and review the Code Interface Report. To write the interface code for your execution framework, use the information in that report.

1. Include the generated header files by adding directives `#include ComponentDeploymentFcn.h`, `#include rtwtypes.h`, and `#include services.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Entry-point functions:

- Initialize entry-point function, `void CD_initialize(void)`. At startup, call this function once. The function uses a target environment receiver service to read data from nonvolatile memory by calling interface function `get_CD_initialize_InBus_NVM`.

- Terminate entry-point function, `void CD_terminate(void)`. At shutdown, call this function once. The function uses a target environment sender service to write data to nonvolatile memory by calling interface function `getref_CD_terminate_OutBus_NVM`.
- Exported periodic function, `void CD_accumulator(void)`. Call this function as needed. The function uses a target environment sender service to write the function output by calling interface function `getref_CD_accumulator_OutBus_y`.
- Exported aperiodic function, `void CD_integrator(void)`. Call this function as needed. The function uses a target environment receiver service to read input into the function by calling interface function `get_CD_integrator_inBus_u`. The CD_integrator function also uses a target environment timer service to get the target environment function clock tick by calling interface function `get_tick_outside_CD_integrator`.

More About

- "Generate Component Source Code for Export to External Code Base" on page 58-40
- "Deploy Applications to Target Hardware" on page 71-2
- "Customize Code Organization and Format" on page 55-46
- "Configure Generated C Function Interface for Model Entry-Point Functions" on page 33-114
- "Service Interfaces" on page 3-21

Generate C++ Function and Class Code for Export-Function Model

This example shows how to generate function code for an export-function model that includes a function-call subsystem. The code generator produces function and class code that does not include scheduling code.

To generate function code for export:

1. Create a model that contains the functions for export.
2. Create a test harness model that schedules execution of the functions during simulation.
3. Simulate the model that contains the functions by using the test harness model.
4. Generate code for the model that contains the functions.

Create Model That Contains Functions and C++ Class Interface for Export

The model with functions for export with a C++ model class interface must satisfy architectural constraints at the model root level. For C++ class generation, blocks that are valid at the root level are:

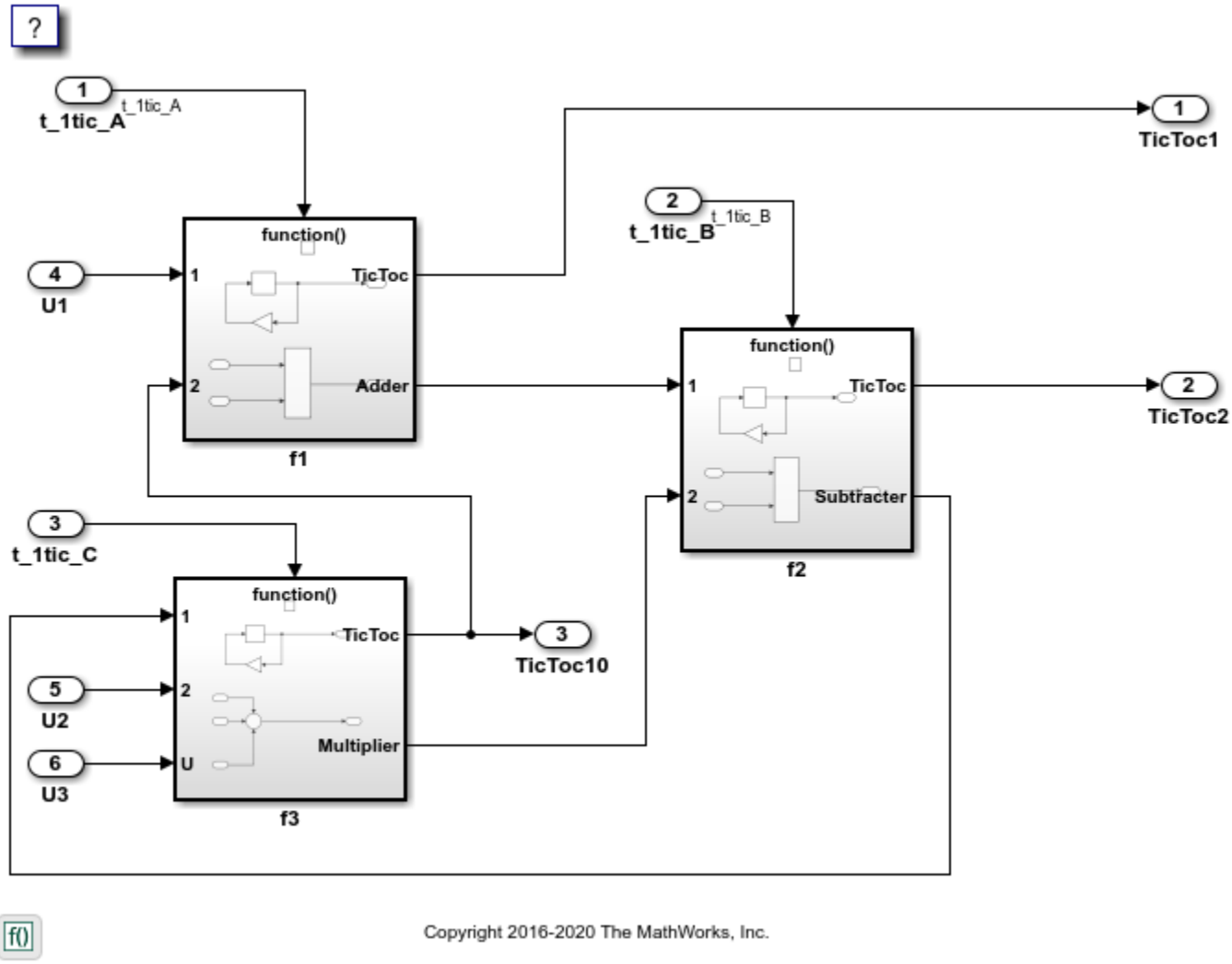
- Inport
- Outport
- Function-Call Subsystem
- Goto
- From
- Merge

Note: Export Function-Call Subsystem with C++ class interface does not support Simulink Function blocks.

The code generator produces function code for the Function-Call Subsystem block. For a Function-call Subsystem block, you connect the block input ports to root Inport blocks that assert function-call signals. The subsystem is executed based on the function-call signal that it receives.

Model `rtwdemo_cppclass_functions` contains function-call subsystems `f1`, `f2`, and `f3` for exporting functions.

```
open_system('rtwdemo_cppclass_functions')
```

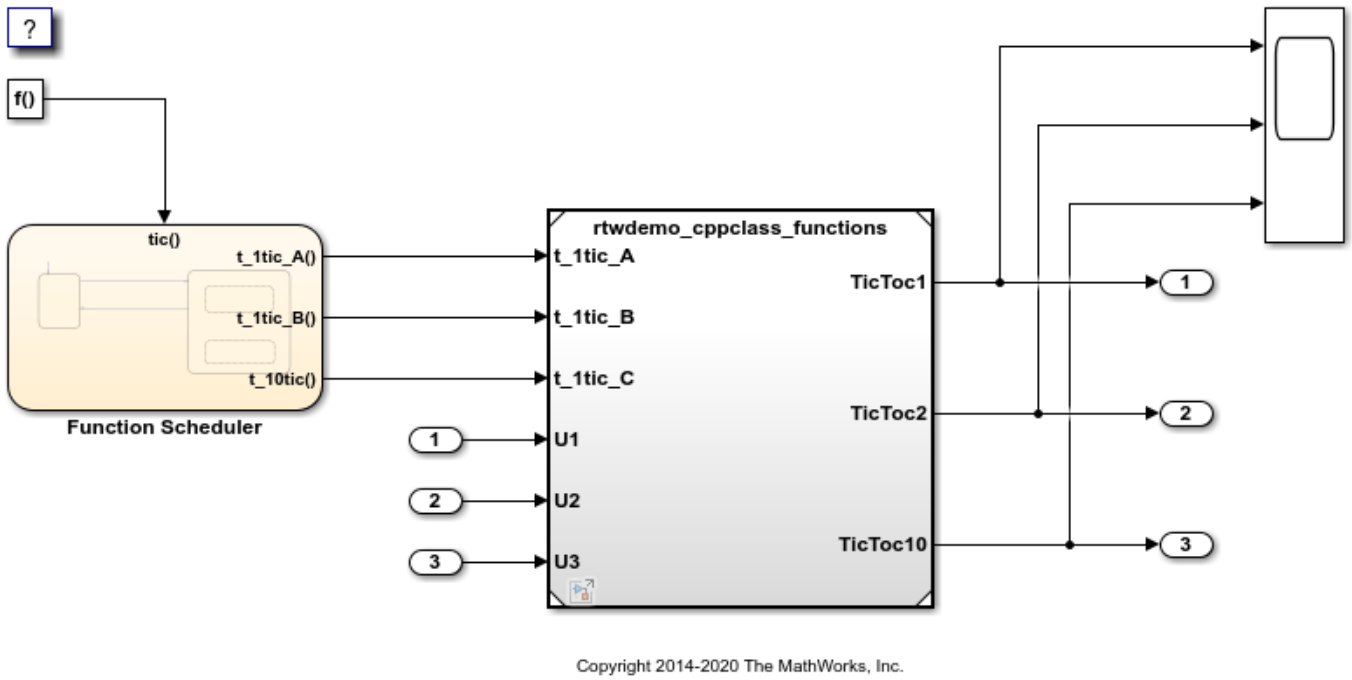


Create Test Harness Model for Simulation

When you export functions, the generated code does not include a scheduler. Create a test harness model to handle scheduling during simulation. Do not use the test harness model to generate code that you deploy.

Model `rtwdemo_cppclass_export_functions` is a test harness. The model provides function-call signals to other models in this example to schedule the model contents.

```
open_system('rtwdemo_cppclass_export_functions')
```



Simulate the Test Harness Model

Verify that the model containing the functions that you want to export is executed as you expect by simulating the test harness model. For example, simulate `rtwdemo_cppclass_export_functions`.

```
sim('rtwdemo_cppclass_export_functions')
```

Generate Function Code and Report

Generate code and a code generation report for the functions that you want to export. For example, generate code for `rtwdemo_cppclass_functions`.

```
slbuild('rtwdemo_cppclass_functions')
```

```
### Starting build procedure for: rtwdemo_cppclass_functions
### Successful completion of build procedure for: rtwdemo_cppclass_functions

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
==============================================================================================
rtwdemo_cppclass_functions  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.682s
```

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.cpp` is an example main program (execution framework) for the model. This code shows how to call the exported functions. The code also shows how to initialize and execute the generated code.
- `rtwdemo_cppclass_functions.cpp` calls the initialization function, including `Initialize Function`, and exported functions for model subsystem components `f1`, `f2`, and `f3`.
- `rtwdemo_cppclass_functions.h` declares model data structures and a public interface to the exported entry-point functions and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Write Interface Code

Open and review the Code Interface Report. To write the interface code for your execution framework, use the information in that report.

1. Include the generated header files by adding directives `#include rtwdemo_cppclass_functions.h` and `#include rtwtypes.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.U1` of type `real_T` with dimension 1
- `rtU.U2` of type `real_T` with dimension 1
- `rtU.U3` of type `real_T` with dimension 1

Entry-point functions:

- Initialize entry-point function, `void initialize(void)`. At startup, call this function once.
- Exported function, `void t_1tic_A(void)`. Call this function as needed.
- Exported function, `void t_1tic_B(void)`. Call this function as needed.
- Exported function, `void t_1tic_C(void)`. Call this function as needed.

Output ports:

- `rtY.TicToc1` of type `int8_T` with dimension [2]
- `rtY.TicToc2` of type `int8_T` with dimension [2]
- `rtY.TicToc10` of type `int8_T` with dimension 1

More About

- “Generate Component Source Code for Export to External Code Base” on page 58-40
- “Deploy Applications to Target Hardware” on page 71-2
- “Customize Code Organization and Format” on page 55-46
- “Generate Code for Simulink Function and Function Caller”

Close Example Models

```
bdclose('rtwdemo_cppclass_export_functions')
bdclose('rtwdemo_cppclass_functions')
```

Generate Code for Export-Function Subsystems

- “Specify a Custom Initialize Function Name” on page 58-52
- “Specify a Custom Description” on page 58-53
- “Optimize Code Generated for Export-Function Subsystems” on page 58-53

To generate code for an export-function subsystem:

1. Verify that the subsystem for which you are generating code satisfies exporting requirements on page 58-40.
2. In the Configuration Parameters dialog box:
   a. Set the parameter **System target file** to an ERT-based system target file, such as `ert.tlc`.
   b. If you want a SIL block with the generated code, for verification purposes, set model configuration parameter **Create block** to `SIL`.
   c. Click **OK** or **Apply**.
3. Right-click the subsystem block and select **C/C++ Code > Export Functions** from the context menu.

   The operation creates and builds a new model, *`subsystem`*`.slx`, that contains the content of the original subsystem and creates a `ScratchModel` that contains a Model block. This block references the newly created *`subsystem`*`.slx` model.

   The code generator produces code and places it in the working folder.

   If you set **Create block** to `SIL` in step 2b, Simulink opens a new window that contains an S-function block that represents the generated code. This block has the same size, shape, and connectors as the original subsystem.

Code generation and optional block creation are now complete. You can test and use the code and optional block as you do for generated ERT code and S-function block. For optional workflow tasks, see “Specify a Custom Initialize Function Name” on page 58-52 and “Specify a Custom Description” on page 58-53.

Specify a Custom Initialize Function Name

You can specify a custom name for the initialize function of your exported function as an argument to the `slbuild` command. The command takes the following form:

```
blockHandle = slbuild('subsystem', 'Mode', 'ExportFunctionCalls',..
            'ExportFunctionInitializeFunctionName', 'fcnname')
```

`fcnname` specifies the function name. For example, if you specify the name `'myinitfcn'`, the build process emits code similar to:

```
/* Model initialize function */
void myinitfcn(void){
```

```
...
}
```

Specify a Custom Description

You can enter a custom description for an exported function by using the Block Properties dialog box of an Inport block.

1. Right-click the Inport block that drives the control port of the subsystem for which you are exporting code.
2. Select **Properties**.
3. In the **General** tab, in the **Description** field, enter your descriptive text.

During function export, the text you enter is emitted to the generated code in the header for the Inport block. For example, if you open the example program `rtwdemo_exporting_functions` and enter a description in the Block Properties dialog box for port `t_1tic_A`, the code generator produces code that is similar to:

```
/*
 * Output and update for exported function: t_1tic_A
 *
 *  My custom description of the exported function
*/
void t_1tic_A(void)
{
...
}
```

Optimize Code Generated for Export-Function Subsystems

To optimize the code generated for an export-function subsystem, specify a separate storage class for each input signal and output signal that crosses the boundary of the subsystem.

For each function-call subsystem that you are exporting:

1. Right-click the subsystem.
2. From the context menu, choose **Block Parameters (Subsystem)**.
3. Select the **Code Generation** tab.
4. Set **Function packaging** to `Auto`.
5. Click **OK** or **Apply**.

See Also

More About

- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3
- “Using Triggered Subsystems”
- “Using Function-Call Subsystems”
- “Export-Function Models Overview”

Generate Shared Library for Export to External Code Base

In this section...

"About Generated Shared Libraries" on page 58-54
"Workflow" on page 58-54
"Generate Shared Libraries" on page 58-55
"Create Application Code That Uses Generated Shared Libraries" on page 58-55
"Limitations" on page 58-58

About Generated Shared Libraries

If you have Embedded Coder software, you can generate a shared library—Windows dynamic link library (`.dll`), UNIX shared object (`.so`), or Macintosh OS X dynamic library (`.dylib`)— from a model component. You or others can integrate the shared library into an application that runs on a Windows, UNIX, or Macintosh OS X development computer. Uses of shared libraries include:

- Adding a software component to an application for system simulation
- Reusing software modules among applications on a development computer
- Hiding intellectual property associated with software that you share with vendors

When producing a shared library, the code generator exports:

- Variables and signals of type `ExportedGlobal` as data
- Real-time model structure (*`model`*`_M`) as data
- Functions essential to executing the model code

Workflow

To generate a shared library from a model component and use the library, complete the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	• "Choose an External Code Integration Workflow" on page 58-2 • "Shared Library Limitations" on page 69-6
2	Configure the model for code generation.	"Generate Code That Matches Appearance of External Code" on page 58-72 "Model Configuration Set Customization"
3	Configure the model for the code generator to produce a shared library and initiate code generation.	"Generate Shared Libraries" on page 58-55

Task	Action	More Information
4	Verify that the generated shared library meets requirements. For example, review the code generation report and view the list of symbols in the library. • On Windows, use the Dependency Walker utility, downloadable from www.dependencywalker.com • On UNIX, use `nm -D` *`model`*`.so` • On Macintosh OS X, use `nm -g` *`model`*`.dylib`	
5	Use the shared library in application code.	"Create Application Code That Uses Generated Shared Libraries" on page 58-55
6	Compile and link application code that loads and uses the generated shared library.	"Build Integrated Code Within the Simulink Environment" on page 58-33
7	Verify that executable program behaves and performs as expected.	"Verification, Testing, and Certification"

Generate Shared Libraries

1. When configuring the model for code generation, select the system target file `ert_shrlib.tlc`.
2. Build the model. The code generator produces source code for the model and a shared library version of the code. The code generator places the source code in the code generation folder and the shared library (`.dll`, `.so`, or `.dylib` file) in your current working folder. The code generator also produces and retains a `.lib` file to support implicit linking.

Create Application Code That Uses Generated Shared Libraries

This example application code is generated for the example "Interface to a Development Computer Simulator by Using a Shared Library" on page 58-59.

1. Create an application header file that contains type declarations for model external input and output. For example:

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_

typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /*_APP_MAIN_HEADER_*/
```

2. In the application C source code, dynamically load the shared library. Use preprocessing conditional statements to invoke platform-specific commands. For example:

```
#if (defined(_WIN32)||defined(_WIN64)) /* WINDOWS */
#include <windows.h>
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

#else /* UNIX */
#include <dlfcn.h>
#define GETSYMBOLADDR dlsym
#define LOADLIB dlopen
#define CLOSELIB dlclose

#endif

int main()
{
    void* handleLib;
...
#if defined(_WIN64)
    handleLib = LOADLIB("./rtwdemo_shrlib_win64.dll");
#else #if defined(_WIN32)
    handleLib = LOADLIB("./rtwdemo_shrlib_win32.dll");
#else /* UNIX */
    handleLib = LOADLIB("./rtwdemo_shrlib.so", RTLD_LAZY);
#endif
#endif
...
    return(CLOSELIB(handleLib));
}
```

3 From the application C source code, access exported data and functions generated from the model. The code uses hooks to add user-defined initialization, step, and termination code.

```
   int32_T i;
 ...
    void (*mdl_initialize)(boolean_T);
    void (*mdl_step)(void);
    void (*mdl_terminate)(void);

    ExternalInputs_rtwdemo_shrlib (*mdl_Uptr);
    ExternalOutputs_rtwdemo_shrlib (*mdl_Yptr);

    uint8_T (*sum_outptr);
...
#if (defined(LCCDLL)||defined(BORLANDCDLL))
    /* Exported symbols contain leading underscores
       when DLL is linked with LCC or BORLANDC */
    mdl_initialize =
        (void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_initialize");
    mdl_step =
        (void(*)(void))GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_step");
    mdl_terminate =
        (void(*)(void))GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_terminate");
    mdl_Uptr =
        (ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
```

```
        "_rtwdemo_shrlib_U");
    mdl_Yptr =
        (ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_Y");
    sum_outptr =
        (uint8_T*)GETSYMBOLADDR(handleLib , "_sum_out");
#else
    mdl_initialize =
        (void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
        "rtwdemo_shrlib_initialize");
    mdl_step =
        (void(*)(void))GETSYMBOLADDR(handleLib ,
        "rtwdemo_shrlib_step");
    mdl_terminate  =
        (void(*)(void))GETSYMBOLADDR(handleLib ,
        "rtwdemo_shrlib_terminate");
    mdl_Uptr =
        (ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
        "rtwdemo_shrlib_U");
    mdl_Yptr =
        (ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
        "rtwdemo_shrlib_Y");
    sum_outptr =
        (uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

    if ((mdl_initialize &&
         mdl_step &&
         mdl_terminate &&
         mdl_Uptr &&
         mdl_Yptr &&
         sum_outptr)) {
        /* user application
           initialization function */
        mdl_initialize(1);
        /* insert other user defined
           application initialization code here */

        /* user application
           step function */
        for(i=0;i<=12;i++){
            mdl_Uptr->Input = i;
            mdl_step();
            printf("Counter out(sum_out):
                   %d\tAmplifier in(Input):
                   %d\tout(Output):
                   %d\n", *sum_outptr, i,
                   mdl_Yptr->Output);
            /* insert other user defined
               application step function
               code here */
        }

        /* user application
           terminate function */
        mdl_terminate();
        /* insert other user defined
           application termination
```

```
        code here */
}
else {
    printf("Cannot locate the specified
            reference(s) in the shared library.\n");
    return(-1);
}
```

Limitations

- Code generation for the `ert_shrlib.tlc` system target file exports the following as data:
  - Variables and signals of type `ExportedGlobal`
  - Real-time model structure (*`model`*_M)
- Code generation for the `ert_shrlib.tlc` system target file supports the C language only (not C ++). When you select `ert_shrlib.tlc`, language selection is unavailable on the **Code Generation** pane in the Configuration Parameters dialog box.
- To reconstruct a model simulation by using a generated shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing must be consistent so that you can compare the simulation and integration results. Additional simulation considerations apply if generating a shared library from a model that enables model configuration parameters **Support: continuous time** and **Single output/update function**. For more information, see Single output/update function (Simulink Coder) dependencies.

See Also

More About

- “Interface to a Development Computer Simulator by Using a Shared Library” on page 58-59
- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3
- “Configure a System Target File” on page 50-2
- “Manage Build Process Files” on page 52-35
- “Model Protection”

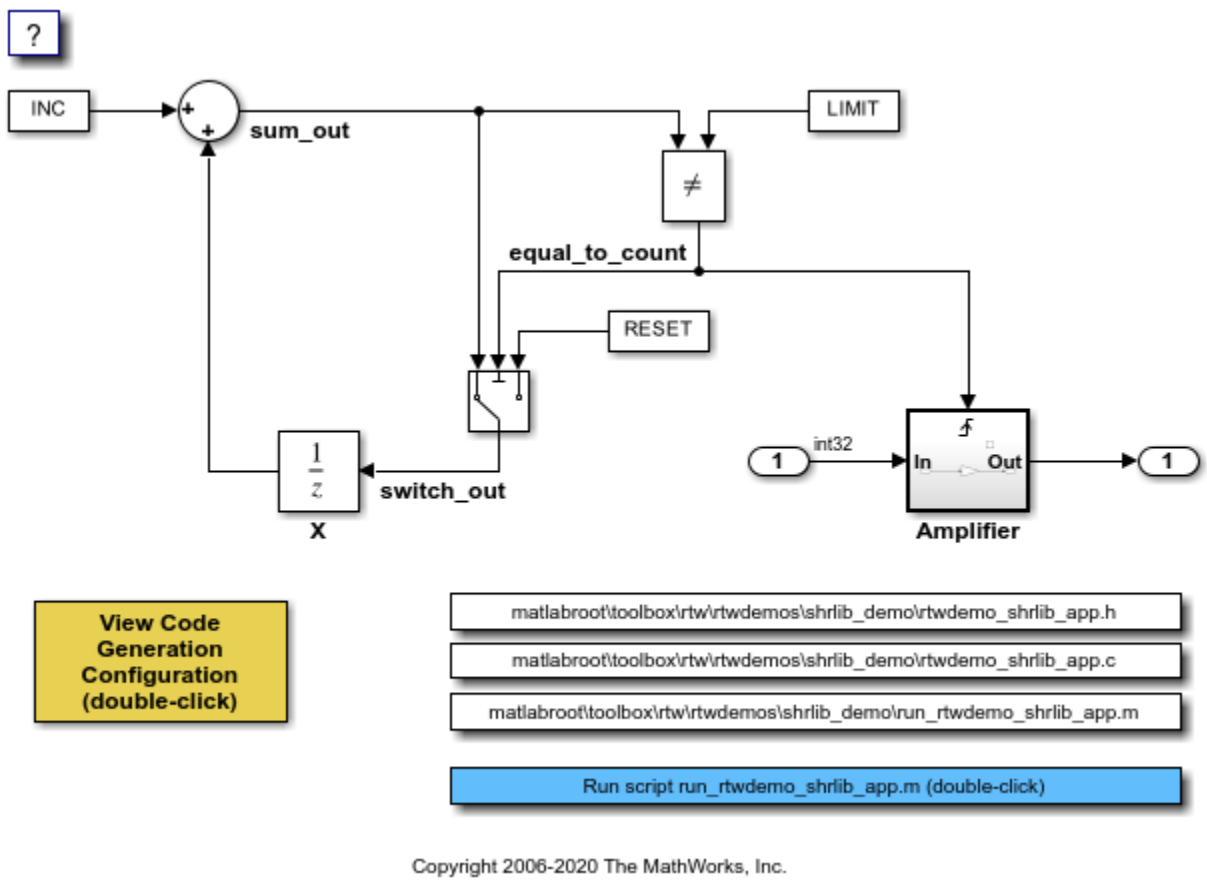
Interface to a Development Computer Simulator by Using a Shared Library

This example generates a shared library for interfacing to a simulator that runs on your development computer. Generate the shared library by using the system target file `ert_shrlib.tlc`.

To build a shared library from the model and use the library in an application:

1. Develop your model. For this example, open the model `rtwdemo_shrlib`. Save a copy of the model to a writable location.

```
open_system('rtwdemo_shrlib');
```



The model is a single-rate, discrete-time model. An 8-bit counter feeds the triggered subsystem named `Amplifier`. Parameters `INC`, `LIMIT`, and `RESET` are set to constant values 1, 4, and 0, respectively. When signal `equal_to_count` is true, the subsystem amplifies its input signal by a gain factor `K=3` and the output signal is updated.

2. Open the **Embedded Coder** app.

3. Set model configuration parameter **System target file** to `ert_shrlib.tlc`.

4. Build the shared library file. The file that the code generator produces depends on your development platform. For example, on a Windows® system, the code generator produces the library file `rtwdemo_shrlib_win64.dll`.

5. Create application code that uses the shared library. This example uses application code that is available in these files:

`matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.h` `matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.c`

To view the source code in these files, in the model, click the white buttons for the `.h` and `.c` files.

6. Compile and link the file application and shared library files to produce an executable program. The following script compiles, builds, and runs the program.

`matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\run_rtwdemo_shrlib_app.m`

To view the script code, in the model, click the white button for the `.m` file.

To build the model and run the application that uses the generated shared library, in the model, double-click the blue button.

For more information about using a shared library, see “Package Generated Code as Shared Libraries” on page 69-2.

See Also

More About

- “Generate Shared Library for Export to External Code Base” on page 58-54

Exchange Data Between External C/C++ Code and Simulink Model or Generated Code

Whether you import your external code into a Simulink model (for example, by using the Legacy Code Tool) or export the generated code to an external environment, the model or the generated code typically exchange data (signals, states, and parameters) with your code.

Functions in C or C++ code, including your external functions, can exchange data with a caller or a called function through:

- Arguments (formal parameters) of functions. When a function exchanges data through arguments, an application can call the function multiple times. Each instance of the called function can manipulate its own independent set of data so that the instances do not interfere with each other.
- Direct access to global variables. Global variables can:
  - Enable different algorithms (functions) and instances of the same algorithm to share data such as calibration parameters and error status.
  - Enable the different rates (functions) of a multitasking system to exchange data.
  - Enable different algorithms to exchange data asynchronously.

In Simulink, you can organize and configure data so that a model uses these exchange mechanisms to provide, extract, and share data with your code.

Before you attempt to match data interfaces, to choose an overall integration approach, see “Choose an External Code Integration Workflow” on page 58-2.

Import External Code into Model

To exchange data between your model and your external function, choose an exchange mechanism based on the technique that you chose to integrate the function.

- To exchange data through the arguments of your external function, construct and configure your model to create and package the data according to the data types of the arguments. Then, you connect and configure the block that calls or represents your function to accept, produce, or refer to the data from the model.

  For example, if you use the Legacy Code Tool to generate an S-Function block that calls your function, the ports and parameters of the block correspond to the arguments of the function. You connect the output signals of upstream blocks to the input ports and set parameter values in the block mask. Then, you can create signal lines from the output ports of the block and connect those signals to downstream blocks.
- To exchange data through global variables that your external code already defines, a best practice is to use a Stateflow chart to call your function and to access the variables. You write algorithmic C code in the chart so that during simulation or execution of the generated code, the model reads and writes to the variables.

  To use such a global variable as an item of parameter data (not signal or state data) elsewhere in a model, you can create a numeric MATLAB variable or `Simulink.Parameter` object that represents the variable. If you change the value of the C-code variable in between simulation runs, you must manually synchronize the value of the Simulink variable or object. If your algorithmic

code (function) changes the value of the C-code variable during simulation, the corresponding Simulink variable or object does not change.

If you choose to create a Simulink representation of the C-code variable, you can configure the Simulink representation so that the generated code reads and writes to the variable but does not duplicate the variable definition. Apply a storage class to the Simulink representation.

Technique for Integrating External Function	Mechanism to Exchange Data with Model	Examples and More Information
S-Function block	Function arguments	To call your function through an S-function that you create by using the Legacy Code Tool, see “Integrate C Functions into Simulink Models with Legacy Code Tool”.
Stateflow chart	Function arguments and direct access to global variables	To call your function and access global variables in a Stateflow chart, see “Access Custom Code Variables and Functions in Stateflow Charts” (Stateflow). For information about creating data items in a chart (which you can pass to your function as arguments), see “Add Stateflow Data” (Stateflow).
`coder.ceval` in MATLAB Function block	Function arguments	To call your function in a MATLAB Function block by using `coder.ceval`, see “Integrate C Code by Using the MATLAB Function Block”. For information about creating data items in a MATLAB Function block (which you can pass to your function as arguments), see “Create and Define MATLAB Function Block Variables”.

Export Generated Code to External Environment

To export the generated code into your external code, see “Exchange Data Between External Calling Code and Generated Code” on page 58-69.

Simulink Representations of C Data Types and Constructs

To model and reuse your custom C data such as structures, enumerations, and `typedef` aliases, use the information in these tables.

Modeling Patterns for Matching Data in External C Code

C Data Type or Construct	Example C Code	Simulink Equivalent	More Information
Primitive type alias (`typedef`)	`typedef float mySinglePrec_T;`	Create a `Simulink.AliasType` object. Use the object to: • Set the data types of signals and block parameters in a model. • Configure data type replacements for code generation. Generating code that uses an alias data type requires Embedded Coder.	For information about defining custom data types for your model, see `Simulink.AliasType` and "Manage Replacement of Simulink Data Types in Generated Code" on page 38-2. For an example that shows how to export the generated code into your external code, see "Replace and Rename Simulink Coder Data Types to Conform to Coding Standards" on page 38-24.
Array	`int myArray[6];`	Specify signal and parameter dimensions as described in "Determine Signal Dimensions". The generated code defines and accesses multidimensional data, including matrices, as column-major serialized vectors. If your external code uses a different format, consider using alternative techniques to integrate the generated code. See "Code Generation of Matrices and Arrays" on page 52-63.	For information about how the generated code stores nonscalar data (including limitations), see "Code Generation of Matrices and Arrays" on page 52-63. For an example that shows how to export the generated code into your external code, see "Reuse Parameter Data in Different Data Type Contexts". To model lookup tables, see `Simulink.LookupTable`.

C Data Type or Construct	Example C Code	Simulink Equivalent	More Information

Enumeration	`typedef enum myColorsType {`
	`  Red = 0,`
	`  Yellow,`
	`  Blue`
	`} myColorsType;`
		Define a Simulink enumeration that corresponds to your enumeration definition. Use the Simulink enumeration to set data types in a model.	To use enumerated data in a Simulink model, see “Use Enumerated Data in Generated Code”.
			
			For an example that shows how to generate enumerated parameter data, see “Enumeration” on page 26-20.
			
			For an example that shows how to export the generated code into your external code by exchanging enumerated data, see “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30.

C Data Type or Construct	Example C Code	Simulink Equivalent	More Information

Structure	`typedef struct myStructType {`
	`  int count;`
	`  double coeff;`
	`} myStructType;`
		Create a `Simulink.Bus` object that corresponds to your structure type.
		
		To create structured signal or state data, package multiple signal lines in a model into a single nonvirtual bus signal.
		
		To create structured parameter data, create a parameter object (such as `Simulink.Parameter`) that stores a MATLAB structure. Use the bus object as the data type of the parameter object.
		
		To package lookup table data into a structure, use `Simulink.LookupTable` and, optionally, `Simulink.Breakpoint` objects.
			For information about bus signals, see “Group Signals or Messages into Virtual Buses”. For information about structures of parameters, see “Organize Related Block Parameter Definitions in Structures”.
			
			For an example that shows how to import your external code into a model by using the Legacy Code Tool, see “Integrate C Function Whose Arguments Are Pointers to Structures”.
			
			For examples that show how to export the generated code into your external code, see “Exchange Structured and Enumerated Data Between Generated and External Code” on page 38-30 and “Access Structured Data Through a Pointer That External Code Defines” on page 40-10.
			
			To package lookup table data into a structure, `Simulink.LookupTable`.
			
			For general information about creating structures in the generated code, see “Organize Data into Structures in Generated Code” on page 36-147.

Additional Modeling Patterns for Code Generation (Embedded Coder)

C Data Type or Construct	Example C Code	Simulink Equivalent	More Information
Macro	`#define myParam 9.8`	Apply the storage classes `Define` and `ImportedDefine` to parameters. With macros, you can reuse a parameter value in multiple locations in an algorithm and change the parameter value between code compilations without consuming memory to store the value. Typically, macros represent engineering constants that you do not expect to change during code execution. This technique requires Embedded Coder.	"Macro Definitions (#define)" on page 26-56
Storage type qualifiers such as `const` and `volatile`	`const myParam = 9.8;`	Apply the storage classes `Const`, `Volatile`, and `ConstVolatile` to data items.	"Protect Global Data with Keywords const and volatile" on page 58-70

Bit field	`typedef struct myBitField {`
	`  unsigned short int MODE : 1;`
	`  unsigned short int FAIL : 1;`
	`  unsigned short int OK : 1;`
	`} myBitField`
		• Apply the storage class `BitField` to signals, states, and parameters whose data type is `boolean`.
		• Use a model configuration parameter to aggregate Boolean data into bit fields.
		
		These techniques require Embedded Coder.
			"Bitfields" on page 26-74
			
			"Optimize Generated Code by Packing Boolean Data into Bitfields" on page 80-11

C Data Type or Construct	Example C Code	Simulink Equivalent	More Information

Call to custom external function that reads or writes to data	External code:
	
	`/* Call this function to acquire the value of the signal. */`
	`double get_inSig(void)`
	`{`
	`  return myBigGlobalStruct.inSig;`
	`}`
	
	Generated algorithmic code:
	
	`algorithmInput = get_inSig();`
		Apply the storage class `GetSet` to signals, states, and parameters. Each data item appears in the generated code as a call to your custom functions that read and write to the target data.
		
		This technique requires Embedded Coder.
			"Access Data Through Functions with Storage Class GetSet" on page 40-25

Considerations for Other Modeling Goals

Goal	Considerations and More Information
Use Simulink to run and interact with the generated code	You can use SIL, PIL, and external mode simulations to connect your model to the corresponding generated application for simulation. When you import parameter data from your external code: • At the time that you begin an external mode simulation, the external executable uses the value that your code uses to initialize the parameter data. However, when you change the corresponding value in Simulink during the simulation (for example by modifying the `Value` property of the corresponding parameter object), Simulink downloads the new value to the executable. • SIL and PIL simulations do not import the parameter value from your code. Instead, the simulations use the parameter value from Simulink. If you include your external code in the Simulink Coder build process, duplicate data definitions can prevent the generated code from compiling. For information about SIL and PIL, see "Choose a SIL or PIL Approach" on page 88-11. For information about external mode simulation, see "External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling" on page 60-2.
Generate code comments that describe attributes of exported data including physical units, real-world initial value, and data type	Generating these comments can help you match data interfaces while handwriting integration code. See "Add Custom Comments for Variables in the Generated Code" on page 55-3.

See Also

Related Examples

- "How Generated Code Exchanges Data with an Environment" on page 36-35
- "Generate Code That Matches Appearance of External Code" on page 58-72

- “Design Data Interface by Configuring Inport and Outport Blocks” on page 36-169
- “Configure Generated Code According to Interface Control Document Specifications” on page 40-54
- “Generate Code That Dereferences Data from a Literal Memory Address” on page 71-15

Exchange Data Between External Calling Code and Generated Code

To export the generated code into your external code, you configure the generated code to match the data interface of your external code. For example, if your external code defines a global variable for storing output data and you need the generated code to read that data as input, you can configure a corresponding Inport block so that the generated code interacts with the existing variable.

- You can generate reentrant code from a model. The generated entry-point functions typically accept data through arguments. Your calling code passes data through these arguments. You can call the functions multiple times in a single application—the application can maintain multiple instances of the model. See “Data Exchange for Reentrant Generated Code”.
- When you generate nonreentrant code, by default, the entry-point functions exchange data between the generated code and your code through global variables. You can generate variable definitions for your code to use or you can share and reuse existing variables that your code already defines. See “Data Exchange for Nonreentrant Generated Code”.

  Alternatively, you can configure the entry-point functions to exchange system inputs and outputs (root-level Inport and Outport blocks) through arguments instead of global variables. Apply function prototype control to the model, which requires Embedded Coder. For more information about function prototype control, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

For general information about exchanging data between generated and external code, including how to match specific C code patterns, see “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code”.

Data Exchange for Reentrant Generated Code

When you generate reentrant code, the model entry-point functions exchange data through arguments. You can control some of the characteristics of the arguments. For more information, see “Control Data and Function Interface in Generated Code”.

For information about creating a separate set of data for each call site in your external code, see “Modify Static Main to Allocate and Access Model Instance Data” on page 71-10.

Data Exchange for Nonreentrant Generated Code

To make the generated code read or write to an item of signal, state, or parameter data as a global variable, apply a storage class to the data in the model. The storage class also determines whether the generated code exports the variable definition to your external code or imports the definition from your code. For general information about controlling the data interface of a model that you configure to generate nonreentrant code, see “Control Data Interface for Nonreentrant Code”. For examples, see “Design Data Interface by Configuring Inport and Outport Blocks” and “Integrate External Application Code with Code Generated from PID Controller” on page 40-45.

When you generate code that defines (allocates memory for) global data, the generated code exports that data. When your external code defines data, the generated code imports that data. Typically, storage classes that import data have the word `Import` in the storage class name, for example, `ImportedExternPointer`.

Control File Placement of Exported Global Data (Embedded Coder)

When you export data from the generated code by using storage classes, the code generator creates an `extern` declaration. By default, this declaration typically appears in the generated header file *`model`*`.h`. You can include (`#include`) this header file in your external code.

By default, the definition (memory allocation) of exported data typically appears in *`model`*`.c`.

You can control the file placement of the declarations and definitions to:

- Create separate object files that store only global parameter data.
- Modularize the generated code by organizing declarations into separate files.

For more information about controlling file placement of declarations and definitions, see “Control Placement of Global Data Definitions and Declarations in Generated Files” on page 37-2. For an example, see “Definition, Initialization, and Declaration of Parameter Data” on page 26-5.

Prevent Duplicate Initialization Code for Global Variables

When your external code defines global variables, you can generate code that interacts with those variables. For example, use the storage class `ImportFromFile` (see “Choose Storage Class for Controlling Data Representation in Generated Code”).

For imported variables that represent signal or state data, the code generator can produce initialization code as described in “Initialization of Signal, State, and Parameter Data in the Generated Code”. If your code already initializes a variable, consider preventing the generation of duplicate initialization code. Create your own storage class and, for that storage class, set **Data initialization** to `None`. For information about creating and applying your own storage class, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture” on page 32-2.

Protect Global Data with Keywords const and volatile

When your external code and the generated code exchange data through global variables, you can generate code that uses C type qualifiers `const` and `volatile` to protect data integrity and improve the safety of your application. For example:

- Apply `const` to a calibration parameter.
- Apply `volatile` to a global variable that stores the output of a hardware device operating asynchronously.
- Apply `const` and `volatile` to signals, states, and parameters that represent data defined by your external code. Then, the generated code declares and interacts with the external data by using the corresponding storage type qualifiers.

You must have Embedded Coder. For more information, see “Protect Global Data with const and volatile Type Qualifiers” on page 45-13.

See Also

Related Examples

- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code”

- “Control Data and Function Interface in Generated Code”
- “Design Data Interface by Configuring Inport and Outport Blocks”
- “Integrate External Application Code with Code Generated from PID Controller” on page 40-45

Generate Code That Matches Appearance of External Code

A key aspect of code integration, especially for larger projects, is adherence to guidelines and standards for code appearance. If code appearance requirements apply to your project, review the requirements in this table. To learn more, see the relevant information.

Requirement	More Information
Thoroughly document code or document code in a specific way.	• "Configure Code Comments" on page 48-22 • "Annotate Code for Justifying Polyspace Checks" on page 55-86 • "Add Custom Comments for Variables in the Generated Code" on page 55-3 • "Add Global Comments in the Generated Code" on page 55-5
Control the length and naming of code identifiers (symbols), including the use of reserved names.	• "Identifier Format Control" on page 55-17 • "Specify Identifier Length to Avoid Naming Collisions" on page 48-29 • "Specify Reserved Names for Generated Identifiers" on page 48-30 • "Customize Generated Identifier Naming Rules" on page 55-10 • "Avoid Identifier Name Collisions with Referenced Models" on page 55-26 • "Control Name Mangling in Generated Identifiers" on page 55-25 • "Specify Boolean and Data Type Limit Identifiers" on page 38-37

Requirement	More Information
Control style aspects of code: • Level of parenthesization • Order of operations in expressions • Empty primary condition expressions in `if` statements • `if-elseif-else` or `switch-case` statements for decision logic • Use of `extern` keyword in function declarations • Use of `static` keyword in function declarations • Use of `default` cases for `switch-case` statements • Use multiplications by powers of two or signed bitwise shifts • Cast expressions • Indent style (Kernighan and Ritchie or Allman) and size • Newline style	"Model Configuration Parameters: Code Style"
Placement of data definitions and declarations, including location of global identifiers and global data declarations (`extern`)	"Control Placement of Global Data Definitions and Declarations in Generated Files" on page 37-2
Appearance of code for flow charts	"Enhance Readability of Code for Flow Charts" on page 55-95
Apply code templates to control organization of code and use of banners	• "Customize Code Organization and Format" on page 55-46 • "Template Symbols and Rules" on page 55-81 • "Specify Templates For Code Generation" on page 55-48 • "Generate Custom File and Function Banners" on page 55-74 • "Change the Organization of a Generated File" on page 55-58 • "Generate Source and Header Files with a Custom File Processing (CFP) Template" on page 55-62 • "Customize Generated File Names" on page 55-60

See Also

More About

- "Choose an External Code Integration Workflow" on page 58-2
- "Model Configuration Set Customization"

Program Building, Interaction, and Debugging in Simulink Coder

- “Select C or C++ Programming Language” on page 59-2
- “Select and Configure C or C++ Compiler” on page 59-3
- “Troubleshoot Compiler Issues” on page 59-7
- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Configure CMake Build Process” on page 59-13
- “Create Custom CMake Toolchain Definition” on page 59-17
- “Configure Toolchain (ToolchainInfo) or Template Makefile Build Process” on page 59-20
- “Define Custom Makefile-Based Toolchains Using Target Framework” on page 59-30
- “Template Makefiles and Make Options” on page 59-36
- “Build Process Workflow for Real-Time Systems” on page 59-39
- “Build Models from a Windows Command Prompt Window” on page 59-45
- “Rebuild a Model” on page 59-48
- “Control Regeneration of Top Model Code” on page 59-49
- “Reduce Build Time for Referenced Models by Using Parallel Builds” on page 59-51
- “Relocate or Share Generated Code” on page 59-56
- “Compile and Debug Generated C Code with Microsoft Visual Studio” on page 59-61
- “Executable Program Generation” on page 59-62

Select C or C++ Programming Language

The default programming language selection for code generation is C language. In the code generation workflow, after the steps in “Select a Solver That Supports Code Generation” and “Select a System Target File from STF Browser”, an optional step is to change the programming language selection for code generation.

To change the programming language setting:

1. In the Configuration Parameters dialog box, from the **Language** list, select `C` or `C++`. Alternatively, set the `TargetLang` parameter at the command line.

   The code generator produces `.c` or `.cpp` files, depending on your selection, and places the generated files in your build folder.

   For more information, see Language.
2. Check whether you must choose and configure a compiler. If you select C++, you must choose and configure a compiler. For details, see “Select and Configure C or C++ Compiler”.
3. Check whether the language standard is configured for your compiler. By default, the code generator uses the ISO/IEC 9899:1999 C (`C99 (ISO)`) library for the C language and the ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) library for the C++ language.

   For more information, see Language standard.

See Also

More About

- “Select and Configure C or C++ Compiler”
- “Troubleshoot Compiler Issues”

Select and Configure C or C++ Compiler

The build process requires a supported compiler. *Compiler*, in this context, refers to a development environment containing a linker and make utility, and a high-level language compiler. For details on supported compiler versions, see:

`https://www.mathworks.com/support/compilers/current_release`

When creating an executable program, the build process must be able to access a supported compiler. The build process can find a compiler to use based on your default MEX compiler.

The build process also requires the selection of a toolchain or template makefile. The toolchain or template makefile determines which compiler runs, during the make phase of the build. For more information, see “Configure Toolchain (ToolchainInfo) or Template Makefile Build Process” on page 59-20.

To determine which templates makefiles are available for your compiler and system target file, see “Compare System Target File Support Across Products”.

For both generated files and user-supplied files, the file extension, `.c` or `.cpp`, determines whether the build process uses a C or a C++ compiler. If the file extension is `.c`, the build process uses C compiler to compile the file, and the symbols use the C linkage convention. If the file extension is `.cpp`, the build process uses a C++ compiler to compile the file, and the symbols use the C++ linkage specification.

Language Standards Compliance

The code generator produces code that is compliant with the following standards:

Language	Supported Standard	Other Names
C	ISO/IEC 9899:1990	C89/C90
	ISO/IEC 9899:1999	C99
C++	ISO/IEC 14882:2003	C++03
	ISO/IEC 14882:2011	C++11

Code that the code generator produces from these sources is ANSI C/C++ compliant:

- Simulink built-in block algorithmic code
- Generated system-level code (task ID [TID] checks, management, functions, and so on)
- Code from other blocksets, including the Fixed-Point Designer product and the Communications Toolbox product
- Code from other code generators, such as MATLAB functions

Also, the code generator can incorporate code from:

- Embedded system target files (for example, startup code, device driver blocks)
- Custom S-functions or TLC files

Note Coding standards for these two sources are beyond the control of the code generator. These standards can be a source for compliance problems, such as code that uses C99 features not supported in the ANSI C, C89/C90 subset.

Programming Language Considerations

The code generator produces C and C++ code. Consider the following as you choose a programming language:

- Does your project require you to configure the code generator to use a specific compiler? C/C++ code generation on Windows requires this selection.
- Does your project require you to change the default language configuration setting for the model? See “Select C or C++ Programming Language” on page 59-2.
- Does your project require you to integrate legacy or custom code with generated code? For a summary of integration options, see “Choose an External Code Integration Workflow” on page 58-2.
- Does your project require you to integrate C and C++ code? If so, see “Choose an External Code Integration Workflow” on page 58-2.

  Note You can mix C and C++ code when integrating generated code with custom code. However, you must be aware of the differences between C and default C++ linkage conventions, and add the `extern "C"'` linkage specifier where required. For the details of the differing linkage conventions and how to apply `extern "C"`, refer to a C++ programming language reference book.
- Does your project require code generation support from other products? See “C++ Language Support Limitations” on page 59-4.

For C++ code generation examples with Stateflow, see the `sfcndemo_cppcount` model or `sf_cpp` model.

C++ Language Support Limitations

To use C++ language support, you could need to configure the code generator to use a specific compiler. For example, if a supported compiler is not installed on your Microsoft Windows computer, the default compiler is the `lcc` C compiler shipped with the MATLAB product. This compiler does not support C++. If you do not configure the code generator to use a C++ compiler before you specify C++ for code generation, the software produces an error message.

Code generator limitations on C++ support include:

- The code generator does not support C++ code generation for the following:
  Simscape Driveline
  Simscape Multibody First Generation (Simscape Multibody Second Generation is supported)
  Simscape Electrical Power Systems
  Simulink Real-Time
- For ERT and ERT-based system target files with **Code interface packaging** set to `Nonreusable function`, the following fields currently do not support the `.cpp` extension.

- **Data definition filename**
- **Definition file** for a data object, which is an object of the `Simulink.Signal` or `Simulink.Parameter` classes or their subclasses.

If you specify a file name with a `.c` extension or with no extension and specify C++ for the code generation language, the code generator produces a `.cpp` file.

Code Generator Assumes Wrap on Signed Integer Overflows

The code generator reduces memory usage and enhances generated code execution by assuming signed integer C operations wrap on overflow. A signed integer overflow occurs when the result of an arithmetic operation is outside the range of values that the output data type can represent. The C programming language does not define the results of such operations. Some C compilers aggressively optimize signed operations for in-range values at the expense of overflow conditions. Other compilers preserve the full wrap-on-overflow behavior. For example, the gcc and MinGW compilers provide an option to wrap on overflow reliably for signed integer overflows. The generated program image for a model can produce results that differ from model simulation results because the handling of overflows varies, depending on your compiler.

When you generate code, if you use a supported compiler with the default options configured by the code generator, the compiler preserves the full wrap-on-overflow behavior. If you change the compiler options or compile the code in another development environment, it is possible that the compiler does not preserve the full wrap-on-overflow behavior. In this case, the executable program can produce unpredictable results.

If this issue is a concern for your application, consider one or more of the following actions:

- Verify that the compiled code produces expected results.
- If your compiler can force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, configure the build process to use the compiler option `-fwrapv`.
- Choose a compiler that wraps on integer overflow.
- If you have Embedded Coder installed, develop and apply a custom code replacement library to replace code generated for signed integers. For more information, see “Code Replacement Customization”.

Choose and Configure Compiler

To view the installed compilers and select the default compiler, in the Command Window, type:

```
mex -setup
```

On a Windows computer, you can install supported compilers and select a default compiler.

On a UNIX platform, the default compiler is GNU `gcc/g++` for GNU or Xcode for Mac.

Unless the build approach configuration selects a specific compiler, the code generator uses the default compiler for the build process.

Primarily, the specified system target file determines the compiler that the code generator requires:

- If you select a toolchain-based system target file such as `grt.tlc` (Generic Real-Time Target), `ert.tlc` (Embedded Coder), or `autosar.tlc` (Embedded Coder for AUTOSAR), the **Build process** subpane displays configuration parameters for the build process. Use the **Toolchain** configuration parameter to select a compiler and associated tools for your model build. To validate the selected toolchain, click the **Validate Toolchain**.
- If you select a template makefile (TMF) based system target file, such as `rsim.tlc`, the **Build process** subpane displays template makefile parameters for configuring the build process. The **Template makefile** parameter displays the default TMF file for the selected system target file. If the system target file supports compiler-specific template makefiles (for example, Rapid Simulation or S-Function system target files), you can set **Template makefile** to a compiler-specific TMF. For valid TMF names, see "Compare System Target File Support Across Products".

If the generated code exceeds the 2 Gb static memory limit, a compiler error occurs. To resolve this issue:

- For GRT-based system target files, set model configuration parameter **Code interface packaging** to `Reusable function`.
- For ERT-based system target files, set model configuration parameter **Code interface packaging** to `Reusable function` and select parameter **Use dynamic memory allocation for model initialization**.

Include S-Function Source Code

When the code generator builds models with S-functions, source code for the S-functions can be either in the current folder or in the same folder as their MEX-file. The code generator adds an include path to the generated makefiles whenever it finds a file named *`sfncname`*`.h` in the same folder as the S-function MEX-file. This folder must be on the MATLAB path.

Similarly, the code generator adds a rule for the folder when it finds a file *`sfncname`*`.c` (or `.cpp`) in the same folder as the S-function MEX-file is in.

See Also

More About

- "Configure Run-Time Environment Options"
- "Compare System Target File Support Across Products"
- "Select C or C++ Programming Language"
- "Troubleshoot Compiler Issues" on page 59-7

External Websites

- https://www.mathworks.com/support/compilers/current_release

Troubleshoot Compiler Issues

In this section...

"Compiler Version Mismatch Errors" on page 59-7
"Results for Model Simulation and Program Execution Differ" on page 59-7
"Generates Expected Code and Produces Unexpected Results" on page 59-8
"Compile-Time Issues" on page 59-8
"LCC Compiler Does Not Support Ampersands in Source Folder Paths" on page 59-9
"LCC Compiler Might Not Support Line Lengths of Rapid Accelerator Code" on page 59-9

Compiler Version Mismatch Errors

Description

The build process produces a compiler version mismatch error.

Action

1. Check the list of supported and compatible compilers available at `www.mathworks.com/support/compilers/current_release/`.
2. Upgrade or change your compiler. For more information, see "Choose and Configure Compiler" on page 59-5.
3. Rebuild the model.

Results for Model Simulation and Program Execution Differ

Description

The program generated for the model produces different results from model simulation results. The generated source code includes an arithmetic operation that produces a signed integer overflow. It is possible that your compiler does not implement wrapping behavior for signed integer overflow conditions. Or, if you are using a compiler that supports wrapping, it is possible that you did not configure it to use the `-fwrapv` option.

For more information, see "Code Generator Relies on Undefined Behavior of C Language for Integer Overflows."

Action

- If your compiler can force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, specify the compiler option `-fwrapv`.
- Choose a compiler that checks for integer overflows.
- If you have Embedded Coder, develop and apply a code replacement library to replace code generated for signed integers.

Generates Expected Code and Produces Unexpected Results

Description

The build process generates expected source code, but the executable program produces unexpected results. The generated source code appears as expected. However, the executable program produces unexpected results.

Action

Do one of the following:

- Lower the compiler optimization level.
  1. Set the **Compiler optimization level** configuration parameter to `Custom`.
  2. In the **Custom compiler optimization flags** field, specify a lower optimization level.
  3. Rebuild the model.
- Disable compiler optimizations.
  1. Set **Compiler optimization level** to `Optimizations off (faster builds)`.
  2. Rebuild the model.

For more information, see “Control Compiler Optimizations” and your compiler documentation.

Compile-Time Issues

Issue	Action
Error is present in the compiler configuration.	Make sure that MATLAB supports the compiler and version that you want to use. For a list of currently supported and compatible compilers, see `www.mathworks.com/support/compilers/current_release/`. If necessary, upgrade or change your compiler (see “Choose and Configure Compiler” on page 59-5 or “Choose and Configure Compiler” on page 59-5).
Environment variables are incorrectly set up for your make utility, compiler, or linker. For example, installation of Cygwin tools on a Windows platform affects environment variables used by other compilers.	Review the environment variable settings for your system by using the `set` command on a Windows platform or `setenv` on a Linux platform. Make sure that the settings match what is required for the tools you are using.
Error is present in custom code specified as an S-Function block or in **Custom Code**. For example, the code refers to a header file that the compiler cannot find.	To isolate the source of the problem, remove the custom code from the model, debug, and rebuild the model.
The model includes a block, such as a device driver block, which is not intended for use with the currently selected system target file.	Remove the system target file-specific block or configure the model for use with another system target file.

Issue	Action
A linker error about an undefined reference to the data appears when the model build generates an executable program from the model reference hierarchy and these conditions are true: • You represent signal, state, or parameter data by creating a data object such as `Simulink.Signal`. You use the object in a model reference hierarchy. • You use a custom storage class with the data object. Custom storage classes require Embedded Coder. • You set the owner of the object to a model that does not directly access the data. • You use the toolchain `lcc-win64`.	To resolve the issue, choose one of these methods: • In the data object, clear the `Owner` property. Alternatively, set the owner to a model that directly accesses the data. • Use a different toolchain, such as `gcc`, instead of `lcc`.

LCC Compiler Does Not Support Ampersands in Source Folder Paths

Description

If you use the LCC compiler and your model folder path contains an ampersand (&), the build process produces an error.

Action

Remove the ampersand from the model folder path. Then, rebuild the model.

LCC Compiler Might Not Support Line Lengths of Rapid Accelerator Code

Description

If you are compiling Rapid Accelerator code, the LCC compiler might produce an error related to line limits. Rapid Accelerator code can have longer line lengths due to obfuscation.

Action

Compile your Rapid Accelerator code using a compiler that supports longer code lines.

See Also

More About

- “Choose and Configure Compiler” on page 59-5
- “Select C or C++ Programming Language”
- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Build Process Workflow for Real-Time Systems” on page 59-39

- “Rebuild a Model” on page 59-48
- “Reduce Build Time for Referenced Models by Using Parallel Builds” on page 59-51
- “Control Regeneration of Top Model Code” on page 59-49
- “Relocate or Share Generated Code” on page 59-56
- “Control Compiler Optimizations”
- “Select and Configure C or C++ Compiler” on page 59-3
- “Executable Program Generation” on page 59-62

External Websites

- `www.mathworks.com/support/compilers/current_release/`

Approaches for Building Code Generated from Simulink Models

Use one of these approaches for building (compiling and linking) code that you generate from Simulink models:

- CMake — CMake is a third-party, open-source tool for build process management, which uses configuration (CMakeLists.txt) files to generate standard build files for native build environments, for example, makefiles, Ninja files, or Microsoft Visual Studio and Xcode projects. You can:
  - Specify a generic CMake toolchain –– see “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
  - Specify a shipped CMake toolchain definition –– see “Configure CMake Build Process” on page 59-13
  - Provide a custom CMake toolchain definition by using the `target` package –– see “Create Custom CMake Toolchain Definition” on page 59-17.

  MATLAB ships the CMake executable file. Simulink Coder and Embedded Coder support CMake versions from 3.12.0 onwards.
- Makefile-based toolchain — The build process generates makefiles and supports custom toolchains. You can define and register custom makefile-based toolchains by using the Target Framework or `ToolchainInfo` objects. See:
  - “Define Custom Makefile-Based Toolchains Using Target Framework” on page 59-30
  - “Register Custom Toolchain and Build Executable” on page 93-21
- Template makefile — The build process uses a template makefile with a toolchain that you specify. Optionally, you can control the build process through a toolchain information object. See “Template Makefile Approach” on page 59-24.

Each build approach supports:

- Simulink model configuration parameters.
- `slbuild`, **Ctrl+B**, and `codebuild`
- “SIL and PIL Simulations” on page 88-2
- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2

For information about customizing a build approach, see “Code Compilation Customization”.

See Also

`slbuild` | `codebuild`

Related Examples

- “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
- “Configure CMake Build Process” on page 59-13
- “Create Custom CMake Toolchain Definition” on page 59-17
- “Define Custom Makefile-Based Toolchains Using Target Framework” on page 59-30
- “Configure Toolchain (ToolchainInfo) or Template Makefile Build Process” on page 59-20

- “Compile Code in Another Development Environment” on page 52-99
- “Build Library or Executable from AUTOSAR Adaptive Model” (AUTOSAR Blockset)

External Websites

- https://cmake.org/

Configure CMake Build Process

CMake is a third-party, open-source tool for build process management. To build code generated from Simulink models, the software provides CMake toolchain definitions for:

- Microsoft Visual C++ and MinGW on Windows, GCC on Linux, and Xcode on Mac computers, using Ninja and makefile generators.
- Microsoft Visual Studio and Xcode project builds.

If a supported toolchain is installed on your development computer, you can specify the corresponding CMake toolchain definition for your model. When you run `slbuild`, press **Ctrl+B**, or run a software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulation, CMake:

1. Uses configuration (`CMakeLists.txt`) files to generate standard build files.
2. Runs the compiler and other build tools to create executable code.

You can also:

- Provide custom CMake toolchain definitions by using the `target` package. For more information, see “Create Custom CMake Toolchain Definition” on page 59-17.
- Generate only code and a configuration file for algorithm export.

Specify CMake Toolchain Definition

In the Configuration Parameters dialog box:

1. On the Hardware Implementation pane, specify parameter values for your target device, which determines the selection of toolchain definitions that is available. For more information, see “Available CMake Toolchain Definitions” on page 59-14.
2. On the Code Generation pane, specify these parameters:
   - **Toolchain** — From the drop-down list, select a CMake toolchain definition.
   - **Build configuration** — Select one of these values:
     - `Release` — Optimizations enabled.
     - `Debug` — Optimizations disabled, debug symbols generated.
     - `RelWithDebInfo` — Optimizations enabled, debug symbols generated.
     - `MinSizeRel` — Optimized for size, debug symbols not generated.
     - Specify — Enables you to specify toolchain details through the **CMake Configure** and **CMake Build** fields.
   - **Toolchain details** — If **Build configuration** is `Specify`, populate these fields:
     - **CMake Configure** — Additional command-line arguments to pass to CMake when generating the project build system builder artifacts for the downstream tools.
     - **CMake Build** — Additional command-line arguments to pass to CMake when using it in build mode.

For more information, see https://cmake.org/cmake/help/latest/manual/cmake.1.html.

Available CMake Toolchain Definitions

Platform	Shipped Toolchain Definition	Device Vendor and Type that Make Definition Available
Windows	Microsoft Visual C++ 2017 v15.0 | CMake/nmake (64-bit Windows)	Intel-x86-64 (Windows64) AMD-x86-64 (Windows64)
	Microsoft Visual C++ 2017 v15.0 | CMake/Ninja (64-bit Windows)	
	Microsoft Visual Studio Project 2017 | CMake (64-bit Windows)	
	Microsoft Visual Studio Project 2019 | CMake (64-bit Windows)	
	Microsoft Visual C++ 2019 v16.0 | CMake/Ninja (64-bit Windows)	
	Microsoft Visual C++ 2022 v17.0 | CMake/nmake (64-bit Windows)	
	Microsoft Visual C++ 2022 v17.0 | CMake/Ninja (64-bit Windows)	
	MinGW64 | CMake/gmake (64-bit Windows)	
	MinGW64 | CMake/Ninja (64-bit Windows)	
	Microsoft Visual C++ 2017 v15.0 | CMake/nmake (32-bit Windows)	Intel-x86-32 (Windows32) AMD-x86-32 (Windows32)
	Microsoft Visual C++ 2017 v15.0 | CMake/Ninja (32-bit Windows)	
	Microsoft Visual Studio Project 2017 | CMake (32-bit Windows)	
	Microsoft Visual Studio Project 2019 | CMake (32-bit Windows)	
	Microsoft Visual C++ 2019 v16.0 | CMake/nmake (32-bit Windows)	
	Microsoft Visual C++ 2019 v16.0 | CMake/Ninja (32-bit Windows)	
	Microsoft Visual C++ 2022 v17.0 | CMake/nmake (32-bit Windows)	
	Microsoft Visual C++ 2022 v17.0 | CMake/Ninja (32-bit Windows)	
Linux	GNU gcc/g++ | CMake/gmake (64-bit Linux)	Intel-x86-64 (Linux 64) AMD-x86-64 (Linux 64)
	GNU gcc/g++ | CMake/Ninja (64-bit Linux)	

Platform	Shipped Toolchain Definition	Device Vendor and Type that Make Definition Available
Mac	`Xcode with Clang | CMake/gmake (64-bit Mac)`	`Intel-x86-64 (Mac OS X)` `AMD-x86-64 (Mac OS X)`
	`Xcode with Clang | CMake/Ninja (64-bit Mac)`	
	`Xcode with Clang Project | CMake (64-bit Mac)`	

Note If the **Test hardware is the same as production hardware** check box is not selected, the **Test hardware** settings for device vendor and type control the availability of shipped toolchain definitions.

Generate Only Code and CMake Configuration File for Algorithm Export

To export generated code to another development environment, use this workflow:

1. In the Configuration Parameters dialog box, from the **Toolchain** drop-down list, select `CMake`. For this option, the code generator:
   - Generates C and C++ source code from the Simulink model but does not build the code.
   - Creates a CMake configuration (`CMakeLists.txt`) file that does not depend on specific build tools.
2. Select the **Package code and artifacts** check box.
3. Generate code, which produces a ZIP file that contains the source code and CMake configuration file.
4. Relocate the ZIP file to your development environment.
5. Unzip the packaged files.
6. Use the CMake build tool and configuration file to create an executable file.

If the ZIP file contains `mlrFiles.zip`, static dependencies from your `matlabroot` folder tree, you must specify the location of the unzipped files for the CMake build tool. In your development environment, set the `MATLAB_ROOT` CMake cache entry by running this command:

```
cmake -S pathToSource -B pathToProposedLocationOfDerivedFiles
-DMATLAB_ROOT=pathToUnzippedmlrFiles
```

For more information, see “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24.

See Also

`slbuild` | `packNGo` | `codebuild` | `codebuild`

Related Examples

- “Create Custom CMake Toolchain Definition” on page 59-17

- “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
- “Relocate or Share Generated Code” on page 59-56
- “Compile Code in Another Development Environment” on page 52-99
- “SIL and PIL Simulations” on page 88-2
- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2

External Websites

- https://www.mathworks.com/support/requirements/supported-compilers.html
- https://cmake.org/cmake/help/latest/index.html

Create Custom CMake Toolchain Definition

CMake is a third-party, open-source tool for build process management. To build code generated from Simulink models, the software provides CMake toolchain definitions that you can select from the **Toolchain** drop-down list. You can also create custom CMake toolchain definitions by using the `target` package. When you add a custom CMake toolchain definition to an internal database, the toolchain is available for building generated code. This example shows how you can create a CMake toolchain definition using the Ninja generator. You can use the toolchain definition on computers with these operating systems:

- Windows (provided MATLAB Support for MinGW-w64 C/C++ Compiler is installed).
- Linux
- Mac

Create a `target.Toolchain` object.

```
tc = target.create('Toolchain', 'Name', 'Example Custom CMake Toolchain');
```

Use the `Builder` property and `target.CMakeBuilder` object to configure how CMake builds generated code.

```
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.ToolchainFile = fullfile(pwd, 'ExampleCMakeToolchain.cmake');
tc.Builder.SupportedBuildTypes(end+1) = ...
   target.create('CMakeBuildType', ...
   'Name', 'FastMath', ...
   'GeneratesDebugSymbols', false, ...
   'DebugBuildType', 'FastMathWithDebug');
tc.Builder.SupportedBuildTypes(end+1) = ...
   target.create('CMakeBuildType', ...
   'Name', 'FastMathWithDebug', ...
   'GeneratesDebugSymbols', true);
```

Using an editor, create `ExampleCMakeToolchain.cmake`, a file that contains these lines.

```
# Example CMake toolchain

# Configure to use GCC
if(WIN32)
    list(APPEND CMAKE_PROGRAM_PATH "$ENV{MW_MINGW64_LOC}")
endif()
set(CMAKE_C_COMPILER "gcc")
set(CMAKE_CXX_COMPILER "g++")

# Add some custom build configurations:  FastMath, and FastMathWithDebug
set(CMAKE_C_FLAGS_FASTMATH "-Ofast")
set(CMAKE_CXX_FLAGS_FASTMATH "-Ofast")
set(CMAKE_C_FLAGS_FASTMATHWITHDEBUG  "-Ofast -g")
set(CMAKE_CXX_FLAGS_FASTMATHWITHDEBUG "-Ofast -g")
```

For information about CMake toolchain files, see https://cmake.org/cmake/help/book/mastering-cmake/chapter/Cross%20Compiling%20With%20CMake.html.

Use the first element of the `EnvironmentConfiguration` property to provide system environment setup commands and paths for the Windows operating system.

```
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Linux = false;
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Windows = true;
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Mac = false;
tc.EnvironmentConfiguration(1).SystemPaths = ...
  {'$(MW_MINGW64_LOC)/bin', '$(MATLAB_ROOT)/toolbox/shared/coder/ninja/$(ARCH)'};
```

Use the second element of the `EnvironmentConfiguration` property to provide system environment setup commands and paths for Linux and Mac operating systems.

```
tc.EnvironmentConfiguration(2) = target.create('EnvironmentConfiguration');
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Linux = true;
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Windows = false;
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Mac = true;
tc.EnvironmentConfiguration(2).SystemPaths = ...
  {'$(MATLAB_ROOT)/toolbox/shared/coder/ninja/$(ARCH)'};
```

By default, the software uses the CMake executable file that ships with MATLAB. To specify another version of CMake, you can create a object that provides the required path.

Optionally, associate the toolchain definition with your target devices, for example:

```
windowsProc = target.get('Processor', 'Intel-x86-64 (Windows64)');
tc.SupportedHardware(end+1) = ...
  target.create('HardwareComponentSupport', 'Component', windowsProc);

linuxProc = target.get('Processor', 'Intel-x86-64 (Linux 64)');
tc.SupportedHardware(end+1) = ...
  target.create('HardwareComponentSupport', 'Component', linuxProc);

macProc = target.get('Processor', 'Intel-x86-64 (Mac OS X)');
tc.SupportedHardware(end+1) = ...
  target.create('HardwareComponentSupport', 'Component', macProc);
```

Add the toolchain definition to the internal database:

```
target.add(tc);
```

To use the custom toolchain definition for building generated code, in the Configuration Parameters dialog box:

1. On the **Hardware Implementation** pane, select your target device by setting **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)`.
2. On the **Code Generation** pane, from the **Toolchain** list, select `Example Custom CMake Toolchain`. The **Build configuration** list contains these values:
   - `Release`
   - `Debug`
   - `RelWithDebInfo`
   - `MinSizeRel`
   - `FastMath`
   - `FastMathWithDebug`
   - `Specify`
3. Under **Advanced parameters**, click **Validate Toolchain**, which generates a validation report.

Validation Report for: Example Custom CMake Toolchain

Toolchain Validation Result: Passed

Validation report:

Validating definition of toolchain "Example Custom CMake Toolchain":

Passed.

Validating that the location of the CMake tool is uniquely defined for your development computer:

Passed.

Validation of build tool "CMake"

Configuring test CMakeLists.txt file:

Passed.

Validating that CMake generator "Ninja" is a single-configuration generator:

Passed.

When you run, for example, the `slbuild` function or a software-in-the-loop (SIL) simulation, the build process uses the custom toolchain to build generated code.

If you want to remove the custom toolchain definition from the internal database, run:

```
customToolChainDef = target.get('Toolchain', 'Example Custom CMake Toolchain');
target.remove(customToolChainDef);
```

```
target.remove summary:

    Objects removed from internal database:
        target.Toolchain    "Example Custom CMake Toolchain"
```

See Also

`target.create` | `target.Toolchain` | `target.CMake` | `target.CMakeBuilder` | `target.CMakeBuildType` | `target.CMakeCacheEntry` | `target.EnvironmentConfiguration` | `target.HardwareComponentSupport`

Related Examples

- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Configure CMake Build Process” on page 59-13

External Websites

- https://cmake.org/cmake/help/book/mastering-cmake/chapter/Cross%20Compiling%20With%20CMake.html

Configure Toolchain (ToolchainInfo) or Template Makefile Build Process

Specify the toolchain (`ToolchainInfo`) or template makefile approach for building code generated from a model.

If you set the System target file configuration parameter to, `grt.tlcert.tlc`, `ert_shrlib.tlc`, or another toolchain-compliant system target file, the build process uses the toolchain approach on page 59-20. I you set the parameter to a non-toolchain-compliant system target file, the build process uses the template makefile approach on page 59-24.

You can switch from the toolchain approach to the template makefile approach by using this command:

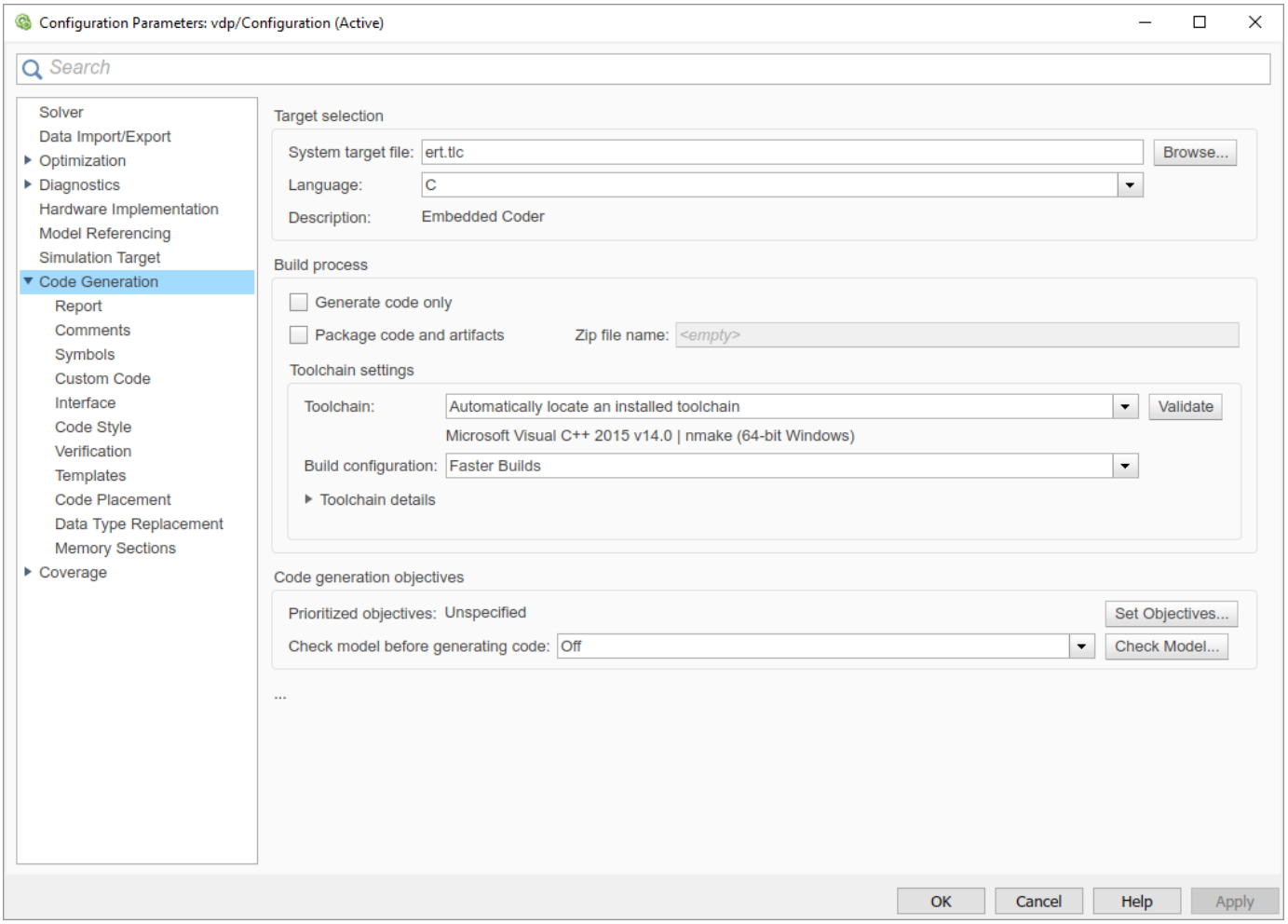
```
set_param(model, 'MakeCommand', 'make_rtw TMF=1')
```

Toolchain Approach

Toolchain settings appear under **Build process** when you set **System target file** to:

- `grt.tlc – Generic Real-Time Target`
- `ert.tlc – Embedded Coder`
- `ert_shrlib.tlc – Embedded Coder (host-based shared library target)`
- A toolchain-compliant system target file

For more information about toolchain-compliant system target files, see “Support Toolchain Approach with Custom Target”.



The **Toolchain settings** include these configuration parameters:

- Toolchain -- Specifies the collection of third-party software tools that builds the generated code. A toolchain can include a compiler, linker, and archiver, and other prebuild or postbuild tools.

  The default value of **Toolchain** is `Automatically locate an installed toolchain`. The **Toolchain** parameter displays name of the located toolchain just below `Automatically locate an installed toolchain`.

  To check that the toolchain is present and validate that the code generator has the information required to use the toolchain, in the **Advanced Parameters** section, click **Validate Toolchain**. The resulting Validation Report gives a pass/fail value for the selected toolchain, and shows issues that require resolution.
- Build configuration -- Specifies the optimization settings. By default, **Build Configuration** is set to `Faster Builds`. You can also select `Faster Runs`, `Debug`, and `Specify`. If you select `Specify`, you can customize the options for the toolchain. When you click **Apply**, the custom toolchain settings apply to the current model.

Note The following system target files, which use the template makefile approach, have the same names but different descriptions from system target files that use the toolchain approach:

- `ert.tlc – Create Visual C/C++ Solution File for Embedded Coder`
- `grt.tlc – Create Visual C/C++ Solution File for Simulink Coder`

To avoid confusion, click **Browse** to select the system target file and look at the description of each file.

Upgrade Model to Use Toolchain Approach

When you open a model created before R2013b that uses the following system target files, the software tries to upgrade the model. The upgrade changes the configuration from using template makefile settings to using the toolchain settings:

- `ert.tlc – Embedded Coder`
- `ert_shrlib.tlc – Embedded Coder (host-based shared library target)`
- `grt.tlc – Generic Real-Time Target`

Note To upgrade models using a custom system target file to use the toolchain approach, see "Support Toolchain Approach with Custom Target" on page 94-59.

Some model configuration parameter values prevent the software from upgrading a model to use toolchain settings. The following instructions show you ways to complete the upgrade process.

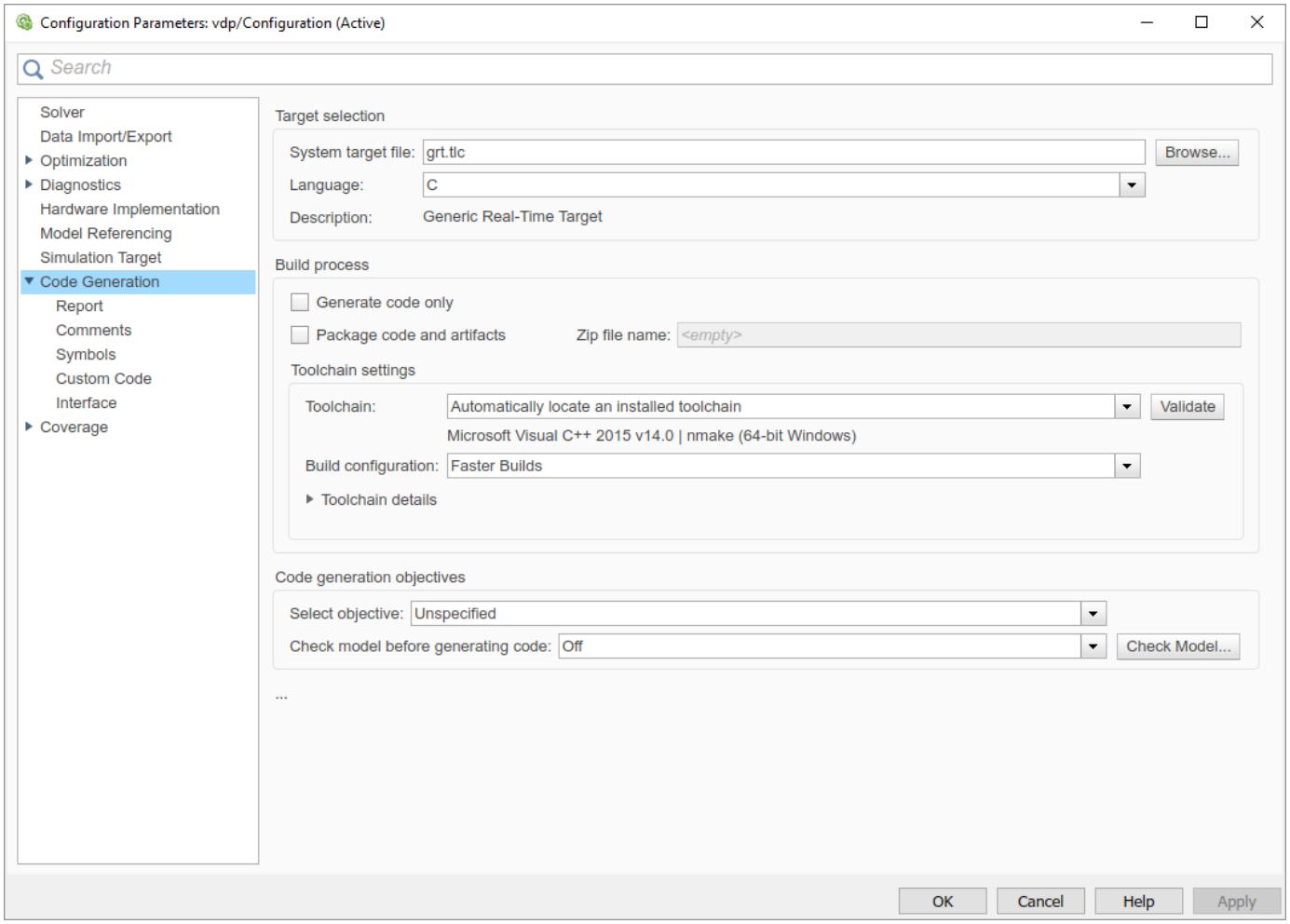
Consider upgrading your models and use the toolchain build approach. Doing so is not required. You can continue generating code from a model that has not been upgraded.

Note The software does not upgrade models that use the following system target files:

- `ert.tlc – Create Visual C/C++ Solution File for Embedded Coder`
- `grt.tlc – Create Visual C/C++ Solution File for Simulink Coder`

To see if a model has been upgraded:

1. If the **Toolchain** and **Build configuration** configuration parameters are specified, the model has been upgraded.



If makefile configuration parameters such as **Generate makefile**, **Make command**, and **Template makefile** are specified, the model has not been upgraded.

Start by creating a working copy of the model using **File** > **Save As**. This action preserves the original model and configuration parameters for reference.

Try to upgrade the model using Upgrade Advisor:

1. On the **Modeling** tab, in the **Evaluate & Manage** section, select **Upgrade Advisor**.
2. In Upgrade Advisor, select **Check and update model to use toolchain approach to build generated code** and click **Run This Check**.
3. Perform the suggested actions and/or click **Update Model**.

When you cannot upgrade the model using Upgrade Advisor, one or more of the following parameters is not set to its default value, shown here:

- **Compiler optimization level** — `Optimizations off (faster builds)`
- **Generate makefile** — Enabled
- **Template makefile** — System target file-specific template makefile

- **Make command** — `make_rtw` without arguments

Sometimes, a model cannot be upgraded. Try the following procedure:

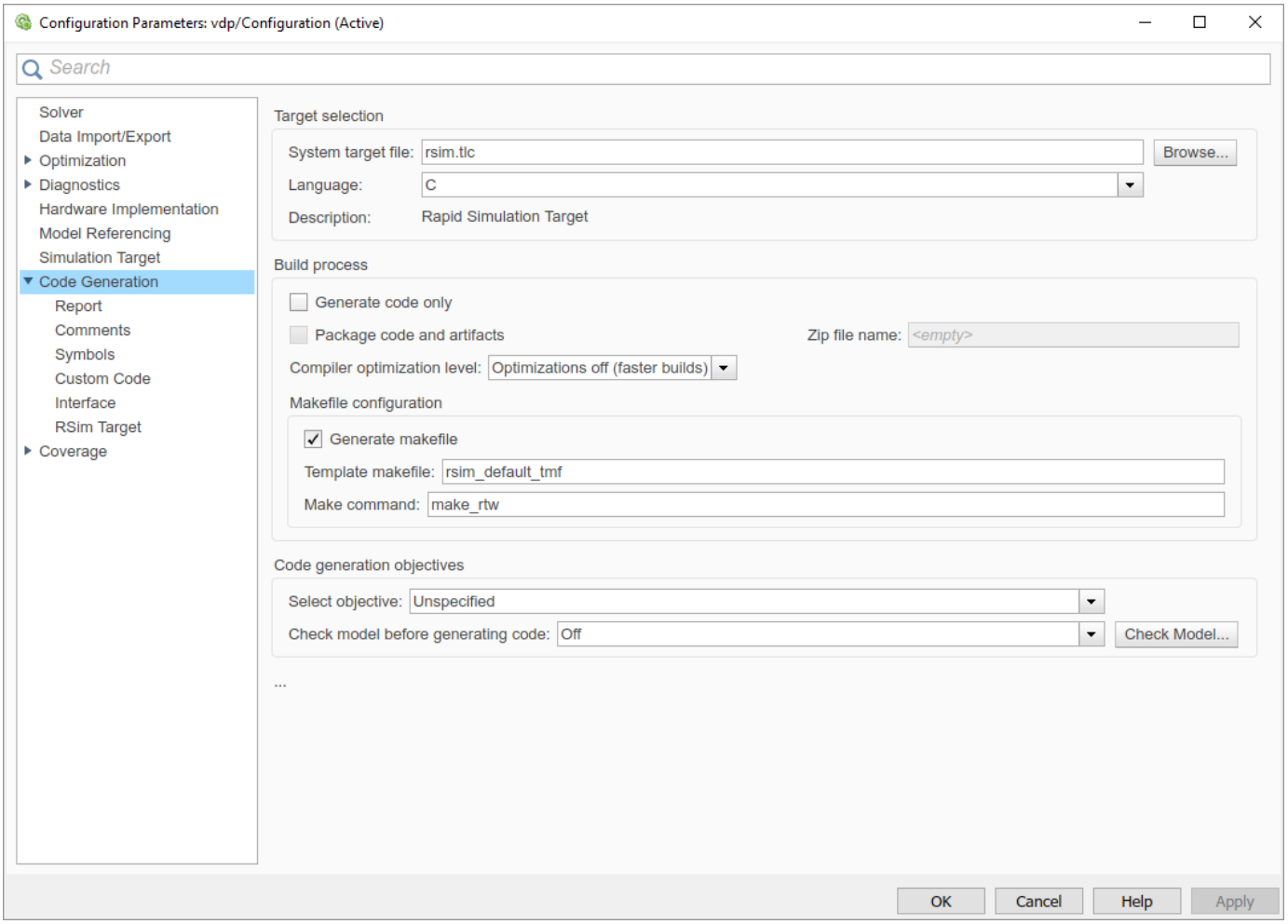
- If **Generate makefile** is disabled, this case cannot be upgradable. However, you can try enabling it and try upgrading the model using Upgrade Advisor.
- If **Compiler optimization level** is set to `Optimizations on (faster runs)`:
  1. Set **Compiler optimization level** is to `Optimizations off (faster builds)`.
  2. Upgrade the model using Upgrade Advisor.
  3. Set **Build configuration** to `Faster Runs`.
- If **Compiler optimization level** is set to `Custom`:
  1. Copy the **Custom compiler optimization flags** to a text file.
  2. Set **Compiler optimization level** to `Optimizations off (faster builds)`.
  3. Upgrade the model using Upgrade Advisor.
  4. Set **Build configuration** to `Specify`.
  5. To perform the same optimizations, edit the compiler options.
- If **Template makefile** uses a customized template makefile, this case cannot be upgradable. However, you can try the following:
  1. Update **Template makefile** to use the default makefile for the system target file.

     Note To get the default makefile name, change the **System target file**, click **Apply**, change it back, and click **Apply** again.
  2. Upgrade the model using Upgrade Advisor.
  3. If the template makefile contains build tool options, such as compiler optimization flags, set **Build configuration** to `Specify` and update the options.
  4. If the template makefile uses custom build tools, create and register a custom toolchain, as described in “Custom Toolchain Registration” . Then, set the **Toolchain** parameter to use the custom toolchain.

     Note After registering the custom toolchain, update **Toolchain** to use the custom toolchain.
  5. If the template makefile contains custom rules and logic, these customizations cannot be applied to the upgraded model.

Template Makefile Approach

When the **System target file** is set to a `tlc` file that uses the template makefile approach, the software displays **Compiler optimization level**, **Generate makefile**, **Make command**, and **Template makefile** parameters.



Specify Whether to Generate a Makefile

The **Generate makefile** option specifies whether the build process is to generate a makefile for a model. By default, the build process generates a makefile. Suppress generation of a makefile, for example in support of custom build processing that is not based on makefiles, by clearing **Generate makefile**. When you clear this parameter:

- The **Make command** and **Template makefile** options are unavailable.
- Set up post code generation build processing using a user-defined command, as explained in “Customize Post-Code-Generation Build Processing”.

Specify a Make Command

Each template makefile-based system target file has an associated `make` command. The code generator uses this internal MATLAB command to control the build process. The command appears in the **Make command** field and runs when you start a build.

Most system target files use the default command, `make_rtw`. Third-party system target files could supply another `make` command. See the documentation from the vendor.

In addition to the name of the `make` command, you can supply makefile options in the **Make command** field. These options could include compiler-specific options, include paths, and other

parameters. When the build process invokes the `make` utility, these options are passed on the `make` command line, which adds them to the overall flags passed to the compiler.

"Template Makefiles and Make Options" on page 59-36 lists the **Make command** options you can use with each supported compiler.

Specify the Template Makefile

The **Template makefile** field has these functions:

- If you selected a system target file with the System Target File Browser, this field displays the name of a MATLAB language file that selects a template makefile for your development environment. For example, in "Model Configuration Parameters: Code Generation", the **Template makefile** field displays `grt_default_tmf`, indicating that the build process invokes `grt_default_tmf.m`.

  "Template Makefiles and Make Options" on page 59-36 gives a detailed description of the logic by which the build process selects a template makefile.
- Alternatively, you can explicitly enter the name of a specific template makefile (including the extension) or a MATLAB language file that returns a template makefile in this field. Use this approach if you are using a system target file that does not appear in the System Target File Browser. For example, use this approach if you have written your own template makefile for a custom system target file.

If you specify your own template makefile, be sure to include the file name extension. If you omit the extension, the build process attempts to find and execute a file with the extension `.m` (that is, a MATLAB language file). The template make file (or a MATLAB language file that returns a template make file) must be on the MATLAB path. To determine whether the file is on the MATLAB path, enter the following command in the MATLAB Command Window:

```
which tmf_filename
```

Associate the Template Makefile with a Toolchain

Specify a toolchain for the template makefile build process. This example provides a definition for a minimal toolchain and shows how you can associate the toolchain with a template makefile that is a copy of `ert_unix.tmf` or `ert_vcx64.tmf`.

1. Use a copy of `ert_unix.tmf` or `ert_vcx64.tmf` as the template makefile.

   ```
   copyfile(fullfile(matlabroot, 'toolbox', 'coder', ...
                     'compile', 'tmf', 'ert_vcx64.tmf'), ...
                     'ert_copy.tmf')
   ```

2. In the template makefile, i.e. `ert_copy.tmf`, edit the `TOOLCHAIN_NAME` macro.

   ```
   TOOLCHAIN_NAME = "Minimal Toolchain for TMF Build"
   ```

   Note The `ert_unix.tmf` and `ert_vcx64.tmf` files from the current release are simplified compared to the corresponding files from releases before R2018a. The software determines whether your template makefile uses the simplified approach by checking for the presence of the variable `TOOLCHAIN_NAME`. A template makefile that is based on `ert_unix.tmf` and `ert_vcx64.tmf` files from releases before R2018a does not specify the `TOOLCHAIN_NAME` variable.

Through the `|>MODEL_MODULES<|` token, the simplified approach specifies the `main`, *`model`*, and other source files that require compiling. The template makefile that is based on `ert_unix.tmf` and `ert_vcx64.tmf` files from releases before R2018a explicitly adds the `main` and *`model`* source files to the list of source files that require compiling.

To update a template makefile that originates from a release before R2018a:

a Remove references to `ert_main.c`, `classic_main.c` or `rt_main.c`. For example, literal references like `ert_main.c` or tokenized references like `ert_main.$(TARGET_LANG_EXT)`.

b Remove references to `$(MODEL).c` or `$(MODEL).$(TARGET_LANG_EXT)`.

---

3 Associate the toolchain with the template make file.

a Create a toolchain definition file. For this example, use this code:

Toolchain Definition File

```
function tc = minimalToolchainForTMF
% Create a toolchain object
tc = coder.make.ToolchainInfo('Name', 'Minimal Toolchain for TMF Build');

% Specify that the linker requires a response file instead of
% including all object file modules on the linker command line
tc.addAttribute('RequiresCommandFile', true);

if ispc
    objExt = '.obj';
else
    objExt = '.o';
end

% Specify source file and object file extension used by the C compiler
tool = tc.getBuildTool('C Compiler');
tool.setFileExtension(  'Source', '.c');
tool.setFileExtension(  'Header', '.h');
tool.setFileExtension(  'Object',   objExt);

% Specify source file and object file extension used by the C++ compiler
tool = tc.getBuildTool('C++ Compiler');
tool.setFileExtension(  'Source', '.cpp');
tool.setFileExtension(  'Header', '.hpp');
tool.setFileExtension(  'Object',   objExt);
```

b Run the toolchain definition file, which generates a `ToolchainInfo` object, `tc`.

```
tc = minimalToolchainForTMF;
```

c Save the `ToolchainInfo` object to a MAT file.

```
save('tcMinimal','tc')
```

d Register the toolchain in `RTW.TargetRegistry`.

i Place this code in an `rtwTargetInfo.m` file.

```
function rtwTargetInfo(tr)
tr.registerTargetInfo(@loc_createToolchain);
end

function config = loc_createToolchain
config(1) = coder.make.ToolchainInfoRegistry;
config(1).Name = 'Minimal Toolchain for TMF Build';
config(1).FileName = fullfile(fileparts(mfilename('fullpath')),...
```

```
    'tcMinimal.mat');
config(1).TargetHWDeviceType = {'*'};
config(1).Platform =  {computer('arch')};
end
```

ii Save the file in the current working folder or in a folder that is on the MATLAB search path.

iii Reset the `TargetRegistry`.

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

4 Associate the template makefile with your model and build the model.

```
set_param(model, 'TemplateMakefile', 'ert_copy.tmf');
slbuild(model);
```

Specify TLC for Code Generation

Target Language Compiler (TLC) is an integral part of the code generator. It enables you to customize generated code. Through customization, you can produce platform-specific code, or you can incorporate your own algorithmic changes for execution speed, code size, or compatibility with existing methods that you prefer to maintain. For additional information, see “Target Language Compiler Overview”.

TLC options that you specify for code generation appear in the summary section of the generated HTML code generation report.

Note Specifying TLC command-line options does not add flags to the make command line.

You can specify Target Language Compiler (TLC) command-line options and arguments for code generation using the model parameter `TLCOptions` in a `set_param` function call. For example,

```
>> set_param(gcs,'TLCOptions','-p0 -aWarnNonSaturatedBlocks=0')
```

Some common uses of TLC options include the following:

- `-aVarName=1` to declare a TLC variable and/or assign a value to it
- `-IC:\Work` to specify an include path
- `-v` to obtain verbose output from TLC processing (for example, when debugging)

Use of Checksums in Recompilation of Generated Code

When you build a model with `GenerateMakefile` set to `'on'` and `GenCodeOnly` set to `'off'`, the software generates a makefile and runs the makefile to compile the generated code. When the compilation is complete, the software creates and saves checksums for source files and configuration data.

If checksums from a previous build are available when you build the model:

1 The software uses the checksums to determine whether the source files and configuration data are different from the previous build.

2 If the software detects a difference, the software:

- Regenerates the makefile.
- Recompiles the generated code.

If the build process uses a template makefile, you can disable the use of checksums by adding this text to the template makefile:

```
DO_NOT_USE_CHECKSUMS=1
```

In this case, the software uses file timestamps to determine whether it should recompile the generated code.

See Also

More About

- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Support Toolchain Approach with Custom Target” on page 94-59
- “Build Process Workflow for Real-Time Systems”
- “Custom Toolchain Registration”
- “Register Custom Toolchain and Build Executable”
- “Template Makefiles and Make Options” on page 59-36
- “Target Language Compiler Overview”
- “Executable Program Generation” on page 59-62
- “Configure CMake Build Process” on page 59-13

Define Custom Makefile-Based Toolchains Using Target Framework

To build code that you generate from Simulink models, you can specify a shipped makefile-based toolchain definition – see “Toolchain Approach” on page 59-20. Using the Target Framework, you can define and register custom makefile-based toolchains. This topic provides toolchain definition examples that you can adapt to build code for your target hardware.

Overview of Makefile-Based Toolchain Definition

To define a toolchain, create a `target.Toolchain` object. Use the object to:

- Associate the toolchain with your target hardware.
- Specify the platforms on which the toolchain can run.
- Provide system environment setup commands and paths for the operating system.

To specify that the toolchain is makefile-based, attach a `target.MakefileBuilder` object to the `Builder` property of the `target.Toolchain` object. The `target.MakefileBuilder` object specifies the type of makefile (GNU Make or NMAKE) that the software generates.

To specify the tools (assembler, compiler, linker, etc) for the toolchain, use `target.BuildTool` objects. Associate each `target.BuildTool` object with a predefined `target.BuildToolType` object that specifies the directives and file types that the `target.BuildTool` object must define. If you specify the name of the `target.BuildToolType` when creating `target.BuildTool`, the software uses default GCC values for directives and file types. If you use a toolchain that is not GCC-based, you must update the prepopulated directives and file-type settings with values for the toolchain.

When you add the `target.Toolchain` object to the internal database, the software uses specifications in the `target.BuildToolType` objects to check that:

- Directive names are valid and the directives have a specification.
- Names for valid file types are associated with file extensions, and the required file extensions are set.

For each `target.BuildTool` object, you can use the `showDirectives` and `showFileExtensions` methods to visualize the build tool configuration.

Define GCC-Based Toolchain for Windows Development Computer

1. Create a `target.Toolchain` object and specify the operating system.

```
mingwtc = target.create('Toolchain', ...
    'Name', 'Example MinGW Toolchain', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
```

2. Place the binary files folder of the MinGW toolchain on the system search path.

```
mingwtc.EnvironmentConfiguration.SystemPaths{end+1} = '$(MW_MINGW64_LOC)/bin';
```

3. Associate the toolchain with the target hardware, which is your Windows development computer.

```
mingwtc.SupportedHardware = target.create('HardwareComponentSupport', ...
    'Component', target.get('Processor', 'Intel-x86-64 (Windows64)'));
```

4. Specify that the toolchain is based on a GNU Make makefile.

```
mingwtc.Builder = target.create('MakefileBuilder', 'GMake');
```

5. Specify the assembler.

```
assembler = target.create('BuildTool', 'Assembler', 'as', ...
    'Name', 'GNU Assembler');
mingwtc.Tools(end+1) = assembler;
```

6. For the Windows operating system, specify C and C++ compilers that use command files and generate object files with the `.obj` extension.

```
cCompiler = target.create('BuildTool', 'C Compiler', 'gcc', ...
    'Name', 'MinGW GCC C Compiler');
cCompiler.setDirective('CommandFile', '@');
cCompiler.setFileExtensions('Object', {'.obj'});
mingwtc.Tools(end+1) = cCompiler;

cppCompiler = target.create('BuildTool', 'C++ Compiler', 'g++', ...
    'Name', 'MinGW GCC C++ Compiler');
cppCompiler.setDirective('CommandFile', '@');
cppCompiler.setFileExtensions('Object', {'.obj'});
mingwtc.Tools(end+1) = cppCompiler;
```

7. Through directives and file extension settings, specify C and C++ linkers that:

   - Support the generation of dynamic link libraries (DLLs)
   - Group libraries that have circular dependencies
   - Use command files
   - Create files for Windows

```
cLinker = target.create('BuildTool', 'Linker', 'gcc', ...
    'Name', 'MinGW Linker', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
cLinker.setDirective('Shared', '-shared -Wl,--out-implib,$(notdir $(basename $(PRODUCT))).lib');
cLinker.setDirective('LibraryGroup', '-Wl,--start-group', '-Wl,--end-group');
cLinker.setDirective('CommandFile', '@');
cLinker.setFileExtensions('Object', {'.obj'});
cLinker.setFileExtensions('Executable', {'.exe'});
cLinker.setFileExtensions('Shared Library', {'.dll'});
mingwtc.Tools(end+1) = cLinker;

cppLinker = target.create('BuildTool', 'Copy', cLinker, ...
    'Name', 'MinGW C++ Linker', ...
    'BuildToolType', target.get('BuildToolType', 'C++ Linker'), ...
    'Command', target.create('Command', 'g++'));
mingwtc.Tools(end+1) = cppLinker;
```

   In this example, you create the `cppLinker` object by creating a copy of `cLinker` and modifying properties of the copy.

8. Specify an archiver.

```
archiver = target.create('BuildTool', 'Archiver', 'ar ruvs', ...
    'Name', 'GNU Archiver');
archiver.setFileExtensions('Object', {'.obj'});
mingwtc.Tools(end+1) = archiver;
```

9. Specify a `make` tool.

```
maketool = target.create('BuildTool', 'Make Tool', 'mingw32-make -j$(NUM_CORES)', ...
    'Name', 'MinGW GNU Make', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
mingwtc.Tools(end+1) = maketool;
```

   The `$(NUM_CORES)` token is expanded to the number of cores on your system, and enables parallel processing of the `make` command.

10 Specify basic system tools like `echo`, `del`, and `move`, which the makefile uses to display, delete, and move files. The predefined `target.Toolset` object contains the tool definitions.

```
basictools = target.get('Toolset', 'Windows system tools for Makefiles');
mingwtc.Tools(end+1) = basictools;
```

11 Specify standard dependencies:

- C math and Winsock libraries to link with the compiled generated code
- Compiler flags that are always passed to the C and C++ compilers

```
mingwtc.BuildRequirements.SharedLibraries{end+1} = 'm';
mingwtc.BuildRequirements.SharedLibraries{end+1} = 'ws2_32';

mingwtc.BuildRequirements.CompilerFlags{end+1} = '-fwrapv';
mingwtc.BuildRequirements.CompilerFlags{end+1} = '-fPIC';

mingwtc.BuildRequirements.LinkerFlags{end+1} = '-static';
mingwtc.BuildRequirements.LinkerFlags{end+1} = '-m64';
```

12 Add the toolchain definition to the internal database.

```
target.add(mingwtc);
```

To use the custom toolchain definition for building generated code, in the Configuration Parameters dialog box:

1 On the **Hardware Implementation** pane, select your target device by setting **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)`.

2 On the **Code Generation** pane, from the **Toolchain** list, select `Example MinGW Toolchain`.

3 Click **OK**.

When you run, for example, the `slbuild` function or a software-in-the-loop (SIL) simulation, the build process uses the custom toolchain to build generated code.

If you want to remove the custom toolchain definition from the internal database, run:

```
customToolChainDef = target.get('Toolchain', 'Example MinGW Toolchain');
target.remove(customToolChainDef);
```

Define Microsoft Visual Studio Toolchain for Building 32-Bit Application

1 Create a toolchain object and specify the operating system.

```
msvctc = target.create('Toolchain', ...
    'Name', '32-bit Visual Studio Toolchain', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
```

2 Create a command that runs `vcvarsall.bat`, which is required to set up environment variables for Microsoft Visual C++. This command assumes that Microsoft Visual Studio 2019 is installed on your development computer. If another version is installed, provide the full path to `vcvarsall.bat`.

```
msvctc.EnvironmentConfiguration.SetupCommand = target.create('Command', ...
    '$(MSVC160_INSTALL_DIR)\VC\Auxiliary\Build\VCVARSALL.BAT x86');
```

3 Associate the toolchain with the target hardware, which is your Windows development computer.

```
msvctc.SupportedHardware = target.create('HardwareComponentSupport', ...
    'Component', target.get('Processor', 'Intel-x86-32 (Windows32)'));
```

4 Specify that the toolchain is based on an NMAKE makefile.

```
msvctc.Builder = target.create('MakefileBuilder', 'NMake');
```

5 Using `target.BuildTool` objects, specify C and C++ compilers:

- Update default GCC directives with Microsoft Visual C++ values.
- Use command files and generate object files with the `.obj` extension.
- Specify additional command-line flags that only apply to the C++ compiler.

```
cCompiler = target.create('BuildTool', 'C Compiler', 'cl', ...
    'Name', 'Microsoft Visual C Compiler', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
cCompiler.setDirective('OutputFlag', '-Fo');
cCompiler.setDirective('Debug', '-Zi');
cCompiler.setDirective('EnableOptimization', '/O2 /Oy-');
cCompiler.setDirective('DisableOptimization', '/Od /Oy-');
cCompiler.setDirective('CommandFile', '@');
cCompiler.setFileExtensions('Object', {'.obj'});
msvctc.Tools(end+1) = cCompiler;
cppCompiler = target.create('BuildTool', 'C++ Compiler', 'cl -EHs /wd4251 /Zc:__cplusplus', ...
    'Name', 'Microsoft Visual C++ Compiler', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
cppCompiler.setDirective('OutputFlag', '-Fo');
cppCompiler.setDirective('Debug', '-Zi');
cppCompiler.setDirective('EnableOptimization', '/O2 /Oy-');
cppCompiler.setDirective('DisableOptimization', '/Od /Oy-');
cppCompiler.setDirective('CommandFile', '@');
cppCompiler.setFileExtensions('Object', {'.obj'});
msvctc.Tools(end+1) = cppCompiler;
```

6 Through directives and file extension settings, specify C and C++ linkers.

```
cLinker = target.create('BuildTool', 'Linker', 'link', ...
    'Name', 'Microsoft Visual C Linker', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
cLinker.setDirective('Library', '-L');
cLinker.setDirective('LibrarySearchPath', '/LIBPATH');
cLinker.setDirective('OutputFlag', '-out:');
cLinker.setDirective('Debug', '/DEBUG');
cLinker.setDirective('Shared', '-dll'); % Allow shared library
cLinker.setDirective('DefFile', '-def:');
cLinker.setDirective('CommandFile', '@');
cLinker.setFileExtensions('Object', {'.obj'});
cLinker.setFileExtensions('Executable', {'.exe'});
cLinker.setFileExtensions('Shared Library', {'.dll'});
msvctc.Tools(end+1) = cLinker;

cppLinker = target.create('BuildTool', 'C++ Linker', 'link', ...
    'Name', 'Microsoft Visual C++ Linker', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
cppLinker.setDirective('Library', '-L');
cppLinker.setDirective('LibrarySearchPath', '/LIBPATH');
cppLinker.setDirective('OutputFlag', '-out:');
cppLinker.setDirective('Debug', '/DEBUG /DEBUGTYPE:cv');
cppLinker.setDirective('Shared', '-dll'); % Allow shared library
cppLinker.setDirective('DefFile', '-def:');
cppLinker.setDirective('CommandFile', '@');
cppLinker.setFileExtensions('Object', {'.obj'});
cppLinker.setFileExtensions('Executable', {'.exe'});
cppLinker.setFileExtensions('Shared Library', {'.dll'});
msvctc.Tools(end+1) = cppLinker;
```

7 Specify an archiver.

```
archiver = target.create('BuildTool', 'Archiver', 'lib /nologo', ...
    'Name', 'Microsoft Visual C/C++ Archiver');
archiver.setDirective('OutputFlag', '-out:');
archiver.setFileExtensions('Object', {'.obj'});
archiver.setFileExtensions('Static Library', {'.lib'});
msvctc.Tools(end+1) = archiver;
```

8 Specify a make tool.

```
maketool = target.create('BuildTool', 'Make Tool', 'nmake', ...
    'Name', 'Microsoft NMake', ...
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);
msvctc.Tools(end+1) = maketool;
```

9 Specify basic system tools like `echo`, `del`, and `move`, which the makefile uses to display, delete, and move files. The predefined `target.Toolset` object contains the tool definitions.

```
basictools = target.get('Toolset', 'Windows system tools for Makefiles');
msvctc.Tools(end+1) = basictools;
```

10 Specify standard dependencies:

- Libraries to link with the compiled generated code
- Preprocessor directives that must be set
- Compiler flags that are always passed to the C and C++ compilers
- Linker flags

```
msvctc.BuildRequirements.SharedLibraries{end+1} = 'kernel32.lib';
msvctc.BuildRequirements.SharedLibraries{end+1} = 'ws2_32.lib';
msvctc.BuildRequirements.SharedLibraries{end+1} = 'mswsock.lib';
msvctc.BuildRequirements.SharedLibraries{end+1} = 'advapi32.lib';

msvctc.BuildRequirements.Defines{end+1} = '_CRT_SECURE_NO_WARNINGS';
msvctc.BuildRequirements.Defines{end+1} = 'WIN32';
msvctc.BuildRequirements.Defines{end+1} = '_MT';

msvctc.BuildRequirements.CompilerFlags{end+1} = '-nologo';
msvctc.BuildRequirements.CompilerFlags{end+1} = '-GS';
msvctc.BuildRequirements.CompilerFlags{end+1} = '-W4';
msvctc.BuildRequirements.CompilerFlags{end+1} = '-MT';

msvctc.BuildRequirements.LinkerFlags{end+1} = '/INCREMENTAL:NO';
msvctc.BuildRequirements.LinkerFlags{end+1} = '/NOLOGO';
```

11 Add the toolchain definition to the internal database.

```
target.add(msvctc);
```

To use the custom toolchain definition for building generated code, in the Configuration Parameters dialog box:

1 On the **Hardware Implementation** pane, select your target device by setting **Device vendor** to `Intel` and **Device type** to `x86-32 (Windows32)`.

2 On the **Code Generation** pane, from the **Toolchain** list, select `32-bit Visual Studio Toolchain`.

3 Click OK.

When you run, for example, the `slbuild` function or a software-in-the-loop (SIL) simulation, the build process uses the custom toolchain to build generated code.

If you want to remove the custom toolchain definition from the internal database, run:

```
customToolChainDef = target.get('Toolchain', '32-bit Visual Studio Toolchain');
target.remove(customToolChainDef);
```

See Also

`target.create` | `target.Toolchain` | `target.MakefileBuilder` | `target.BuildTool`

Related Examples

- “Approaches for Building Code Generated from Simulink Models” on page 59-11

Template Makefiles and Make Options

The code generator includes a set of built-in template makefiles that build programs for specific system target files.

Types of Template Makefiles

There are two types of template makefiles:

- *Compiler-specific* template makefiles are for a particular compiler or development system.

  By convention, compiler-specific template makefiles names correspond to the system target file and compiler (or development system). For example, `grt_vcx64.tmf` is the template makefile for building a generic real-time program under the Visual C++ compiler; `ert_lcc64.tmf` is the template makefile for building an Embedded Coder program under the `lcc` compiler.
- *Default* template makefiles make your model designs more portable, by choosing the compiler-specific makefile and compiler for your installation. “Select and Configure C or C++ Compiler” on page 59-3 describes the operation of default template makefiles in detail.

  Default template makefiles have names that follow the pattern *`target`*`_default_tmf`. They are MATLAB language files that, when run, select the TMF for the specified system target file configuration. For example, `grt_default_tmf` is the default template makefile for building a generic real-time program; `ert_default_tmf` is the default template makefile for building an Embedded Coder program.

For details on the structure of template makefiles, see “Customize Template Makefiles” on page 94-44. This section describes compiler-specific template makefiles and common options you can use with each.

Specify Template Makefile Options

You can specify template makefile options through the **Make command** configuration parameter. Append the options after `make_rtw` (or other `make` command), as in the following example:

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

The syntax for `make` command options differs slightly for different compilers.

Note To control compiler optimizations for a makefile build, use the **Compiler optimization level** configuration parameter, which provides:

- System target file-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)`, which easily allow you to toggle compiler optimizations on and off during code development
- The value `Custom` for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Template Makefiles for UNIX Platforms

The template makefiles for UNIX platforms are for the Free Software Foundation's GNU Make. These makefiles conform to the guidelines specified in the IEEE[7] Std 1003.2-1992 (POSIX) standard:

- `ert_unix.tmf`
- `grt_unix.tmf`

You can supply options to makefiles through the **Make command** configuration parameter. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,

  ```
  OPTS="-DMYDEFINE=1"
  ```
- `OPT_OPTS`— Optimization options. Default is `-O`. To enable debugging, specify the option as `OPT_OPTS=-g`. Because of optimization problems in IBM_RS, the default is to not optimize.
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```
- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the Microsoft Visual C++ Compiler

To build an executable file using the Visual C++ compiler, use one of the *`target`*`_vcx64.tmf` template makefiles:

- `ert_vcx64.tmf`
- `grt_vcx64.tmf`

You can supply options to makefiles through the **Make command** configuration parameter. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPT_OPTS` — Optimization option. Default is `-O2`. To enable debugging, specify the option as `OPT_OPTS=-Zi`.
- `OPTS` — User-specific options. For example:

  ```
  OPTS="/D MYDEFINE=1"
  ```
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.

7 IEEE is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

- `USER_INCLUDES` — Additional include paths, for example,

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```
- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the LCC Compiler

The code generator provides template makefiles to create an executable for the Windows platform using Lcc compiler Version 2.4 and GNU Make (`gmake`).

- `ert_lcc64.tmf`
- `grt_lcc64.tmf`

You can supply options to makefiles by using the **Make command** configuration parameter. Options specified in the **Make command** field are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,

  ```
  OPTS="-DMYDEFINE=1"
  ```
- `OPT_OPTS` — Optimization options. Default is to not use options. To enable debugging, specify `-g4`:

  ```
  OPT_OPTS="-g4"
  ```
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths. For example:

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```

  For `lcc`, use `/` as file separator before the file name instead of `\`, for example, `d:\work\proj1/myfile.c`.
- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

See Also

More About

- “Configure a System Target File” on page 50-2
- “Customize System Target Files”
- “Associate the Template Makefile with a Toolchain” on page 59-26

Build Process Workflow for Real-Time Systems

The building process includes generating code in C or C++ from a model and building an executable program from the generated code. This example can use a generic real-time (GRT) or an embedded real-time (ERT) system target file (STF) for code generation. The resulting standalone program runs on your development computer, independent of external timing and events.

Working Folder

This example uses a local copy of the `slexAircraftExample` model, stored in its own folder, `aircraftexample`. Set up your *working folder* as follows:

1. In the MATLAB Current Folder browser, navigate to a folder to which you have write access.
2. To create the working folder, enter the following MATLAB command:

   ```
   mkdir aircraftexample
   ```

3. Make `aircraftexample` your working folder:

   ```
   cd aircraftexample
   ```

4. Open the `slexAircraftExample` model:

   ```
   openExample('slexAircraftExample')
   ```

   The model appears in the Simulink Editor model window.
5. In the model window, choose **File** > **Save As**. Navigate to your working folder, `aircraftexample`. Save a copy of the `slexAircraftExample` model as `myAircraftExample`.

Build Folder and Code Generation Folders

While producing code, the code generator creates a *build folder* within your working folder. The build folder name is *`model_target`*`_rtw`, derived from the name of the source model and the chosen system target file. The build folder stores generated source code and other files created during the build process. Examine the build folder contents at the end of this example.

When a model contains Model blocks (references to other models), the model build creates special subfolders in your “Code generation folder” to organize code for the referenced models. These code generation folders exist alongside product build folders and are named `slprj`. For more information, see “Generate Code for Model Reference Hierarchy”.

Under the `slprj` folder, a subfolder named `_sharedutils` contains generated code that can be shared between models.

Set Model Parameters for Code Generation

To generate code from your model, you must change some of the model configuration parameters. In particular, the generic real-time (GRT) system target file and most other system target files require that the model specifies a fixed-step solver.

Note For models that specify variable-step solvers, the code generator produces code only if the models also specify rapid simulation (`rsim`) or S-function system target files.

1. Open the `myAircraftExample` model if it is not already open.
2. In the Configuration Parameters dialog box, specify configuration parameter values for the solver:
   - **Start time**: `0.0`
   - **Stop time**: `60`
   - **Type**: `Fixed-step`
   - **Solver**: `ode5 (Dormand-Prince)`
   - **Fixed step size (fundamental sample time)**: `0.1`
   - **Treat each discrete rate as a separate task**: `Off`
3. Click **Apply**.
4. Save the model.

Configure Build Process

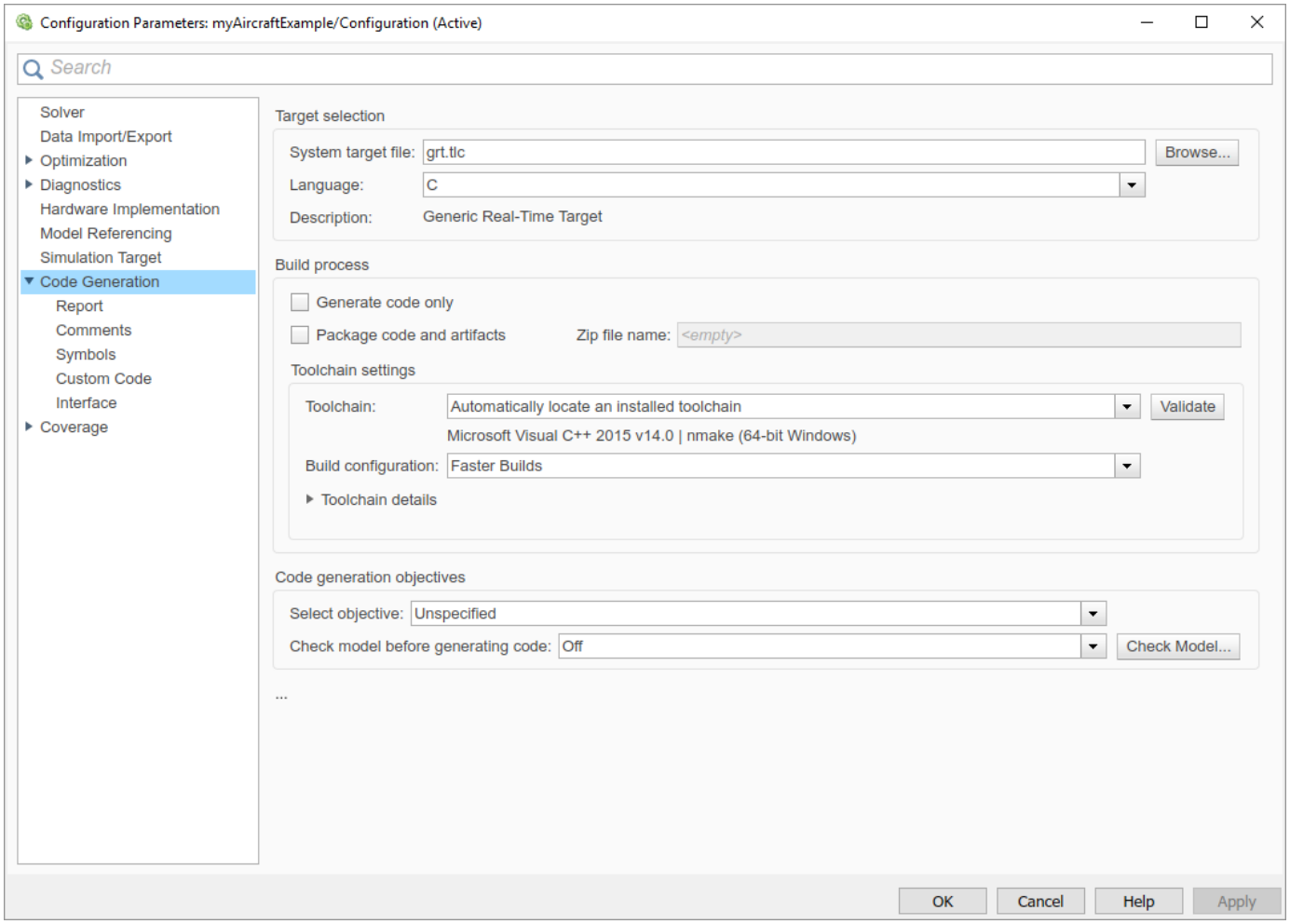
To configure the build process for your model, choose a system target file, a toolchain or template makefile, and a `make` command.

In these examples and in most applications, you do not need to specify these parameters individually. The examples use the ready-to-run generic real-time target (GRT) configuration. The GRT system target file builds a standalone executable program that runs on your desktop computer.

To select the GRT system target file:

1. Open the `myAircraftExample` model if it is not already open.
2. In the Configuration Parameters dialog box, in the **System target file** field, enter `grt.tlc`. Then click **Apply**.

   You see selections for **Toolchain** (`Automatically locate an installed toolchain`), and **Build Configuration** (`Faster Builds`).



3 Save the model.

Note If you click **Browse**, a System Target File Browser opens and displays the system target files on the MATLAB path. Some system target files require additional products. For example, `ert.tlc` requires Embedded Coder.

Set Code Generation Parameters

1 Open the `myAircraftExample` model if it is not already open.

2 In the Configuration Parameters dialog box, specify settings:

   a Use the default settings for the **Code Generation** advanced parameters, which control build verbosity and debugging:

   - **Verbose build** (`RTWVerbose`)
   - **Retain .rtw file** (`RetainRTWFile`)
   - **Profile TLC** (`ProfileTLC`)
   - **Start TLC debugger when generating code** (`TLCDebug`)

- **Start TLC coverage when generating code** (`TLCCoverage`)
- **Enable TLC assertion** (`TLCAssert`)

b Use the default **Code Generation** > **Comments** settings.

c The **Code Generation** > **Identifiers** options control the look and feel of generated code. Use the default settings.

d Select **Code Generation** > **Interface**.

i From the **Shared code placement** list, select `Shared location`. The build process places generated code for utilities in a subfolder within your "Code generation folder".

ii Under the **Advanced parameters**, clear the **Classic call interface** check box.

iii Under the **Advanced parameters**, select the **Single output/update function** check box.

e In **Code Generation** > **Report**, select **Create code generation report** and **Open report automatically**. This action enables the software to create and display a code generation report for the `myAircraftExample` model.

3 Click **Apply** and save the model.

Build and Run a Program

The build process generates C code from the model. It then compiles and links the generated program to create an executable image. To build and run the program:

1 With the `myAircraftExample` model open, perform one of these actions:

- On the **Apps** tab, open the **Simulink Coder** app. In the **C Code** tab, click **Build**.
- Press **Ctrl+B**.
- Run the `slbuild` command from the MATLAB command line.

You see code generation and compilation messages in the Command Window. The initial message is:

```
### Starting build procedure for model: myAircraftExample
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final messages include:

```
### Created executable myAircraftExample.exe
### Successful completion of build procedure for model: myAircraftExample
### Creating HTML report file index.html
```

The code generation folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the build process has created an `slprj` folder and a `myAircraftExample_grt_rtw` folder in your "Code generation folder".

Note After generating the code for the `myAircraftExample` model, the build process displays a code generation report. See "Report Generation" for more information about how to create and use a code generation report.

2 To see the contents of the working folder after the build, enter the `dir` or `ls` command:

```
>> dir

.                              myAircraftExample.slx           slprj
..                             myAircraftExample.slx.autosave
myAircraftExample.exe          myAircraftExample_grt_rtw
```

3 To run the executable from the Command Window, type `!myAircraftExample`. The `!` character passes the command that follows it to the operating system, which runs the standalone `myAircraftExample` program.

```
>> !myAircraftExample

** starting the model **
** created myAircraftExample.mat **
```

4 To see the files created in the build folder, use the `dir` or `ls` command again. The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform:

```
>> dir myAircraftExample_grt_rtw

.                     rt_main.obj                  myAircraftExample_data.c
..                    rtmodel.h                    myAircraftExample_data.obj
buildInfo.mat         rtw_proj.tmw                 myAircraftExample_private.h
codeInfo.mat          myAircraftExample.bat        myAircraftExample_ref.rsp
myAircraftExample.c   myAircraftExample_types.h    html
myAircraftExample.h   myAircraftExample.mk
rt_logging.obj        myAircraftExample.obj
```

Contents of the Build Folder

The build process creates a build folder and names it *`model_target`*`_rtw`, where *`model`* is the name of the source model and *`target`* is the system target file selected for the model. In this example, the build folder is named `myAircraftExample_grt_rtw`.

The build folder includes the following generated files.

File	Description
`myAircraftExample.c`	Standalone C code that implements the model
`myAircraftExample.h`	An include header file containing definitions of parameters and state variables
`myAircraftExample_private.h`	Header file containing common include definitions
`myAircraftExample_types.h`	Forward declarations of data types used in the code
`rtmodel.h`	Header file for including generated code in the static main program (its name does not change, and it simply includes `myAircraftExample.h`)

The code generation report that you created for the `myAircraftExample` model displays a link for each of these files. You can click the link explore the file contents.

The build folder contains other files used in the build process. They include:

- `myAircraftExample.mk` — Makefile for building executable using the specified Toolchain.
- Object (`.obj`) files
- `myAircraftExample.bat` — Batch control file
- `rtw_proj.tmw` — Marker file

- `buildInfo.mat` — Build information for relocating generated code to another development environment
- `myAircraftExample_ref.rsp` — Data to include as command-line arguments to `mex` (Windows systems only)

The build folder also contains a subfolder, `html`, which contains the files that make up the code generation report. For more information, see "Reports for Code Generation".

Customized Makefile Generation

After producing code, the code generator produces a customized makefile, *`model`*`.mk`. The generated makefile instructs the `make` system utility to compile and link source code generated from the model, any required harness program, libraries, or user-provided modules. The code generator produces the file *`model`*`.mk` regardless of the approach that you use for build process control:

- If you use the toolchain approach, the code generator creates *`model`*`.mk` based on the model **Toolchain settings**. You can modify generation of the makefile through the `rtwmakecfg.m` API.
- If you use the template makefile approach, the code generator creates *`model`*`.mk` from a system template file, *`system`*`.tmf` (where *`system`* stands for the selected system target file name). The system template makefile is designed for your system target file. You can modify the template makefile to specify compilers, compiler options, and additional information for the creation of the executable.

For more information, see "Configure Toolchain (ToolchainInfo) or Template Makefile Build Process" on page 59-20.

See Also

More About

- "Configure Toolchain (ToolchainInfo) or Template Makefile Build Process" on page 59-20
- "Manage Build Process Folders"
- "Reports for Code Generation"
- "Select and Configure C or C++ Compiler" on page 59-3

Build Models from a Windows Command Prompt Window

This example shows how to build models by using a batch file, entering commands at the Command Prompt in Windows®.

About MATLAB Command-Line (Start Up) Arguments

When you start MATLAB® from a Command Prompt in Windows (as done in a batch file), you can control MATLAB start up with a number of command-line arguments.

For a description of these command-line arguments, in the Command Prompt window, type `matlab -help`.

To start MATLAB from a Command Prompt window, use these steps:

1. From the Windows Start menu, open a Command Prompt window.
2. From the Windows Command Prompt, type: `matlab`.

Tip: To display the path to the MATLAB root folder, at the MATLAB command prompt type: `matlabroot`.

Run MATLAB with a Batch File

When you run MATLAB with a batch file, you can:

- Control MATLAB start up with command-line arguments
- Run a series of operating system commands (such as source control checkout/commit)
- Run a series of MATLAB scripts

A batch approach also lets you automate your overall build process. Such a process can generate code from one or more Simulink® models, then use your makefile to compile custom code and generated code.

This batch file sets the `MATLABROOT` environment variable, sets the `PATH` environment variable to include `MATLABROOT`, and starts MATLAB with an input script argument `%1` and a logfile argument.

Note: Customize the `MATLABROOT` value in the batch file to match your system. The batch file assumes that a `c:\temp` folder exists on your system.

Create a batch file named `mat.bat`

```
SET MATLABROOT="C:\Program Files\MATLAB\R2019a"
PATH=%MATLABROOT%;%PATH%
START matlab.exe -batch %1 -logfile c:\temp\logfile
PAUSE
```

Create a MATLAB script `myFilesToBuild.m`

```
my_rtwdemo_counter_builder
my_rtwdemo_rtwintro_builder
exit
```

Create a MATLAB script `my_rtwdemo_counter_builder.m`

```
open_system('rtwdemo_counter');
save_system('rtwdemo_counter','my_rtwdemo_counter')
slbuild('my_rtwdemo_counter');
close_system('my_rtwdemo_counter');
```

Create a MATLAB script `my_rtwdemo_rtwintro_builder.m`

```
open_system('rtwdemo_rtwintro');
save_system('rtwdemo_rtwintro','my_rtwdemo_rtwintro')
slbuild('my_rtwdemo_rtwintro');
close_system('my_rtwdemo_rtwintro');
```

Run the batch file

From the Windows Start menu, open a Command Prompt window, change folders to the folder containing the batch file, and type:

```
mat myFilesToBuild
```

When you run the batch file with the input MATLAB script, the batch file runs MATLAB and loads, builds, and closes each of the example Simulink models.

Observe the log of MATLAB operations

After the batch file runs, view the `c:\temp\logfile` file.

Omitting the semicolon (;) from the sl`build` line in each script provides more build information in the log file.

Optimize Your Batch File

Use the MATLAB command-line arguments to optimize the batch file. For example:

- Suppress the MATLAB splash screen on startup with the `-nosplash` argument.
- Provide command-line input to the input script or function selected with the `-batch` argument.

You can call a function `myfile.m`, which accepts two arguments:

```
matlab -batch myfile(arg1,arg2)
```

To pass numeric values into `myfile.m`, replace `arg1` and `arg2` with numeric values.

To pass string or character values into `myfile.m`, replace `arg1` and `arg2` with the string or character values surrounded in single quotes. For example, to pass the string values `hello` and `world` into `myfile.m`, in the Command Prompt window, type:

```
matlab -batch myfile('hello','world')
```

Copyright 2007-2019 The MathWorks, Inc.

See Also

More About

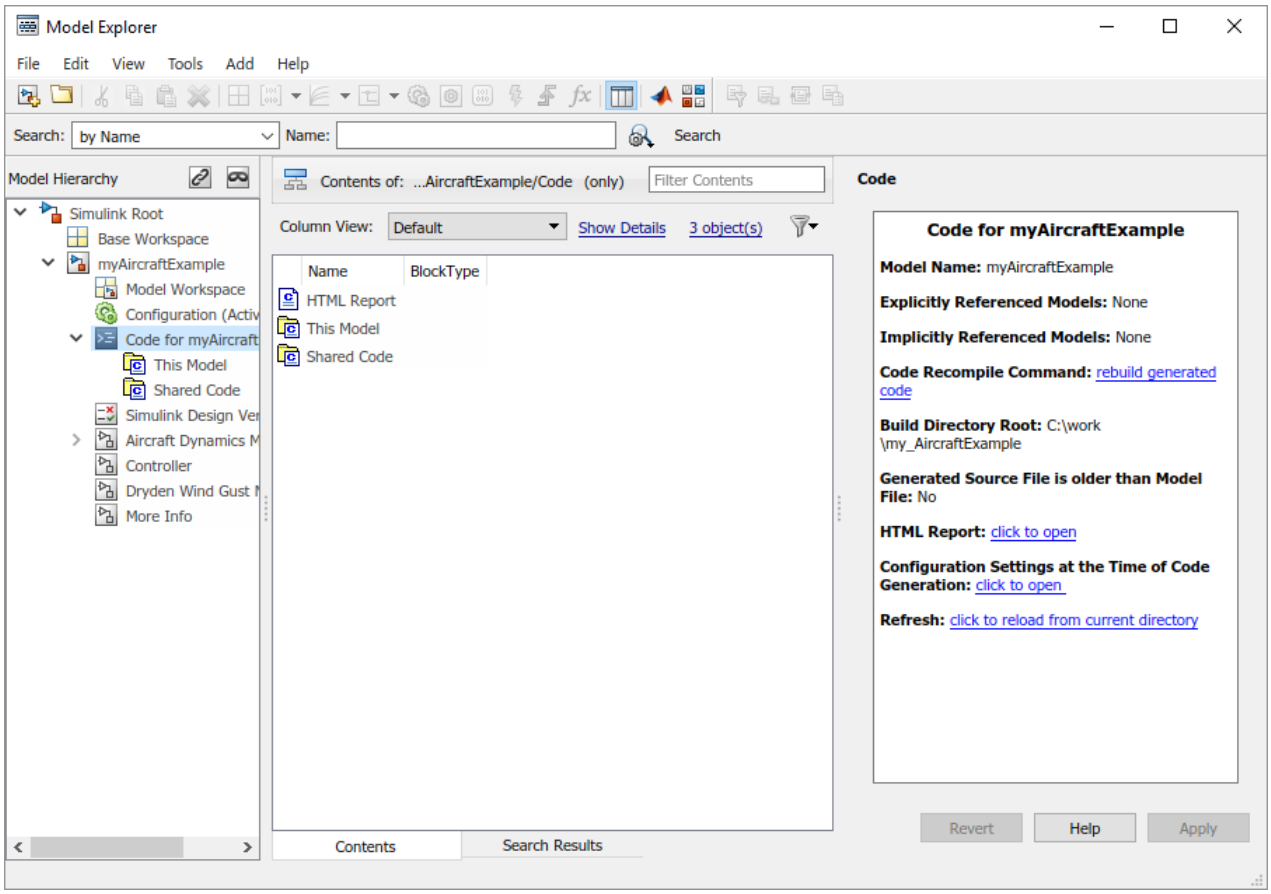
- matlab (Windows)

- “Compile and Debug Generated C Code with Microsoft Visual Studio”

Rebuild a Model

If you update generated source code or makefiles manually to add customizations, you can rebuild the files with the `rtwrebuild` command. This command recompiles the modified files by invoking the generated makefile. Alternatively, you can rebuild from the Model Explorer:

1. Open the Model Explorer
2. In the **Model Hierarchy** pane, expand the node for the model.
3. Click the **Code for *model*** node.
4. In the **Code** pane, next to **Code Recompile Command**, click **rebuild generated code**.



See Also

More About

- `rtwrebuild`
- “Build Process Workflow for Real-Time Systems”

Control Regeneration of Top Model Code

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. The model build regenerates top model code if any of the following conditions is true:

- The structural checksum of the model has changed.
- The top-model-only checksum has changed. The top-model-only checksum provides information about top model parameters, such as application lifespan, maximum stack size, make command, verbose and `.rtw` file debug settings, and `TLCOptions`.
- Any of the following TLC debugging configuration parameters is selected:
  - **Start TLC debugger when generating code** (`TLCDebug`)
  - **Start TLC coverage when generating code** (`TLCCoverage`)
  - **Enable TLC assertion** (`TLCAssert`)
  - **Profile TLC** (`ProfileTLC`)

Whether the top model code is regenerated, the build process calls the build process hooks and reruns the makefile. The hooks include the *`STF`*`_make_rtw_hook` functions and the post code generation command. This process recompiles and links the external dependencies.

System target file authors can perform actions related to code regeneration in the *`STF`*`_make_rtw_hook` functions that the build process calls. These actions include forcing or reacting to code regeneration. For more information, see “Control Code Regeneration Using STF_make_rtw_hook.m” on page 93-27.

Regeneration of Top Model Code

If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code. An example check is whether previously generated code is not current due to a model update.

The build process omits regeneration of the top model code when the checks indicate both:

- The top model generated code is current for the model.
- Model settings do not require full regeneration.

This omission can significantly reduce model build times.

With an Embedded Coder license, if you modify a code generation template (CGT) file then rebuild your model, the code generation process does not force a top model build. In this case, see “Force Regeneration of Top Model Code” on page 59-49.

Force Regeneration of Top Model Code

If you want to control or override the default top model build behavior, use one of the following command-line options:

- To ignore the checksum and force regeneration of the top model code:

  - `slbuild(model,'ForceTopModelBuild',true)`
  - `slbuild(model,'StandaloneCoderTarget','ForceTopModelBuild',true)`
- To clean the model build area enough to trigger regeneration of the top model code at the next build (`slbuild` only):
  `slbuild(model,'CleanTopModel')`

You can also force regeneration of the top model code by deleting the `slprj` folder in the "Code generation folder". If you delete the `slprj` folder, delete the *`model`*`.slxc` file too.

See Also

`codebuild`

More About

- "Control Code Regeneration Using STF_make_rtw_hook.m" on page 93-27
- "Approaches for Building Code Generated from Simulink Models" on page 59-11
- "Rebuild a Model" on page 59-48
- "Reduce Build Time for Referenced Models by Using Parallel Builds" on page 59-51

Reduce Build Time for Referenced Models by Using Parallel Builds

For models that contain large model reference hierarchies, you can reduce code generation and compilation time by building the referenced models in parallel. With Parallel Computing Toolbox software, you can distribute the code generation and compilation for referenced models across a parallel pool of MATLAB workers. If you also have MATLAB Parallel Server™ software, you can distribute the code generation and compilation across remote workers in your MATLAB Parallel Server configuration.

Build Referenced Models in Parallel

To build referenced models in parallel:

1. Open the Configuration Parameters dialog box for the top model of the model hierarchy.
2. Select the **Enable parallel model reference builds** check box.
3. For each MATLAB worker, you can set up a MATLAB environment that is consistent with the MATLAB environment of the client. From the **MATLAB worker initialization for builds** drop-down list, select one of these values:
   - `None` –– Simulink does not initialize workers.
   - `Copy base workspace` –– Simulink attempts to copy the base workspace to each MATLAB worker.
   - `Load top model` –– Simulink loads the top model onto each MATLAB worker.
4. Build your model from the command line or the Simulink Editor Code perspective:
   - Command line –– Use, for example, `slbuild`. The build process displays build log messages in the Command Window.
   - Simulink Editor Code perspective –– Click the **Build** button. The build process displays build log messages in the Diagnostic Viewer.

     If the Parallel Computing Toolbox parallel preference **Automatically create a parallel pool** is selected, a parallel pool of MATLAB workers is automatically started. For more information, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox).

     If a parallel pool of MATLAB workers is not running when you build your model, MATLAB automatically opens a parallel pool of workers by using the default cluster profile. To change the default behaviour of the worker cluster, you can modify properties of the cluster profile. If you have not touched your parallel preferences, the default profile is `local`. Control parallel behavior with the parallel preferences, including scaling up to a cluster, automatic pool creation, and preferred number of workers. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

     For more general information about parallel computing, see “Run MATLAB Functions with Automatic Parallel Support” (Parallel Computing Toolbox).

Monitor Parallel Building of Referenced Models

Through the Build Status window, you can:

- View the overall build progress for a model hierarchy.
- View the build status of individual models in the model hierarchy.
- Cancel the parallel build process.

Note The Build Status window supports only parallel building of model hierarchies. Do not use the Build Status window for serial builds.

To open the Build Status window, use one of these line commands:

- `coder.buildstatus.open('model')`
- `slbuild('model', ... 'OpenBuildStatusAutomatically',true)`
- `slbuild('model','StandaloneCoderTarget', ... 'OpenBuildStatusAutomatically',true)`

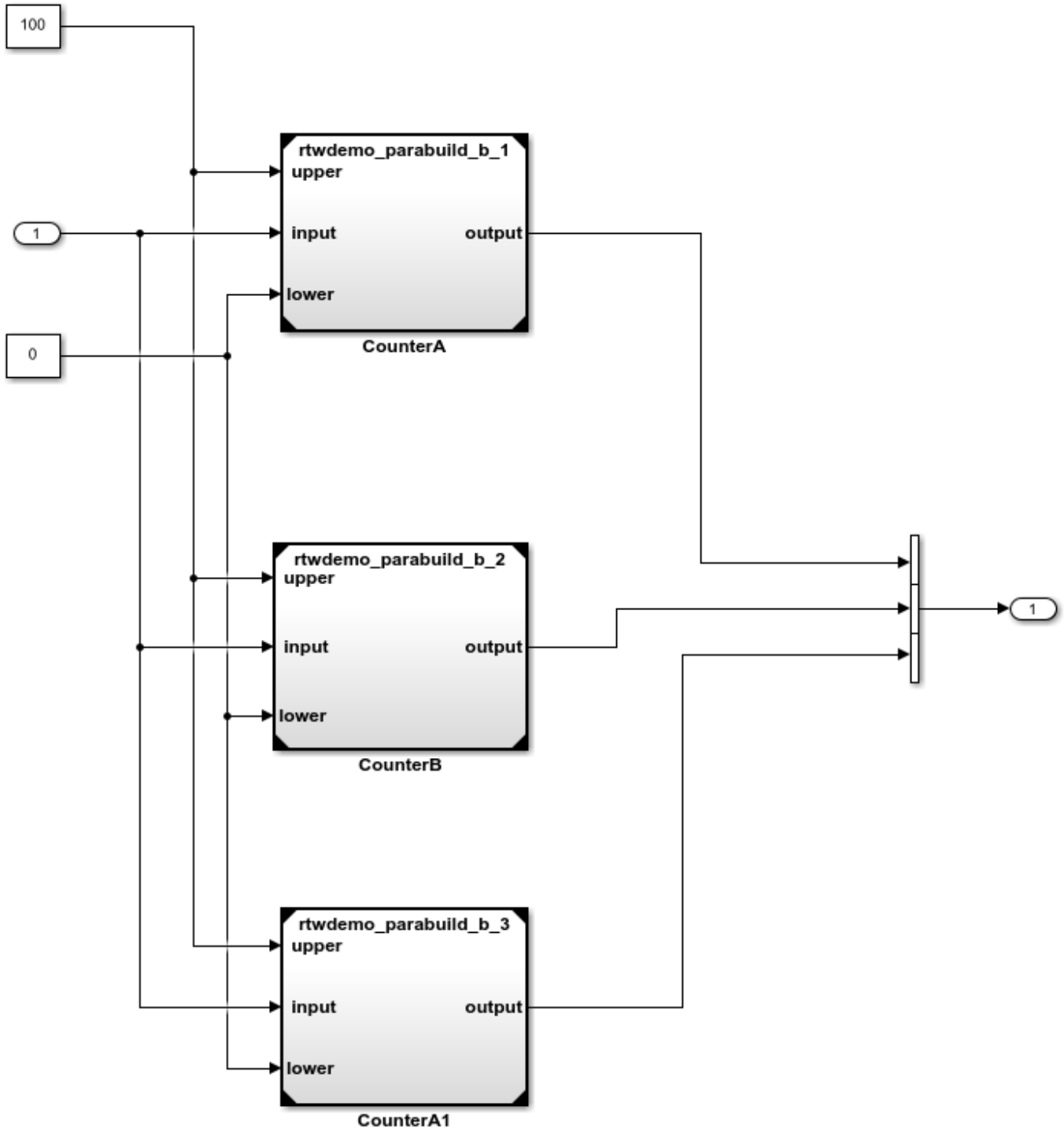
This example shows how you can monitor the build process for a model hierarchy. In the hierarchy, `rtwdemo_parabuild_a_1` references `rtwdemo_parabuild_b_1`, `rtwdemo_parabuild_b_2`, and `rtwdemo_parabuild_b_3`.

1. Copy the model files to a local folder.

```
src_dir = ...
    fullfile(matlabroot,'toolbox','rtw','rtwdemos');
if exist(fullfile('.','myTempFolder'),'dir')
    rmdir('myTempFolder','s')
end
mkdir myTempFolder

copyfile(fullfile(src_dir,'rtwdemo_parabuild_a_1.slx'), ...
         'myTempFolder');
copyfile(fullfile(src_dir,'rtwdemo_parabuild_b_1.slx'), ...
         'myTempFolder');
copyfile(fullfile(src_dir,'rtwdemo_parabuild_b_2.slx'), ...
         'myTempFolder');
copyfile(fullfile(src_dir,'rtwdemo_parabuild_b_3.slx'), ...
         'myTempFolder');

cd myTempFolder
```

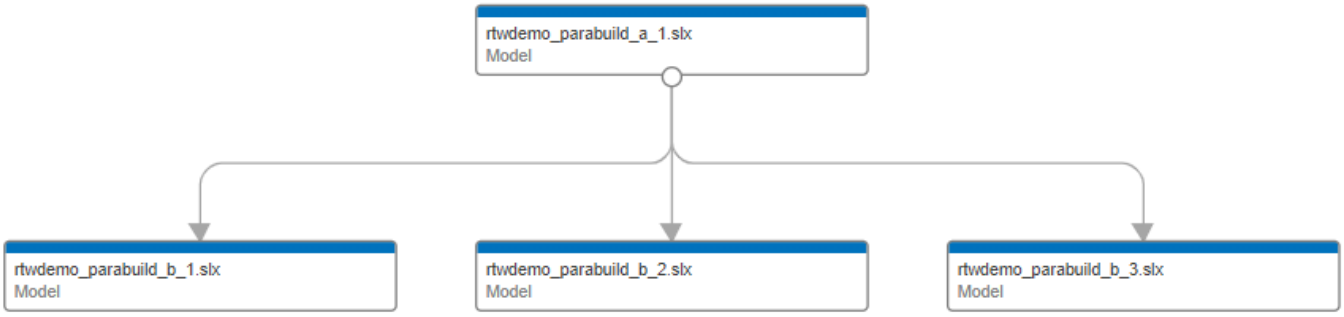
2. Open the top model.

```
open_system('rtwdemo_parabuild_a_1')
```



3 View the model dependencies with the Dependency Analyzer.

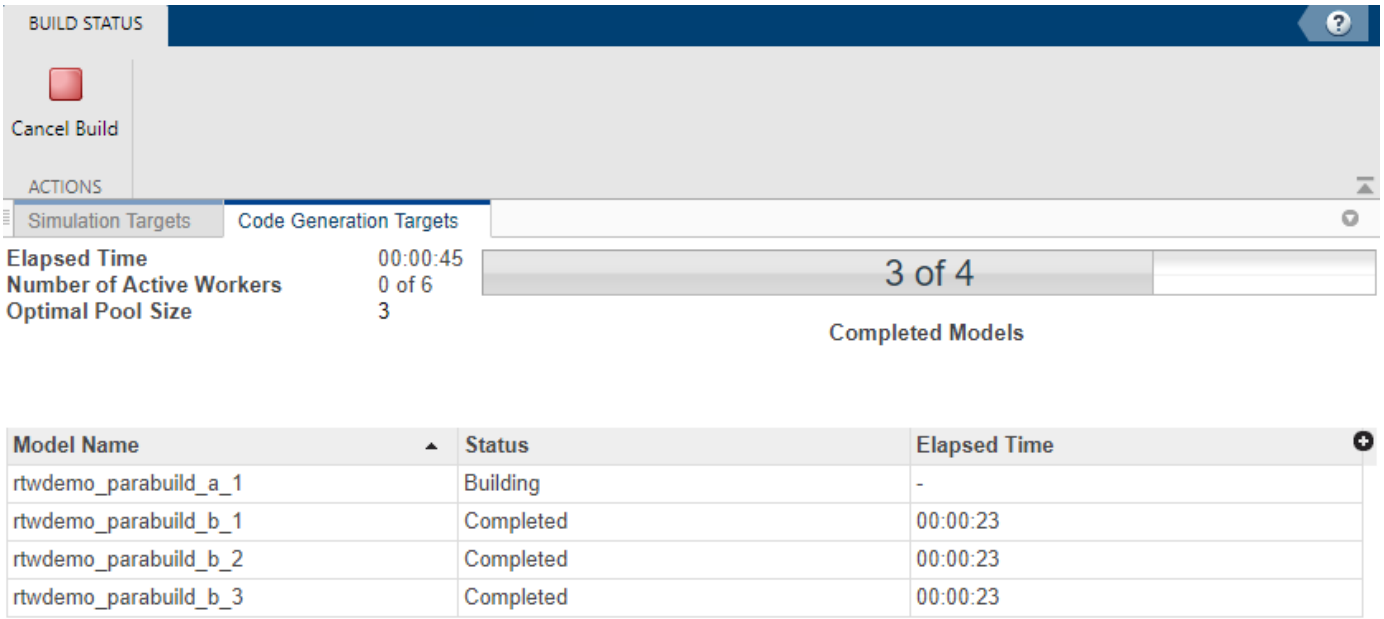
On the **Modeling** tab, click on the down arrow to expand the **Design** gallery. Under **Dependencies**, click **Dependency Analyzer**.



4 Start parallel building of the model hierarchy and open the Build Status window.

```
slbuild('rtwdemo_parabuild_a_1','OpenBuildStatusAutomatically',true)
```

The Build Status window displays the overall build progress and the build status for models as the code generator works through the model hierarchy.



This table explains the **Status** column values.

Status	Description
Blocked	The code generator cannot schedule the model build because the build is blocked by dependencies. For example, a child model build is not complete.
Scheduled	In parallel building, the code generator schedules a model build when the build is not blocked by dependencies. The status of the model build is `Scheduled` until a parallel pool worker becomes available.
Building	The model is being built on a worker.
Completed	When a model is built without errors, the status changes to `Completed`.
Error	If the model build fails, the status changes to `Error`.

Status	Description
Up To Date	Code generation is not required for the model because the current generated code is up to date.
Canceling	When you click the **Cancel Build** button, the build process changes the status of incomplete builds to `Canceling`.
Canceled	The canceling operation is complete.

The **Elapsed Time** column shows the build times for models in the hierarchy. To reduce the total build time, analyse the build times for referenced models together with model dependency information. Then, consider, for example:

- Restructuring the referenced model hierarchy.
- Increasing the number of workers in the parallel pool.

See Also

`slbuild` | `coder.buildstatus.open` | `coder.buildstatus.close`

More About

- “Control Regeneration of Top Model Code” on page 59-49
- “Reduce Update Time for Referenced Models by Using Parallel Builds”
- “Build Process Workflow for Real-Time Systems”
- “Support Model Referencing” on page 94-61

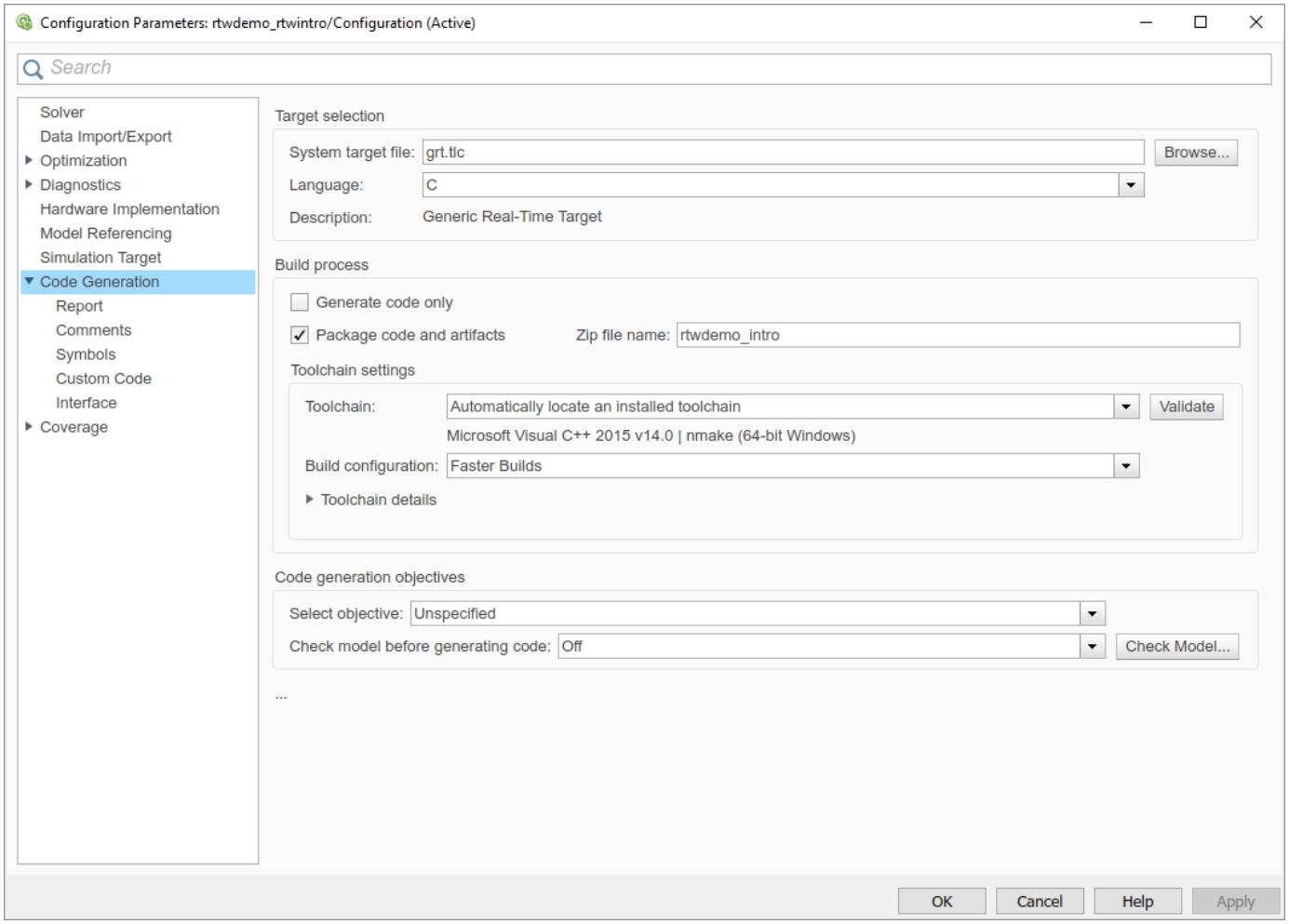
Relocate or Share Generated Code

To relocate or share static and generated code files, you can use the `packNGo` function. For example, use the function when you want to relocate generated code to a development environment that does not provide MATLAB and Simulink products.

Package and Relocate Generated Code

To package generated code:

1. In the Configuration Parameters dialog box, select **Package code and artifacts**. This option configures the build process to run the `packNGo` function after code generation.
2. In the **Zip file name** field, enter the name of the ZIP file in which you want to package the generated code and artifacts. You can specify the file name with or without the `.zip` extension. If you do not specify an extension or an extension other than `.zip`, the ZIP utility adds the `.zip` extension. If you do not specify a value, the build process uses the name *`model`*`.zip`, where *`model`* is the name of the top model.



3. Apply the changes and then generate code for your model.

4. Verify that the packaged code is ready for relocation by using a ZIP tool to inspect the resulting ZIP file.

Relocate the ZIP file to the destination development environment and use a ZIP tool to unpack the file. To compile the unpacked code, use a build tool.

If the destination environment provides Simulink Coder or Embedded Coder, you can use `codebuild` to compile the unpacked code.

Package and Relocate Generated Code From Command Line

To package and relocate generated code by using line commands, use this workflow:

1. Select a structure for the `zip` file. on page 59-57
2. Select a name for the `zip` file. on page 59-58
3. Package the model code files in the `zip` file. on page 59-58
4. Inspect the generated `zip` file. on page 59-59
5. Relocate and unpack the `zip` file. on page 59-59

Select a Structure for the ZIP File

Before you generate and package the files for a model build, decide whether you want the files to be packaged in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure.

If...	Then Use a...
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	Single, flat folder structure
The destination development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code depends on the relative location of files	Hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of `zip` files, a primary `zip` file, which in turn contains the following secondary `zip` files:

- `mlrFiles.zip` — files in your *`matlabroot`* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated code generation for the model
- `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees

Paths for the secondary `zip` files are relative to the root folder of the primary `zip` file, maintaining the source development folder structure.

Select a Name for the Zip File

By default, the `packNGo` function names the primary `zip` file *`model`*. You have the option of specifying a different name. If you specify a file name and omit the file type extension, the function appends `.` to the name that you specify.

Package Model Code in a Zip File

Package model code files by using the `PostCodeGenCommand` configuration parameter, `packNGo` function, and build information object for the model. You can set up the packaging operation to use:

- A system generated build information object.

  In this case, before generating the model code, use `set_param` to set the configuration parameter `PostCodeGenCommand` to an explicit call to the `packNGo` function. For example:

  ```
  set_param(bdroot, 'PostCodeGenCommand', 'packNGo(buildInfo);');
  ```

  After generating and writing the model code to disk and before generating a makefile, this command instructs the build process to evaluate the call to `packNGo`. This command uses the system generated build information object for the currently selected model.
- A build information object that you construct programmatically.

  In this case, you could use other build information functions to include paths and files selectively in the build information object that you then specify with the `packNGo` function. For example:

  ```
  .
  .
  .
  myModelBuildInfo = RTW.BuildInfo;
  addSourceFiles(myModelBuildInfo, {'test1.c' 'test2.c' 'driver.c'});
  .
  .
  .
  packNGo(myModelBuildInfo);
  ```

The following examples show how you can change the default behavior of `packNGo`.

To...	Specify...
Change the structure of the file packaging to hierarchical	`packNGo(buildInfo, 'packType' 'hierarchical');`
Rename the primary `zip` file	`packNGo(buildInfo, 'fileName' 'zippedsrcs');`
Change the structure of the file packaging to hierarchical and rename the primary `zip` file	`packNGo(buildInfo, 'packType' 'hierarchical'... 'fileName' 'zippedsrcs');`
Include header files found on the include path in the `zip` file	`packNGo(buildInfo, 'minimalHeaders' false);`
Generate warnings for parse errors and missing files	`packNGo(buildInfo, 'ignoreParseError' true... 'ignoreFileMissing' true);`

Note The `packNGo` function can modify the build information in the `RTW.BuildInfo` object that is passed as the first argument of the function. The function might find additional files from source and include paths recorded in build information for the model and add that information to the `RTW.BuildInfo` object.

Inspect Generated ZIP File

To verify that the generated ZIP file is ready for relocation, use a ZIP tool to inspect the file. Some ZIP tools allow you to view the file contents without unpacking the file. If unpacking is required and the model code files are packaged as a hierarchical structure, you have to unpack the primary and secondary ZIP files. When you unpack the secondary ZIP files, relative paths of the files are preserved.

Relocate and Unpack ZIP File

Relocate the ZIP file to the destination development environment and use a ZIP tool to unpack the file. To compile the unpacked code, use a build tool.

If the destination environment provides Simulink Coder or Embedded Coder, you can use `codebuild` to compile the unpacked code.

Code Packaging Example

This example shows how to package code files generated for the example model `rtwdemo_rtwintro`:

1. Set your working folder to a writable folder.
2. Open the model `rtwdemo_rtwintro` and save a copy to your working folder.
3. In the Command Window, enter:

   ```
   set_param('rtwdemo_rtwintro', 'PostCodeGenCommand',...
   'packNGo(buildInfo, ''packType'' ''hierarchical'')');
   ```
4. Generate code for the model.
5. Inspect the generated ZIP file, `rtwdemo_rtwintro.zip`. The ZIP file contains two secondary ZIP files, `mlrFiles.zip` and `sDirFiles.zip`.
6. Inspect `mlrFiles.zip` and `sDirFiles.zip`.
7. Relocate the ZIP file to your destination folder and unpack it.
8. To compile the unpacked code, use the `codebuild` function.

Limitations

packNGo Function

For information about limitations that apply to this function, see `packNGo`.

Executable File with Nondefault Extension

If a build process uses the template makefile approach, then `packNGo` uses the executable file extension specified by the linker tool to determine binary artifacts that require packaging.

If you generate an executable file with an extension that is not a default value, check that the extension is saved in the toolchain associated with the template makefile. For more information, see “Associate the Template Makefile with a Toolchain”.

If the build process generates an executable file with an extension that is different from the extension saved in the toolchain, `packNGo` does not package the executable file.

See Also

`packNGo` | `codebuild`

More About

- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Configure CMake Build Process” on page 59-13
- “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
- “Compile Code in Another Development Environment” on page 52-99
- “Build Process Workflow for Real-Time Systems”
- “Executable Program Generation” on page 59-62

Compile and Debug Generated C Code with Microsoft Visual Studio

This example shows how to generate code from a model and produce a Visual Studio® Solution. For the base example, see `rtwdemo_counter`.

About This Example

Building the example model generates a Visual Studio Solution. These model configuration parameters control the code generation:

- **System target file**
- **Toolchain**

When you build the model with this configuration, Simulink® Coder™ generates code in a Visual Studio Solution. Add this solution to a Visual Studio C/C++ project to integrate generated code with your custom Visual Studio code.

Before You Start

- This example builds code for Microsoft® Visual Studio running on Microsoft Windows® platforms.
- Simulink Coder uses the GRT code format, which is intended for rapid prototyping.
- Embedded Coder® uses the ERT code format, which is intended for production deployment.

Example Steps

1. Open the example model `rtwdemo_counter_msvc`. In the Command Window, type: open_system('rtwdemo_counter_msvc');
2. Set **System target file** to `grt.tlc` (`Generic Real-Time Target`).
3. Set **Toolchain** to a value of the form `Microsoft Visual Studio Project <year> | CMake (64-bit Windows)` that specifies an installed version of Microsoft Visual Studio. For example, `Microsoft Visual Studio Project 2019 | CMake (64-bit Windows)`.
4. From the **Apps** gallery, open the Simulink Coder app.
5. To generate code and build a program executable, on the **C Code** tab, click **Build**.
6. When the build process is complete, in the `rtwdemo_counter_msvc_grt_rtw > build` folder, you see a generated Visual Studio Solution.
7. In Microsoft Visual Studio, open the `rtwdemo_counter_msvc.sln` solution file.

Use the solution to build and debug the generated code in Visual Studio.

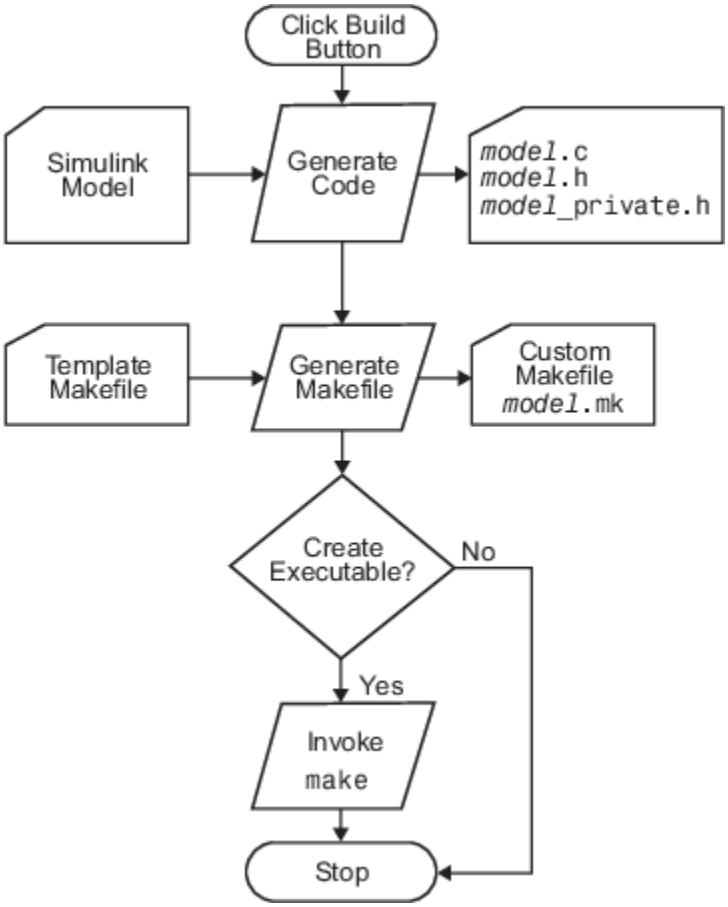
See Also

Related Examples

- “Choose an External Code Integration Workflow”

Executable Program Generation

The following figure shows how the process for program building.

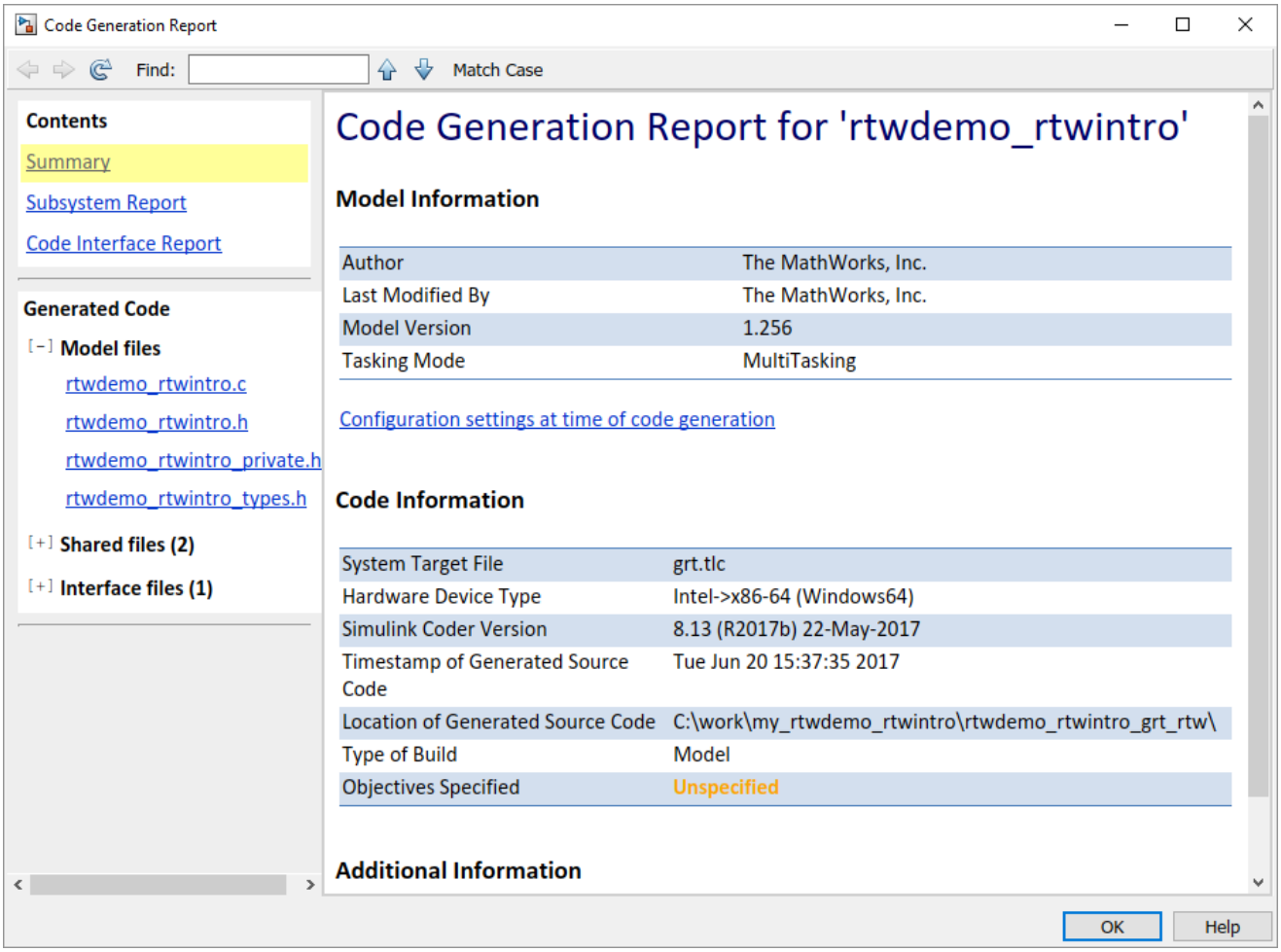


During the final stage of processing, the build process invokes the generated makefile, *model*.mk, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the environment for invoking the `make` utility and related compiler tools. To avoid recompiling C files, the `make` utility performs date checking on the dependencies between the object and C files; only out-of-date source files are compiled. Optionally, the makefile can download the resulting executable image to your target hardware.

This stage is optional, as illustrated by the control logic in the preceding figure. You could choose to omit this stage (for example, if you are targeting an embedded microcontroller board).

To omit this stage of processing, in the Configuration Parameters dialog box, select the **Generate code only** check box. You can then cross-compile your code and download it to your target hardware.

If you also select the **Create code generation report** check box, the code generator produces an HTML report. To view the report, on the **C Code** tab, select **Open Latest Report**. This example is a code generation report for a generic real-time (GRT) system target file.



See Also

More About

- “Approaches for Building Code Generated from Simulink Models” on page 59-11
- “Build Process Workflow for Real-Time Systems”

Host/Target Communication in Simulink Coder

- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2
- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Graphical Controls for XCP External Mode Simulations” on page 60-27
- “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35
- “Register Custom Launcher” on page 60-43
- “Customize XCP Server Software” on page 60-45
- “External Mode Simulation with TCP/IP or Serial Communication” on page 60-55
- “Create a Transport Layer for TCP/IP or Serial External Mode Communication” on page 60-94

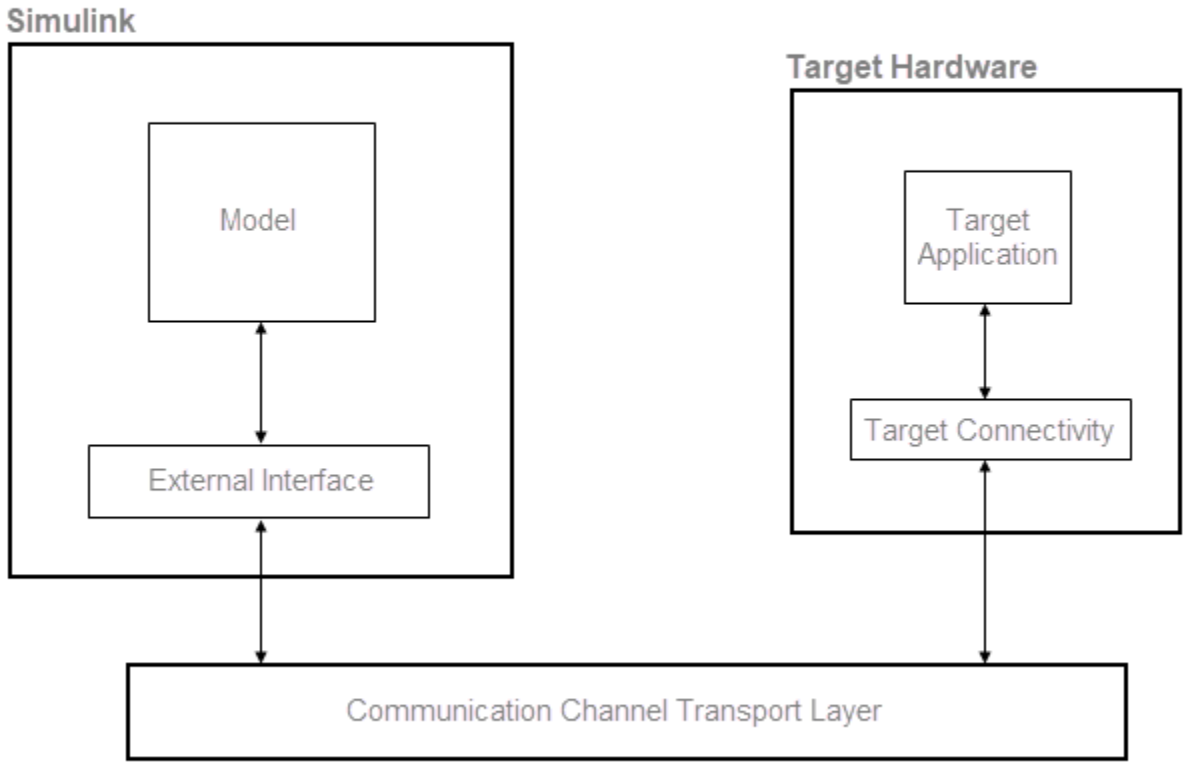
External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling

You can use external mode simulations for rapid prototyping. An external mode simulation establishes a communication channel between Simulink on your development computer (host) and the target hardware that runs the executable file created by the code generation and build process.

Through the communication channel, you can:

- Modify or tune block parameters in real time. When you change parameters in your model, Simulink downloads the new values to the executing target application.
- Monitor and save signal data from the executing target application.
- Retrieve execution-time metrics if code execution profiling is enabled for the external code simulation.

The low-level transport layer of the channel handles the transmission of messages. Simulink and the generated model code are independent of this layer. The transport layer and its interface code are isolated in separate modules that format, transmit, and receive messages and data packets.



Communication Mechanisms for External Mode

Simulink supports two communication mechanisms for external mode simulation. To decide which mechanism to use, see the information in this table.

Communication Mechanism	Supports	Reasons for Selecting Communication Mechanism
XCP, the Universal Measurement and Calibration Protocol	ERT and GRT system target files. Some Simulink support packages. For details, see https://www.mathworks.com/hardware-support.html?q=&page=1.	XCP external mode: • Uses a standard communication protocol. • Requires only a lightweight communication software stack on the target hardware. • Supports signal logging and streaming for Dashboard blocks and the Simulation Data Inspector. You can stream signals from within a referenced model hierarchy. • Provides documented `ext_mode.h` API for external mode target connectivity. • Supports streaming of execution-time metrics to the Simulation Data Inspector for host-based and support package external mode simulations. For information about running XCP external mode simulations, see: • "External Mode Simulation by Using XCP Communication" on page 60-8 • "Customize XCP Server Software" on page 60-45
TCP/IP and serial (RS-232)	ERT, GRT, and RSim system target files. Simulink support packages.	XCP external mode simulations do not support your target hardware because of known limitations. For information about running TCP/IP and serial external mode simulations, see: • "External Mode Simulation with TCP/IP or Serial Communication" on page 60-55 • "Create a Transport Layer for TCP/IP or Serial External Mode Communication" on page 60-94

Simulink Feature Support for External Mode

This table summarizes feature support for both forms of external mode simulations.

Feature		XCP Support	TCP/IP and Serial Support
Parameter tuning	With Dashboard blocks	Yes	Yes

Feature		XCP Support	TCP/IP and Serial Support
	Of tunable block parameters	Yes	Yes
Simulation Data Inspector		Yes. Includes signals within referenced models.	No
Logic Analyzer		Yes. Includes signals within referenced models.	No
Blocks that receive and display signals from target application	Dashboard library	Yes	No
	Floating Scope, Scope	Yes	Yes
	Spectrum Analyzer, Time Scope (DSP System Toolbox)	Yes	Yes
	Display	Yes	Yes
	To Workspace	Yes	Yes
	User-written S-Function. A method, which enables user-written blocks to support external mode, is built into the S-function API. See `matlabroot/simulink/include/simstruc.h`.	Yes	Yes
	Record, XY Graph	Yes	Yes
Signal Viewing Subsystem on page 60-74		Yes	Yes

External Mode Simulations by Using the Run on Custom Hardware App

To run external mode simulations on your development computer or target hardware that is not supported by MathWorks support packages, use the Run on Custom Hardware app. For an external mode simulation, you:

1. Build the target application on your development computer.
2. Deploy the target application to the target hardware.
3. Connect Simulink to the target application that runs on the target hardware.
4. Start execution of generated code on the target hardware.

With the app, you can perform the steps separately or by using one-click. This table shows the workflow support that the app provides for your target hardware.

Workflow Feature or Task		Target Hardware	
		Development Computer	Custom Hardware
One-click		Yes	Yes, if you set up connectivity on page 60-35 between Simulink and your target hardware.
Step-by-step	**Build for Monitoring**	Yes	Yes. Software builds target application on development computer by using the toolchain that you specify.
	Deploy	Yes	Yes, if you set up connectivity on page 60-35 between Simulink and your target hardware.
	Connect	Yes	Yes
	Start	Yes	Yes

External Mode Control Panel

Using the External Mode Control Panel, you can perform some of the functions that the Run on Custom Hardware app provides, for example, **Connect**, **Start**, and **Disconnect**. You can also open the External Signal & Triggering dialog box, which enables you to:

- Select logged signals that you want to monitor.
- Configure a trigger for uploading target application data.

For details, see XCP panel on page 60-28 or TCP/IP and serial panel on page 60-61.

Security for External Mode Simulations

Ensure that access to your development computer and the target hardware is secure.

Provide a secure communication channel between your development computer and the target application by applying the security measures listed in this table.

Communication Protocol	Security Measure
XCP on TCP/IP, TCP/IP	Run your development computer and target application within a trusted private network or virtual private network (VPN).
XCP on serial, serial	Use a point-to-point link between your development computer and target application, for example, a serial cable.

Before generating code for production, to reduce the risk of unintended access to the target application, disable external mode settings.

See Also

More About

- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35
- “Customize XCP Server Software” on page 60-45
- “External Mode Simulation with TCP/IP or Serial Communication” on page 60-55
- “Create a Transport Layer for TCP/IP or Serial External Mode Communication” on page 60-94
- “Create Execution-Time Profile for Generated Code” on page 81-2

External Mode Simulation by Using XCP Communication

In an external mode simulation, you can tune parameters in real time and monitor target application signals. Using the Run on Custom Hardware app, you can set up and run on your development computer or custom hardware external mode simulations that use an XCP communication channel.

For an external mode simulation, you:

1. Build the target application on your development computer.
2. Deploy the target application to the target hardware.
3. Connect Simulink to the target application that runs on the target hardware.
4. Start execution of generated code on the target hardware.

Run XCP External Mode Simulation on Development Computer

Configure and run an external mode simulation that uses the XCP communication protocol. During the simulation:

- Monitor a signal by using a Scope block, a Dashboard block, and the Simulation Data Inspector.
- Tune a parameter by using a Dashboard block.

Configure Signal Monitoring and Parameter Tuning for XCP

1. Create a folder for this example.

   ```
   mkdir ext_mode_xcp_example
   cd ext_mode_xcp_example
   ```

2. Open Simulink and create a simple model, `xcpExample`, which contains these blocks:

   - Sine Wave
   - Gain
   - Scope
   - Half Gauge
   - Knob

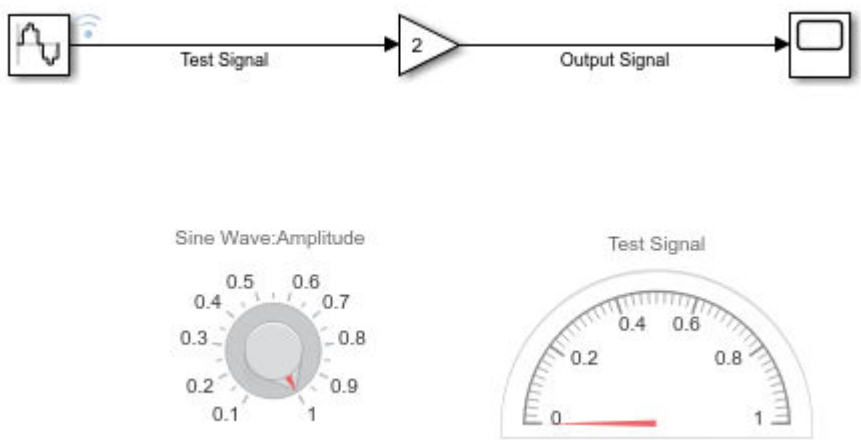
3. Double-click the Sine Wave block. Set **Sample time** to 0.1, and then click **OK**.
4. Double-click the Gain block. Set **Gain** to 2, and then click **OK**.
5. Connect the Sine Wave block to the Gain block and name the connection, for example, `Test Signal`.
6. Connect the Gain block to the Scope block and name the connection, for example, `Output Signal`.
7. Configure `Test Signal` for logging:

   a. Select `Test Signal`.

   b. On the **Signal** tab, in the **Monitor** section, select **Log Signals**.

   If you do not enable signal logging, you cannot stream `Test Signal` to the Simulation Data Inspector.

8. Configure the Half Gauge block to monitor the value of `Test Signal`:
   a. Double-click the Half Gauge block.
   b. In the Simulink Editor, select `Test Signal`.
   c. In the Block Parameters dialog box:
      - Connect the block to `Test Signal`.
      - In the **Maximum** field, enter a value, for example, `1`.
   d. Click **OK**.
9. Configure the Knob block to tune the **Amplitude** parameter of the Sine Wave block:
   a. Double-click the Knob block.
   b. In the Simulink Editor, select the Sine Wave block.
   c. In the Block Parameters dialog box:
      - Connect the block to the **Amplitude** parameter of the Sine Wave block.
      - In the **Minimum** and **Maximum** fields, enter values, for example, 0.1 and 1 respectively.
   d. Click **OK**.
10. Save the model as `xcpExample`.



Specify Hardware and Prepare Model

1. From the **Apps** tab on the Simulink toolstrip, in the **Setup to Run on Hardware** section, click **Run on Custom Hardware**. On the Run on Custom Hardware pop-up dialog box, click **Finish**.
2. In the **Hardware** section, specify the system target file. For this example, use the default, `grt.tlc`. If you have Embedded Coder, you can specify `ert.tlc`.
3. In the **Prepare** gallery, under **Model Configuration**, click **Hardware Settings**. The Configuration Parameters dialog box opens, displaying **Hardware Implementation** settings that are determined by the system target file.
   a. On the **Solver** pane, specify `Fixed-step` in the **Type** field. Then, under **Solver details**, specify 0.1 in the **Fixed-step size** field.

b On the **Code Generation** > **Optimization** pane, set **Default parameter behavior** to `Tunable`.

c On the **Code Generation** > **Interface** pane, select the **External mode** check box. Then set **Transport layer** to `XCP on TCP/IP`. Selecting `XCP on TCP/IP` automatically specifies `ext_xcp` for **MEX-file name**, selects the **Automatically allocate static memory** check box, and enables **Maximum duration**. For this example, use the default value of **Maximum duration**.

d If **System target file** is `ert.tlc`, on the **Code Generation** > **Templates** pane select the **Generate an example main program** check box, and set **Target operating system** to `BareBoardExample`.The code generator uses the documented APIs of the external mode abstraction layer that are available in *`matlabroot`*`\toolbox\coder\xcp\src\target\ext_mode\include\ext_mode.h` to create an example that shows how you can provide XCP external mode target connectivity for a bare board that does not run an operating system.

e Click **OK** to apply the changes and close the dialog box.

4 Save the model.

Build and Run Target Application

In the **Run on Hardware** section:

1 To specify a nondefault value for the simulation stop time, in the **Stop Time** field, specify your value, for example, `inf`.

2 Click **Monitor & Tune** to start the process. The software:

a Builds the target application files including the executable file. The debugging symbols for signals are created as part of the executable file or in a separate file, depending on the configuration of Code Generation. To learn more about different configurations for Code Generation, see “Approaches for Building Code Generated from Simulink Models”.

b Deploys the target application as a separate process on your development computer.

c Connects Simulink to the target application.

d Starts the generated model code.

To perform the steps separately, click the following buttons under **Monitor & Tune** > **Step by Step Commands**, in this order:

a **Build for Monitoring**

b **Deploy**

c **Connect**

d **Start**

Monitor Signal and Tune Parameter

You can monitor `Test Signal` through the:

- Scope block –– Double-click the block.
- Simulation Data Inspector –– Click the Simulation Data Inspector button. When the Simulation Data Inspector opens, select the **Test Signal** check box, which displays streamed data.
- Half-Gauge block.

To change the amplitude of the sine wave, rotate the pointer on the Knob block to the required value.

This table describes more ways of tuning tunable block parameters during a simulation.

Approach	Details
Model Data Editor	To tune parameters through the Model Data Editor: **1** On the **Hardware** tab of the Simulink Editor, in the **Prepare** section, click **Tune Parameters**, which opens the Model Data Editor. **2** If you want to update multiple tunable parameters simultaneously, in the **Prepare** section, toggle on **Hold Updates**. **3** On the **Parameters** tab of the Model Data Editor, in the **Value** column, specify new values for your tunable parameters. **4** Toggle off **Hold Updates** or click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application simultaneously. If **Hold Updates** is off, then immediately after you specify a new value, Simulink downloads the new value to the target application. For more information, see **Model Data Editor**.
Block Parameter dialog box	To tune parameters through the Block Parameter dialog box: **1** For each block that you want to update: **a** Double-click the block, which opens the Block Parameter dialog box. **b** In the parameter fields, specify new parameter values. **c** Click **Apply** or **OK**. **2** If **Hold Updates** is on, toggle off **Hold Updates** or click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application simultaneously. If **Hold Updates** is off, then immediately after you click **Apply** or **OK**, Simulink downloads the new block values to the target application.

Approach	Details
MATLAB workspace	If block parameters are MATLAB workspace variables: 1 In the Command Window, assign new values to the variables. 2 On the **Hardware** tab of the Simulink Editor, in the **Prepare** section, click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application. For more information, see “Create and Edit Variables”.

For more information about parameter tuning with generated code, see “External Mode Simulation with TCP/IP or Serial Communication” on page 60-55 and “Create Tunable Calibration Parameter in the Generated Code” on page 36-112.

If your model contains a Stateflow chart, you can view state activity. For more information, see “Animate Stateflow Charts” (Stateflow).

Stop Target Application

To stop the execution and disconnect the target application before **Stop Time** is reached, in the **Run on Hardware** section of the **Hardware** tab, click **Stop** .

If you want to disconnect the target application from Simulink without stopping code execution, click **Stop** > **Step by Step Commands** > **Disconnect**.

Triggered Signal Monitoring for XCP External Mode Simulations

For XCP external mode simulations, you can configure a trigger that starts the uploading of target application data for signal monitoring in Simulink.

1 **Create a simple model in a local folder.**

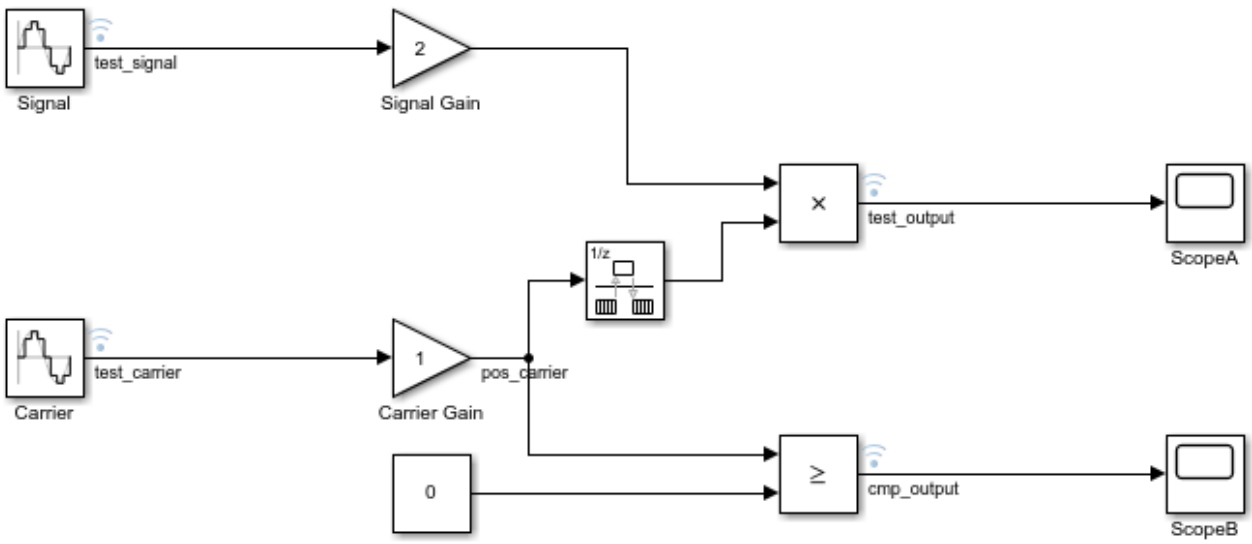
In the Command Window, enter these commands:

```
% Create work folder
if mkdir('workFolder')
    rmdir 'workFolder' s
    mkdir('workFolder')
end
cd workFolder
open_system(new_system('myModel'));

% Add, configure, and connect blocks
add_block('simulink/Sources/Sine Wave','myModel/Signal',...
                                        'Bias','5',...
                                        'Amplitude','2',...
                                        'SampleTime','0.01');
set_param('myModel/Signal','position',[140,80,180,120]);

add_block('simulink/Math Operations/Gain','myModel/Signal Gain',...
                                        'Gain','2');
set_param('myModel/Signal Gain','position',[340,80,380,120]);
add_line('myModel','Signal/1','Signal Gain/1');
```

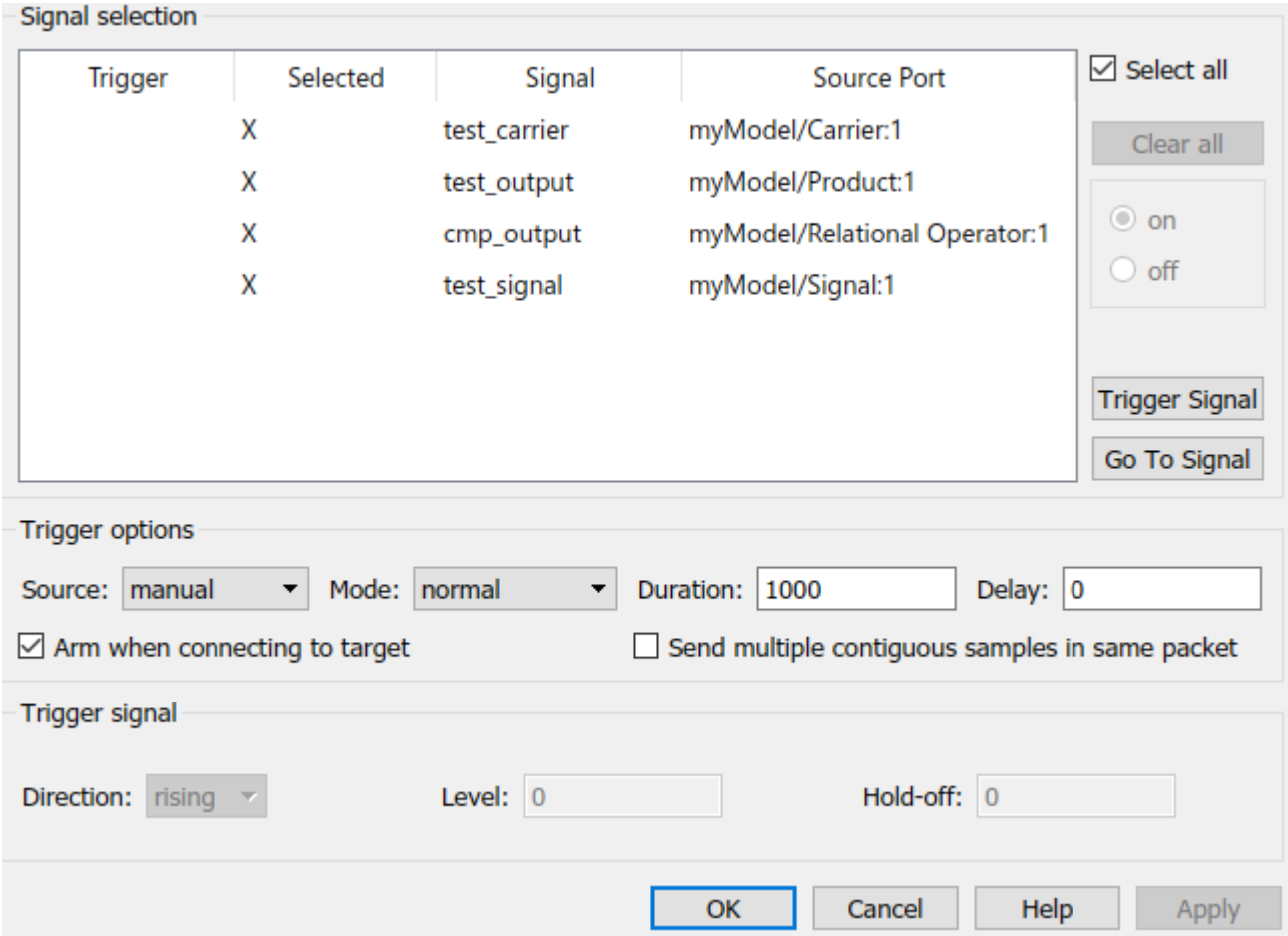
```
% Label signal
signal=get_param('myModel/Signal','PortHandles');
set_param(signal.Outport(1),'Name','test_signal');

% And so on
add_block('simulink/Math Operations/Product','myModel/Product');
set_param('myModel/Product','position',[540,160,580,200]);
add_line('myModel','Signal Gain/1','Product/1');

add_block('simulink/Sources/Sine Wave','myModel/Carrier',...
                                       'Bias','5',...
                                       'Amplitude','10',...
                                       'Frequency','3',...
                                       'SampleTime','0.02');

set_param('myModel/Carrier','position',[140,240,180,280]);

add_block('simulink/Math Operations/Gain','myModel/Carrier Gain');
set_param('myModel/Carrier Gain','position',[340,240,380,280]);
add_line('myModel','Carrier/1','Carrier Gain/1');

signal=get_param('myModel/Carrier','PortHandles');
set_param(signal.Outport(1),'Name','test_carrier');

add_block('simulink/Signal Attributes/Rate Transition',...
          'myModel/Rate Transition');
set_param('myModel/Rate Transition','position',[440,200,480,240]);
add_line('myModel','Carrier Gain/1','Rate Transition/1');
add_line('myModel','Rate Transition/1','Product/2');

add_block('simulink/Sources/Constant','myModel/Constant',...
          'Value','0');
set_param('myModel/Constant','position',[340,320,380,360]);

add_block('simulink/Logic and Bit Operations/Relational Operator',...
          'myModel/Relational Operator','Operator','>=');
set_param('myModel/Relational Operator','position',[540,300,580,340]);
add_line('myModel','Carrier Gain/1','Relational Operator/1');
add_line('myModel','Constant/1','Relational Operator/2');

signal=get_param('myModel/Carrier Gain','PortHandles');
set_param(signal.Outport(1),'Name','pos_carrier');

add_block('simulink/Sinks/Scope','myModel/ScopeA');
set_param('myModel/ScopeA','position',[740,160,780,200]);
add_line('myModel','Product/1','ScopeA/1');

signal=get_param('myModel/Product','PortHandles');
set_param(signal.Outport(1),'Name','test_output');

add_block('simulink/Sinks/Scope','myModel/ScopeB');
set_param('myModel/ScopeB','position',[740,300,780,340]);
add_line('myModel','Relational Operator/1','ScopeB/1');

signal=get_param('myModel/Relational Operator','PortHandles');
set_param(signal.Outport(1),'Name','cmp_output');

% Finally, save the model
save_system('myModel');
```

2. In the Simulink Editor, configure these signals for logging by right-clicking each signal and then selecting **Log Selected Signals**:

   - `test_signal`
   - `test_carrier`
   - `test_output`
   - `cmp_output`



3. From the **Apps** tab on the Simulink toolstrip, in the **Setup to Run on Hardware** section, click **Run on Custom Hardware**. On the Run on Custom Hardware pop-up dialog box, click **Finish**.
4. In the **Prepare** gallery, under **Model Configuration**, click **Hardware Settings**.
5. On the **Solver** pane, in the **Type** field, specify `Fixed-step`.
6. On the **Code Generation** > **Interface** pane, select the **External mode** check box.
7. Set **Transport layer** to `XCP on TCP/IP`, which specifies `ext_xcp` for **MEX-file name**.
8. Click **OK**. Then save the model.
9. To open the External Signal & Triggering dialog box, from the **Prepare** gallery, under **Signal Monitoring & Tracing**, click **Control Panel**.
10. In the **Configuration** section, click **Signal & Triggering**, which opens the External Signal & Triggering dialog box.



Through this dialog box, you can select signals for monitoring and configure the trigger for uploading data from the target application. For this example, use the default settings.

- The **Select all** check box is selected — The dialog box displays an X in each row of the **Selected** column. You can monitor all logged signals during the simulation.
- **Source** is set to **manual** –– The External Mode Control Panel provides manual control of data uploading, for example, the **Arm Trigger** button.
- The **Arm when connecting to target** check box is selected — When you connect Simulink to the target application, the trigger is armed.

11. Click **OK** to close the External Signal & Triggering dialog box, then click **OK** to close the External Mode Control Panel.
12. In the Simulink Editor, go to the **Hardware** tab.
13. In the **Stop Time** field of the **Run on Hardware** section, specify your value, for example, `inf`.
14. Save the model.
15. Click **Monitor & Tune** to start the process. The software:

    a. Builds the target application.
    b. Deploys the target application as separate process on your development computer.
    c. Connects Simulink to the target application.

d Starts the generated model code.

To view the scope displays, double-click `ScopeA` and `ScopeB`.

To monitor signals in the Simulation Data Inspector, in the **Review Results** section, click **Data Inspector**.

From the External Mode Control Panel, you can suspend or resume signal logging and monitoring. In the **Connection and triggering** section, click **Cancel Trigger** or **Arm Trigger** respectively.

To stop the execution of the target application and disconnect Simulink from the target environment, click **Stop Real-Time Code**.

For information about signal monitoring and trigger options, see “XCP External Mode Control Panel” on page 60-28 and “XCP External Signal & Triggering Dialog Box” on page 60-29.

Run XCP External Mode Simulation From Command Line

You can use commands to run XCP external mode simulations. To retrieve and set the values of model parameters, use the `get_param` and `set_param` commands.

To run these commands, you must have a Simulink model open and a target application running.

1. Set the model simulation mode to external mode.

   ```
   set_param(gcs,'SimulationMode','external');
   ```

2. Connect Simulink to the target application.

   ```
   set_param(gcs,'SimulationCommand','connect')
   ```

3. Run generated model code.

   ```
   set_param(gcs,'SimulationCommand','start');
   ```

4. To tune a parameter, change its workspace variable value through a line command. For example, if a block parameter value is specified as a `Simulink.Parameter` object, assign the new value to the `Value` property.

   ```
   myParamObj.Value = 5.23;
   ```

5. To download the new value to the target application, update the model.

   ```
   set_param(gcs,'SimulationCommand','update');
   ```

6. Stop the target application and disconnect Simulink from the target environment.

   ```
   set_param(gcs,'SimulationCommand','stop');
   ```

   To disconnect Simulink from the target application without stopping execution of generated code, use this command:

   ```
   set_param(gcs,'SimulationCommand','disconnect');
   ```

The `set_param` commands that use the `'SimulationCommand'` argument are asynchronous. If you run the commands successively from a script, each command starts without waiting for the previous command to complete. To check that each command is complete, in the script, use the `get_param` command with the `'SimulationStatus'` argument. For example, for steps 1 to 3, specify these commands in the script:

```
set_param(gcs,'SimulationMode','external');
set_param(gcs,'SimulationCommand','connect');

isExternalSimulationActive = false;
while ~isExternalSimulationActive
    simStatus = get_param(gcs, 'SimulationStatus');
    isExternalSimulationActive = strcmp(simStatus, 'external');
end
set_param(gcs,'SimulationCommand','start');
```

For more information, see “Run Simulations Programmatically”.

The Diagnostic Viewer displays error messages produced by the `get_param` and `set_param` commands.

This table gives parameters that you can use in `get_param` and `set_param` commands.

XCP External Mode Command-Line Parameters

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeArmWhenConnect` `off`, **`on`**	External Signal & Triggering: **Arm when connecting to target** check box	Arm the trigger as soon as the Simulink Coder software connects to the target.
`ExtModeBatchMode` **`off`**, `on`	External Mode Control Panel: **Batch download** check box	Enable or disable downloading of parameters in batch mode.
`ExtModeConnected` **`off`**, `on`	External Mode Control Panel: **Connect/Disconnect** button	Indicate the state of the connection with the target application.
`ExtModeEnableFloating` `off`, **`on`**	External Mode Control Panel: **Enable data uploading** check box	Enable or disable the arming and canceling of triggers when a connection is established with floating scopes.
`ExtModeLogAll` `off`, **`on`**	External Signal & Triggering: **Select all** check box	Upload available signals from the target to the host.
`ExtModeSendContiguousSamples` **`off`**, `on`	External Signal & Triggering: **Send multiple contiguous samples in same packet** check box	Specify XCP DAQ packed mode for streaming signals to Simulink. Target application sends samples from multiple time steps in a packet. `ExtModeTrigDuration` determines number of samples sent in the packet.
`ExtModeTrigDelay` *`integer`* (**`0`**)	External Signal & Triggering: **Delay** text field	Specify the amount of time (expressed in base rate steps) that elapses between a trigger occurrence and the start of data collection.
`ExtModeTrigDirection` *`character vector`* - **`rising`**, `falling`, `either`	External Signal & Triggering: **Direction** menu	Specify the direction in which the signal must be traveling when it crosses the threshold value.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigDuration` *`integer`* (**`1000`**)	External Signal & Triggering: **Duration** text field	Specify the number of base rate steps for which external mode logs data after a trigger event. If `ExtModeSendContiguousSamples` is `on`, the number of samples sent in a packet from the target application to Simulink is equal to: `ExtModeTrigDuration/n` where `n` = (base rate)/(sample rate) For `n=1`, number of samples sent in a packet is exactly `ExtModeTrigDuration` value. For subrates in multirate models, number of samples sent in a packet is rounded down to the nearest integer. The target application discards buffer samples that are not sent to Simulink.
`ExtModeTrigHoldOff` *`integer`* (**`0`**)	External Signal & Triggering: **Hold-off** text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.
`ExtModeTrigLevel` *`integer`* (**`0`**)	External Signal & Triggering: **Level** text field	Specify the threshold value the trigger signal must cross to fire the trigger.
`ExtModeTrigMode` *`character vector`* - **`normal`**, `oneshot`	External Signal & Triggering: **Mode** menu	Specify whether the trigger is to rearm automatically after each trigger event or whether only one buffer of data is to be collected each time the trigger is armed.
`ExtModeTrigSignalBlockPath` *`character vector`*	External Signal & Triggering: In **Signal selection** view, select signal and then click **Trigger Signal** button.	Specify the path to the block that contains the output port that provides the trigger signal.
`ExtModeTrigSignalOutputPortIndex` *`integer`*	External Signal & Triggering: In **Signal selection** view, select signal and then click **Trigger Signal** button.	Specify the index of the trigger signal output port for the block specified by `ExtModeTrigSignalBlockPath`.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigType` *`character vector`* - **`manual`**, `signal`	External Signal & Triggering: **Source** menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.

Memory Allocation for Communication Buffers During XCP External Mode Simulation

You do not have to manually determine memory requirements for XCP-based external mode simulations with models that, for example, stream a large amount of data to Simulink or run on low-memory target devices.

Setting **Transport layer** to `XCP on TCP/IP` or `XCP on Serial` enables these configuration parameters:

- **Automatically allocate static memory** (`ExtModeAutomaticAllocSize`)
- **Maximum duration** (`ExtModeMaxTrigDuration`) — Default is 10 base-rate steps.

The software uses model information, including the **Maximum duration** value, to allocate memory for:

- Internal data structures used by the XCP stack
- Storage of XCP packets

If you clear the **Automatically allocate static memory** check box:

- Memory allocated for internal data structures is a default value.
- You can use the **Static memory buffer size** (`ExtModeStaticAllocSize`) field to specify memory allocated for XCP packets. The default value is 1,000,000 bytes.

If insufficient memory is allocated, the external mode simulation issues an out-of-memory error when it tries to reserve memory for internal data structures or signal streaming. In this case, reselect the **Automatically allocate static memory** check box to enable the software to compute the amount of memory required.

Target Application Arguments

You can run your target application with optional arguments. You can pass the optional arguments to your target application through your custom launcher implementation.

Argument	Description
`-w`	Specify that the target application enters and stays in a wait state until it receives a message from Simulink. If you do not specify `-w`, the target application executes model code immediately. The model code runs with parameter values from the time when you built the model.

Argument	Description
`-tf` *`time`*	Override the model parameter `StopTime`. Specify *`time`* in: • Seconds if `PurelyIntegerCode` is set to `'off'`. • Base rate steps if `PurelyIntegerCode` is set to `'on'`. `-tf inf` specifies that the model runs indefinitely when model code is executed.

For host-based external mode simulations, you can specify additional `rtiostream on page 88-42` arguments.

Argument	Description
`-verbose` *`level`*	Specify verbosity level: • 0 –– No information • 1 –– Detailed information
`-port` *`number`*	For `XCP on TCP/IP` transport layer, specify the port number of the TCP/IP server. Use an integer value between 256 and 65535. Default is 17725. For `XCP on Serial` transport layer, specify the serial port ID. For example: On Windows, `'COM1'` or 1 for `COM1`, `'COM2'` or 2 for `COM2`, and so on. On Linux, `'/dev/ttyS0'`, and so on.
`-baud` *`value`*	For `XCP on Serial` transport layer, specify the baud value: 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600 (default), or 115200.

XCP External Mode Limitations

This table describes limitations that apply to external mode simulations that use XCP communication.

Feature	Details
Parameter updates that change model structure	You cannot change: • The number of states, inputs, or outputs of a block • The sample time or the number of sample times • The integration algorithm for continuous systems • The name of the model or of a block If you make parameter updates that change the model structure, you must rebuild the target application. You can change numerator and denominator polynomial parameters for the Transfer Fcn, Discrete Transfer Fcn, and Discrete Filter blocks if the number of states does not change. You cannot change zero entries in the State-Space, Zero-Pole, and Discrete Zero-Pole blocks in the user-specified or computed parameters, that is, the A, B, C, and D matrices obtained by a zero-pole to state-space transformation. In the State-Space block, if you specify the matrices in the controllable canonical realization, then changes to the A, B, C, and D matrices that preserve this realization and the dimensions of the matrices are allowed. If the Simulink block diagram does not match the target application, Simulink produces an error stating that the checksums do not match. The checksums take into account the top models, but not referenced models. To rebuild the target application, use the updated block diagram.
Signal value display	The graphical display of signal values during a simulation is not supported. For example, you cannot use the **Data Display in Simulation** menu items **Show Value Labels When Hovering**, **Toggle Value Labels When Clicked**, and **Show Value Label of Selected Port**. For more information, see “Display Signal Values in Model Diagrams”.
Data archiving	**Data Archiving** features available on the **External Mode Control Panel** are not supported. You can use the Simulation Data Inspector to collect and export data.
Overriding signal logging settings	If you use the Signal Logging Selector to override signal logging settings, the controls to override the **Decimation** and **Limit data points** settings are not supported.

Feature	Details
Subsystem signal logging	For external mode simulations, signal logging occurs in response to an event on the target hardware. Currently, the only supported logging event is a new sample time tick. Some subsystems are enabled or disabled by control signals. For these subsystems, the software logs signal values even when the control signals for the subsystems are not enabled. For example, if signals in an Enabled Subsystem block are configured for logging, the software logs the signal values even when the control signal for the subsystem is not greater than zero. In this case, the logged values are the values the signals had when the control signal was last greater than zero. For more information about the subsystems for which this limitation applies, see “Simulink Subsystem Semantics”.
Compiler debug symbol format	Your toolchain must generate debug information in one of these formats: • DWARF • PDB
Global variables	Signals, parameters, and states must be specified as global variables. The target memory locations where the variables are stored must lie in the range 0 – 4294967295.
Parameter structures	You cannot tune parameters that are structures.
Pure integer code	Pure integer code is supported. For code generation, if `PurelyIntegerCode` is `'on'`, specify `FixedStep` with a resolution that is greater or equal to 1 microsecond. For example, specify 1.000001, but not 1.0000001. If you do not specify `-tf` *`finalTime`* in the execution command, the target application runs the generated model code indefinitely, ignoring `StopTime`. If you specify `-tf` *`finalTime`* in the execution command: • The *`finalTime`* value represents base rate steps, not seconds. • The maximum value for *`finalTime`*, in ticks, is `MAX_int32_T`.
Variable-size signals	Uploading of variable-size signals is not supported.
Compiler support	`lcc-win64` is not supported.
Address granularity	Target hardware that uses word addresses is not supported.
Portable word sizes	When the production target hardware is word-addressable, the generation of a host-based target application by using portable word sizes is not supported. The production target hardware must support 8-bit, 16-bit, and 32-bit native data types.

Feature	Details
Scope blocks	If your model uses Scope blocks and the amount of data logged during an external mode simulation becomes very large, Simulink might slow down. Instead of Scope blocks, you can use the Simulation Data Inspector or equivalent blocks from the Dashboard library. For more information, see “Decide How to Visualize Simulation Data”.
Scope and Floating Scope blocks, and Scope Viewer	Some signal data types are not supported. The simulation produces a warning.
Floating Scope blocks and Scope Viewer	You cannot use Floating Scope blocks and Scope Viewers to monitor signals in referenced models during external mode simulations. To monitor referenced model signals, enable signal logging and use the Simulation Data Inspector.
Scopes in referenced models	In a model hierarchy, if the top model runs in external mode, scopes in the referenced models are not displayed. To monitor signals within a referenced model, enable signal logging and use the Simulation Data Inspector.
Scope and Display blocks connected to Simulink message blocks	During external mode simulations, Scope and Display blocks connected to Simulink message blocks do not provide displays. To visualize messages, use the Simulation Data Inspector or Dashboard blocks.
Nonzero simulation start time	Nonzero simulation start times are not supported. Use the default value for **Solver** > **Start time**, `0.0`.
Intermediate step values	Some Simulink blocks can generate multiple values in a simulation time step. For example: • A Simulink Function or Function-Call Subsystem block that is invoked multiple times in a time step. • Iterator subsystem blocks that execute multiple times in a time step. For each time step in an external mode simulation, Simulink uploads from the target application only the final values of such blocks. Simulink does not upload the intermediate values generated during the step.
Service code interface	Using XCP communication for tuning parameters and monitoring signals as generated code executes in a target environment is not supported for code generated from component models configured with an ERT-based system target file and service code interface.
File-scoped data	File-scoped data is not supported. For example, data items to which you apply the built-in custom storage class `FileScope`. The simulation produces a warning. For more information about the `FileScope` storage class, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.
Row-major code generation	Code generated with the row-major format is not supported.

<table>
<tr><th>Feature</th><th>Details</th></tr>
<tr><td>Concurrent execution</td><td>On Windows and Linux computers, concurrent execution is supported.

On Mac computers, concurrent execution using the native threads example is not supported. If the External mode and Allow tasks to execute concurrently on target check boxes are selected, you cannot build the target application unless you are using a target hardware support package that supports concurrent execution for external mode simulations.

On a Windows computer, at run time, the base sample time in the target application is quantized to the nearest multiple of the system timer interrupt interval (usually 15.625 ms). Sampling subrates are the expected multiples of the base sample rate. Logged timestamps in the Simulation Data Inspector (SDI) correspond to simulation times rather than wall-clock times. As an example, this table shows the step times for a Simulink model that has a base sample time of 10 ms and a subrate sample time of 20 ms.
<table>
<tr><th colspan="3">Base Rate</th><th colspan="3">Subrate</th></tr>
<tr><td>Step</td><td>Wall-Clock Time (ms)</td><td>Logged Timestamp (ms)</td><td>Step</td><td>Wall-Clock Time (ms)</td><td>Logged Timestamp (ms)</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>15.625</td><td>10</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>31.250</td><td>20</td><td>1</td><td>31.250</td><td>20</td></tr>
<tr><td>3</td><td>46.875</td><td>30</td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>62.500</td><td>40</td><td>2</td><td>62.500</td><td>40</td></tr>
</table></td></tr>
<tr><td>Simulation data export</td><td>You can export logged signal and top-model output data to the MATLAB workspace:

• For logged signal data, select the Signal logging check box.
• For top-model output data, select the Output check box and, from the Format list, select <code>Dataset</code>.

If you want to export the data to a MAT file instead of the MATLAB workspace, also select the Log Dataset data to file check box.

The export of simulation state data is not supported.

Saving the simulation output as a single object is not supported. The simulation produces a warning.</td></tr>
</table>

See Also

More About

- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2

- “Mark Signals for Logging”
- “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35
- “Customize XCP Server Software” on page 60-45
- Simulation Data Inspector

External Websites

- https://www.asam.net/standards/detail/mcd-1-xcp/wiki/

Graphical Controls for XCP External Mode Simulations

In this section...
“Equivalent Controls on Hardware Tab and External Mode Control Panel” on page 60-27 “XCP External Mode Control Panel” on page 60-28 “XCP External Signal & Triggering Dialog Box” on page 60-29

You can control an XCP external mode simulation through:

- The **Hardware** tab on the Simulink Editor toolbar. To display the **Hardware** tab, from the **Apps** tab on the Simulink toolstrip, click **Run on Custom Hardware**.
- The External Mode Control Panel. To open this dialog box, on the **Hardware** tab, in the **Prepare** section, click . Then, under **Signal Monitoring & Tracing**, click **Control Panel**.

Equivalent Controls on Hardware Tab and External Mode Control Panel

This table lists external mode actions that you can perform by using controls that are available in the **Hardware** tab and the External Mode Control Panel.

External Mode Action	Hardware Tab	External Mode Control Panel
Build target application.	**Run on Hardware** > **Build for Monitoring**	N/A
Run application on target hardware.	**Run on Hardware** > **Deploy**	N/A
Connect Simulink to a waiting or running target application.	**Run on Hardware** > **Connect**	**Connect**
Start real-time execution of generated code in the target environment.	**Run on Hardware** > **Start**	**Start Real-Time Code**
Disconnect Simulink from the target environment, but do not stop real-time execution of code.	**Run on Hardware** > **Disconnect**	**Disconnect**
Stop target application execution and disconnect Simulink from the target environment. If you want to reestablish the external mode connection after stopping the target application, you must perform the **Deploy** step.	In **Run on Hardware** section, **Stop** button .	**Stop Real-Time Code**

External Mode Action	Hardware Tab	External Mode Control Panel
Tune batch of block parameters. You can modify values of tunable parameters in Simulink, and then update the parameters in the target application simultaneously. See “Monitor Signal and Tune Parameter”.	In **Prepare** section, **Hold Updates** and **Update All Parameters** buttons.	**Batch download** and **Download**

XCP External Mode Control Panel

The External Mode Control Panel provides these controls for XCP external mode simulations.

Section	Control	Description
Connection and triggering	**Connect**	Connect Simulink to a waiting or running target application. When Simulink is connected to the target application, **Connect** changes to **Disconnect**.
	Start Real-Time Code	Start real-time execution of generated code in the target environment. When the generated code starts executing, the button changes to **Stop Real-Time Code**.
	Arm Trigger	Arm the trigger that you specify through the External Signal & Triggering dialog box. If Simulink is not connected to the target application, the button appears dimmed. If Simulink is connected to the target application and the trigger is armed, the button becomes **Cancel Trigger**.
	Cancel Trigger	Disarm the trigger. Once the trigger is disarmed, the button changes to **Arm Trigger**.
Floating scope	**Enable data uploading**	Selecting the check box arms the trigger for uploading target application data to Floating Scope blocks.
	Duration	Not supported.
Parameter tuning	**Batch download**	To tune a batch of block parameters: **1** In the External Mode Control Panel, select **Batch download**. **2** In the Simulink Editor, modify the required block parameters. **3** When you modify parameters, the External Mode Control Panel displays this message next to **Download**: `Parameter changes pending...` Simulink stores the modified parameters locally. **4** Click **Download**. Simulink downloads the batch of modified parameters to the target application.

Section	Control	Description
	Download	If the **Batch download** check box is not selected, the button appears dimmed.
Configuration	**Signal & Triggering**	Open the External Signal & Triggering dialog box.
	Data Archiving	XCP external mode simulations do not support this button, which appears dimmed.

XCP External Signal & Triggering Dialog Box

For an XCP external mode simulation, you can use the External Signal & Triggering dialog box to:

- Specify the logged signals that you want to monitor.
- Configure a trigger that starts the uploading of data from the target application.

The External Signal & Triggering dialog box provides these controls.

Section	Control	Description
Signal selection	**Select all**	The **Signal selection** view lists top-model and referenced-model signals that are configured for logging. Simulink uploads a logged signal for monitoring only if the signal is selected in the **Signal selection** view. By default, this check box is selected, which enables monitoring of all logged signals.
	Clear all	Deselect the signals that are selected for monitoring. If the **Select all** check box is selected, this button appears dimmed.
	on, **off**	If you select a row, clicking **on** selects the corresponding signal for monitoring. The **Selected** column displays the letter X. Clicking **off** deselects the signal. If the **Select all** check box is selected, these radio buttons appear dimmed.

Section	Control	Description
	Trigger Signal	If you select a row, clicking this button sets the corresponding signal as the trigger for uploading target application data. The **Trigger** column displays the letter `T`. If the signal is already selected, clicking this button deselects the signal. If the configuration parameter `PurelyIntegerCode` is set to `'on'`, the trigger signal must be a non-complex, `int32` scalar. If the configuration parameter `PurelyIntegerCode` is set to `'off'`, the trigger signal must be a non-complex, `double` scalar.
	Go To Signal	To locate a signal in the model, select the corresponding row in the **Signal selection** view and then click this button.
Trigger options	**Source**	Specify how the trigger is armed: • `manual` (default) — Arm the trigger by selecting the **Arm when connecting to target** check box or by clicking the **Arm Trigger** button on the External Mode Control Panel. • `signal` — Use specific characteristics of the trigger signal to start uploading data.

Section	Control	Description
	Mode	Specify trigger rearming after a trigger event is complete: • `normal` (default) — The external mode simulation rearms the trigger after each trigger event. • `one-shot`— The external mode simulation collects only one buffer of data for each time that you arm the trigger.

Section	Control	Description
	Duration	Specify the number of base rate steps for which the external mode simulation uploads data after a trigger event (default is 1000). For example, if **Duration** is set to 1000, and the base (fastest) rate of the model is one second: • For a signal sampled at the base rate, one second (1.0 Hz), the external mode simulation collects 1000 contiguous samples during a trigger event. • For a signal sampled at two seconds (0.5 Hz), the external mode simulation collects 500 samples during a trigger event. If **Send multiple contiguous samples in same packet** is also selected, on the target hardware, the target application: **1** Stores data samples in a buffer. **2** Sends samples from multiple time steps in a packet. The **Duration** value, say `durationFieldValue`, determines the number of samples sent in a packet: `numOfSamplesSent = durationFieldValue/n` where `n` = (base rate)/(sample rate) When the sample rate is the base rate (`n=1`), `numOfSamples` is exactly `durationFieldValue`. For subrates (`n=2, 3, ...`), `numOfSamplesSent` is rounded down to the nearest integer. The target application discards

Section	Control	Description
		buffer samples that are not sent to Simulink.
	Delay	Apply a delay to data collection, which is the time between a trigger event and the start of data collection expressed in base rate steps. You can specify a positive value or zero (default). An XCP external mode simulation does not support a negative delay (pretriggering).
	Arm when connecting to target	This check box is selected by default. Connecting Simulink to the target application arms the trigger. • If the trigger **Source** is `manual`, data uploading begins immediately. • If the trigger **Source** is `signal`, monitoring of the trigger signal begins immediately. Data uploading begins when the trigger signal satisfies trigger conditions (as defined in the **Trigger signal** section). If you clear the check box, you can arm the trigger by clicking the **Arm Trigger** button on the External Mode Control Panel.
	Send multiple contiguous samples in same packet	This check box is not selected by default. If you select the check box, the XCP server uses the XCP DAQ packed mode for streaming signals to Simulink, which improves the ratio of useful data to control data in streamed signals. The target application sends samples from multiple time steps in a packet. The **Duration** value determines the number of samples in the packet.

Section	Control	Description
Trigger signal	**Direction**	To enable the uploading of data from the target application when the trigger signal crosses the threshold value (specified by **Level**), specify the required direction of the signal: • `rising` (default) — Signal must be increasing in value. • `falling` — Signal must be decreasing in value. • `either` — Signal can be increasing or decreasing in value.
	Level	Specify the threshold value that the signal must cross to start the uploading of target application data. Default is 0.
	Hold-off	Specify, in base rate steps, the time between the termination of one trigger event and the rearming of the trigger. Applies only when **Mode** value is normal. Default is 0.

See Also

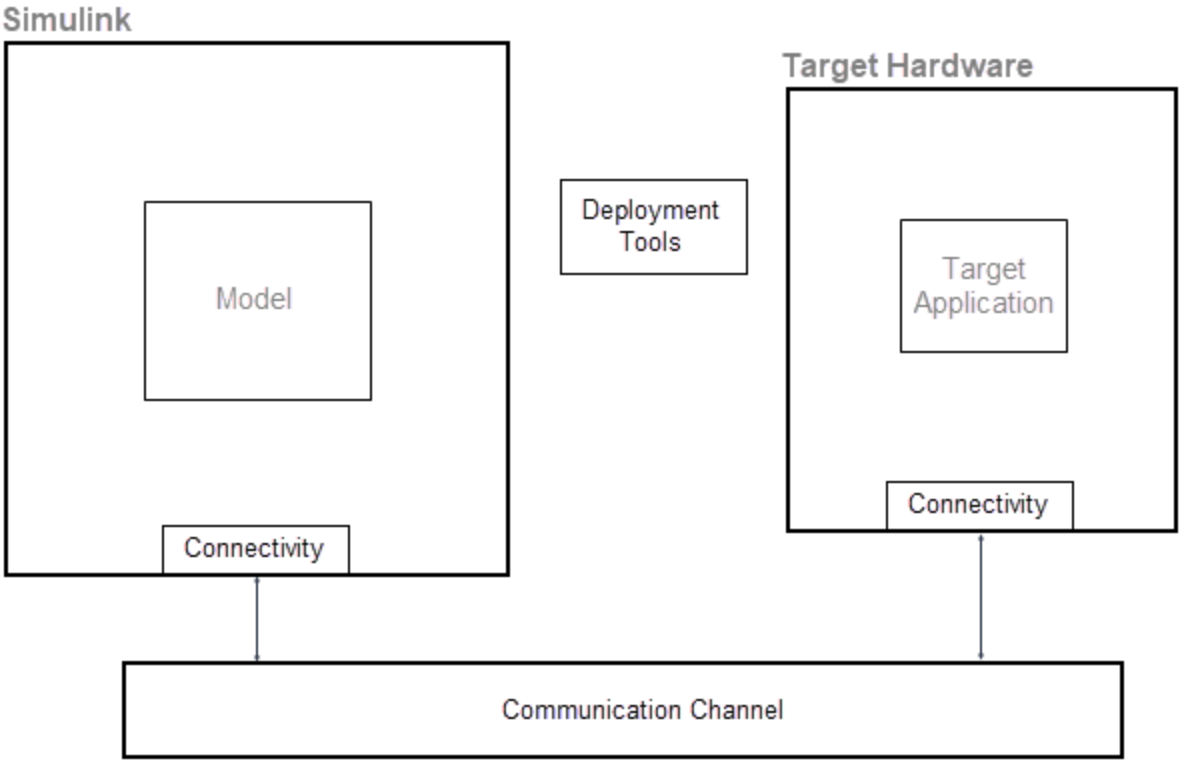
More About

- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Triggered Signal Monitoring for XCP External Mode Simulations” on page 60-12

Set Up External Mode Connectivity Between Simulink and Target Hardware

For external mode simulations, you can use the `target Package` to provide connectivity between Simulink and your target hardware.

This diagram gives an overview of the components of an external mode simulation.



The target package provides classes for the implementation of components. This table lists the main classes.

Component	Class	Purpose
Target hardware	`target.Board`	Provide MATLAB with a description of target hardware.
Deployment tools	`target.SystemCommandExecutionTool`	You can use the classes to:

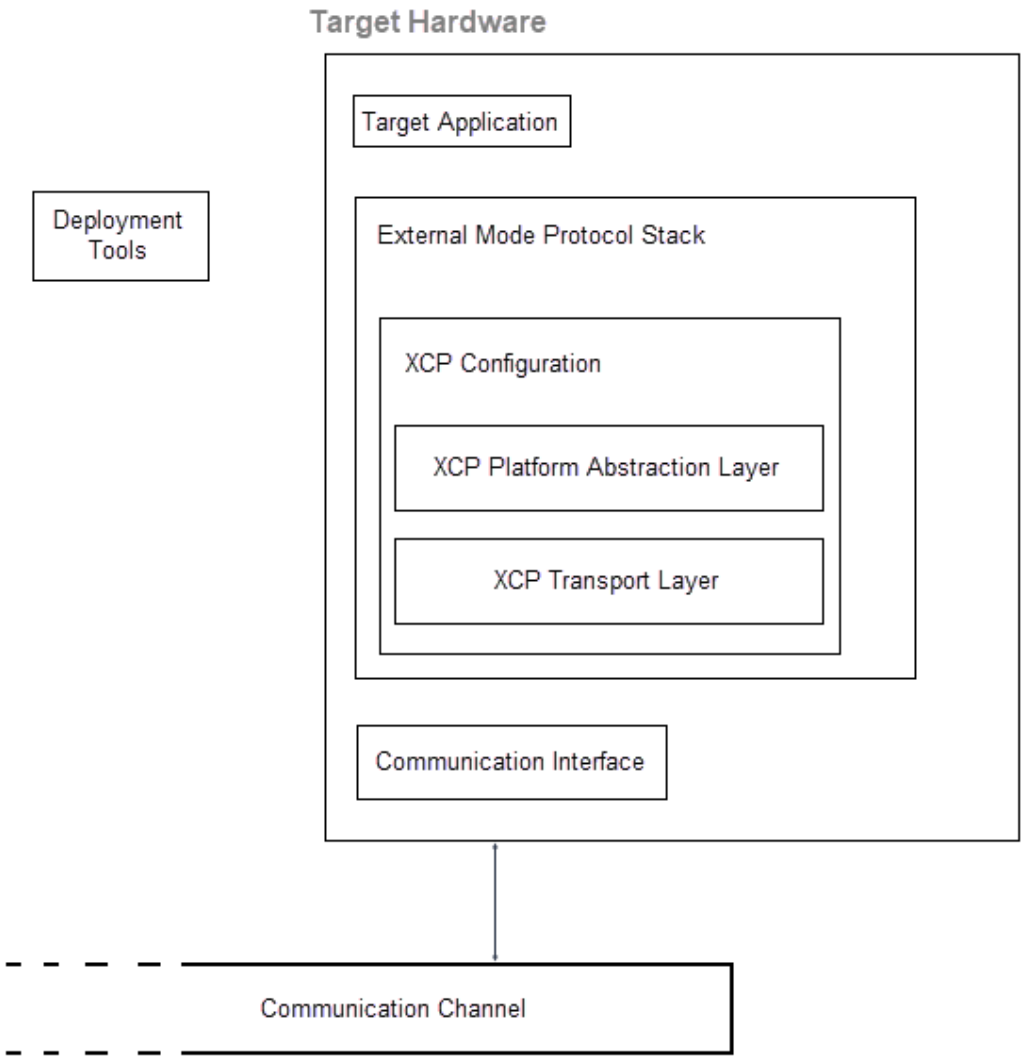
Component	Class	Purpose
	`target.ExecutionService`	• Capture system commands for running the target application from your development computer. • Describe the execution service implementation for the target application. • Provide the MATLAB service interface for the tool that manages the target application execution. To provide **Monitor & Tune**, **Deploy**, **Connect**, and **Start** functionality, the Run on Custom Hardware app requires the use of `target.ExecutionTool`.
	`target.ExecutionTool`	
Connectivity	`target.ExternalMode`	Provide communication protocol for data transfer between Simulink and target hardware.
	`target.CommunicationInterface`	Provide target hardware with details of the communication channel and the rtiostream API on page 90-16 implementation.

Customise Connectivity for XCP External Mode Simulations

For code that is generated by using ERT (`ert.tlc`) and GRT (`grt.tlc`) system target files, you can run external mode simulations that use the XCP communication protocol:

- On your development computer.
- On other target hardware by using support packages.

If your system target file for custom target hardware is derived from the ERT or GRT system target files, use classes from the `target Package` to customise connectivity. For example, `target.ExternalMode` and `target.CommunicationInterface`.



This example shows how you can customise connectivity for XCP-based external mode simulations. To set up connectivity between Simulink and the target hardware:

1 Create board description

Create a `target.Board` object that provides MATLAB with a description of your target hardware. To support code generation, associate the object with a processor.

```
processor = target.get('Processor', 'Intel-x86-64 (Windows64)');
board = target.create('Board', ...
                'Name', 'Example Intel Board', ...
                'Processors', processor);
```

2 **Choose deployment tool**

Use one of these classes:

- `target.ExecutionTool` -- Provides the `getApplicationStatus` method, which is required by the Run on Custom Hardware app for **Monitor & Tune**, **Deploy**, **Connect**, and **Start** functionality.
- `target.SystemCommandExecutionTool` -- The `getApplicationStatus` method is not available with this class. The **Monitor & Tune**, **Deploy**, **Connect**, and **Start** functionality of the Run on Custom Hardware app is not fully supported.

In step 3 and step 4, the availability of a tool that can download the target application onto the target hardware is assumed.

3 **Using `target.ExecutionTool`**

Implement the MATLAB service interface in a class called `MyImplementationClass`. For an example pseudocode implementation, see "Execution Tool Template" on page 60-41.

```
classdef MyImplementationClass < target.ExecutionTool
    properties (Access=private)
    …
    end

    methods
    …
    end

    methods (Access=private)
    …
    end
end
```

Create a `target.ExecutionService` object that contains details about executing the target application. The object describes the tool that is required to run the target application on the target hardware.

```
matlabExecution = ...
                  target.create('ExecutionService', ...
                                'Name', 'My Application Launcher');
matlabExecution.APIImplementation = ...
                  target.create('APIImplementation', ...
                                'Name', 'MyServiceImplementation');
matlabExecution.APIImplementation.BuildDependencies = ...
                  target.create('MATLABDependencies');
matlabExecution.APIImplementation.BuildDependencies.Classes = ...
                               {'MyImplementationClass'};
matlabExecution.APIImplementation.API = ...
                  target.get('API', 'ExecutionTool');
```

Associate the `target.ExecutionService` object with the `target.Board` object.

```
board.Tools.ExecutionTools = matlabExecution;
```

4 **Using `target.SystemCommandExecutionTool`**

Create a `target.SystemCommandExecutionTool` object that captures system commands for running the target application from your development computer.

```
executionTool = target.create('SystemCommandExecutionTool', ...
                 'Name', 'My Execution Tool');
executionTool.StartCommand.String = 'customDownloadTool';
executionTool.StartCommand.Arguments = {'$(EXE)'};
```

Associate the `target.SystemCommandExecutionTool` object with the `target.Board` object.

```
board.Tools.ExecutionTools = executionTool;
```

5 Create communication interface for target hardware

Create a `target.CommunicationInterface` object that provides the target hardware with details of the communication channel and the rtiostream API on page 88-42 implementation.

For the `rtiostream` API, use a `BuildDependencies` object to specify the source files that are compiled with the target application. This example uses the shipped TCP/IP `rtiostream` implementation source file, which enables you to run external mode simulations on your development computer. For your target hardware, provide your own `rtiostream` implementation.

```
buildDependencies = target.create('BuildDependencies', ...
                'SourceFiles', ...
                {fullfile('$(MATLAB_ROOT)', ...
                              'toolbox', 'coder', ...
                              'rtiostream', ...
                              'src', ...
                              'rtiostreamtcpip', ...
                              'rtiostream_tcpip.c')});

apiImplementation = target.create('APIImplementation', ...
                'Name', 'TCP x86 RTIOStream Implementation', ...
                'API', target.get('API', 'rtiostream'), ...
                'BuildDependencies', buildDependencies);

tcpComms = target.create('CommunicationInterface', ...);
                'Name', 'Windows TCP Interface', ...
                'Channel', 'TCPChannel', ...
                'APIImplementations', apiImplementation);
```

To specify a TCP channel, this example uses the name-value argument `'Channel', 'TCPChannel'`. You can specify a serial communication channel by using the name-value argument `'Channel', 'RS232Channel'`.

Associate the `target.CommunicationInterface` object with the `target.Board` object.

```
board.CommunicationInterfaces = tcpComms;
```

6 Specify communication protocol stack

Create an object that provides the communication protocol for data transfer between Simulink and target hardware.

a Use the `target.XCPPlatformAbstraction` class to utilize your custom implementation of the XCP platform abstraction layer. The layer provides a static memory allocator and other platform abstraction functionality.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
                 'Name', 'XCP Platform Abstraction');

xcpPlatformAbstraction.BuildDependencies.Defines = {'XCP_CUSTOM_PLATFORM'};
customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
                 {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
                 {customPlatformAbstractionPath};
```

b Use the `target.XCPTCPIPTransport` class to implement the XCP transport layer, which transmits and receives messages from the communication medium according to ASAM specifications. In this example, the transport protocol is TCP/IP. For serial transport, replace `target.XCPTCPIPTransport` with `target.XCPSerialTransport`.

```
xcpTransport = target.create('XCPTCPIPTransport', ...
                'Name', 'XCP Transport');
```

c Create a `target.XCP` object that represents the XCP protocol stack for the target hardware.

```
xcpConfiguration = target.create('XCP', ...
                'Name', 'XCP Configuration', ...
                'XCPTransport', xcpTransport, ...
                'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

d Create an object that represents the XCP external mode connectivity options that are available in the XCP protocol stack.

```
extModeConnectivity = ...
             target.create('XCPExternalModeConnectivity', ...
             'Name', 'External Mode Connectivity', ...
             'XCP', xcpConfiguration);
```

e Create an external mode protocol stack object.

```
externalMode = target.create('ExternalMode', ...
                'Name', 'External Mode', ...
                'Connectivities', extModeConnectivity);
```

Note You can create the object in a single call to `target.create`.

```
externalMode = target.create('ExternalMode', ...
        'Name', 'External Mode', ...
        'XCPTransportLayer', 'TCP', ...
        'Defines', {'XCP_CUSTOM_PLATFORM'}, ...
        'SourceFiles', {fullfile('pathToImplementationFolder', 'myXCPPlatform.c'}, ...
        'IncludePaths', {'pathToImplementationFolder'});
```

---

f Add the object to the list of communication protocols that the target hardware supports.

```
board.CommunicationProtocolStacks = externalMode;
```

7 Make board object persist across MATLAB sessions

Add the board object to an internal database and make the object persist over MATLAB sessions.

```
target.add(board, 'UserInstall', true);
```

8 Select board for model

In the Configuration Parameters dialog box, set **Hardware board** to `Example Intel Board`.

9 Specify connection between Simulink and target hardware

In the Configuration Parameters dialog box:

- From the **Transport layer** list, select `XCP on TCP/IP`. The transport protocol is in the list because of the connectivity customization.
- In the **Mex-file arguments** field, enter `'localhost' 1 0`. For more information, see MEX-file arguments.

Customize Connectivity for TCP/IP or Serial External Mode Simulations

For TCP/IP or serial external mode simulations, you can customize connectivity through a workflow that:

- Implements transport and communication protocols.
- Specifies the execution tool for the target application by using the `target Package` package.

To set up connectivity between Simulink and the target hardware, use the workflow described in "Customise Connectivity for XCP External Mode Simulations" with these differences:

- After step 1, using the information in "Choose Communication Protocol for Client and Server" on page 60-78 and "Create a Transport Layer for TCP/IP or Serial External Mode Communication" on page 60-94, implement the client and server sides of external mode communication for TCP/IP or serial protocols.
- Do not perform step 5 and step 6.

Execution Tool Template

This section provides a pseudocode example for a `target.ExecutionTool` service interface. The tool starts and tracks an application on the target hardware.

```
classdef MyExecutionTool < target.ExecutionTool

  methods
    function errFlag = startApplication(this)
      % Call "customDownloadTool" to download the application.
      [status, result] = ...
       system(sprintf('customDownloadTool %s', this.Application));
      if status == 0
        errFlag = false;
      else
        disp(result);
        errFlag = true;
      end
    end

    function errFlag = stopApplication(~)
      % Add code here to stop the application, if possible.
      errFlag = false;
    end

    function [status, errFlag] = getApplicationStatus(~)
      % Add code here to return the application status, if known.
      status = target.ApplicationStatus.Unknown;
      errFlag = false;
    end
  end
end
```

See Also

`target Package`

Related Examples

- "External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling" on page 60-2
- "External Mode Simulation by Using XCP Communication" on page 60-8
- "Choose Communication Protocol for Client and Server" on page 60-78
- "Create a Transport Layer for TCP/IP or Serial External Mode Communication" on page 60-94
- "Set Up PIL Connectivity by Using target Package" on page 88-48

External Websites

- https://www.asam.net/standards/detail/mcd-1-xcp/wiki/

Register Custom Launcher

Note The use of a custom launcher to set up connectivity between Simulink and your target hardware is not recommended. Use the `target` package instead. For more information, see “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35

To support the one-click workflow or **Deploy** step for custom hardware, register your custom launcher for the target application.

1. Create an `rtw.connectivity.Config` subclass, which requires `myCustomLauncher`, a launcher that you provide. `myCustomLauncher` is a subclass of `rtw.connectivity.Launcher`. In `myCustomLauncher`, to support one-click functionality, provide an implementation of the `getApplicationStatus` method. For the communicator and builder subclasses, you can use dummy classes.

```
classdef myTargetConnectivityConfig < rtw.connectivity.Config
  methods
    % Constructor
    function this = myTargetConnectivityConfig(componentArgs)

      % Launcher with port 0 argument to start TCP/IP server on free port
      modelName = componentArgs.getModelName();
      bDirInfo = RTW.getBuildDir(modelName);
      exeName = fullfile(bDirInfo.CodeGenFolder, modelName);
      if ispc
        exeName = [exeName, '.exe'];
      end
      launcher = myCustomLauncher(componentArgs);
      launcher.setExe(exeName);

      % Create dummy communicator and builder
      communicator = rtw.connectivity.RtIOStreamHostCommunicator.empty();
      builder = rtw.connectivity.MakefileBuilder.empty();

      % Call super class constructor to register components
      this@rtw.connectivity.Config(componentArgs, builder, launcher, communicator);
    end
  end
end
```

2. To register the implementation in Simulink, create an `sl_customization.m` file.

```
function sl_customization(cm)
  cm.registerTargetInfo(@loc_createConfig);
end
% local function
function config = loc_createConfig
  config = rtw.connectivity.ConfigRegistry;
  config.ConfigClass = 'myTargetConnectivityConfig';

  % Specify compatible model configuration. For example,
  % through SystemTargetFile and TargetHWDeviceType properties.
  config.SystemTargetFile = {'ert.tlc'};
  config.TargetHWDeviceType = { 'ARM Compatible->ARM Cortex' };
end
```

   See “Create sl_customization.m File”.

3. Add the folder containing `sl_customization.m` to the search path and refresh your customizations.

```
addpath(sl_customization_path);
sl_refresh_customizations;
```

See Also

Related Examples

- “Customize XCP Server Software” on page 60-45

Customize XCP Server Software

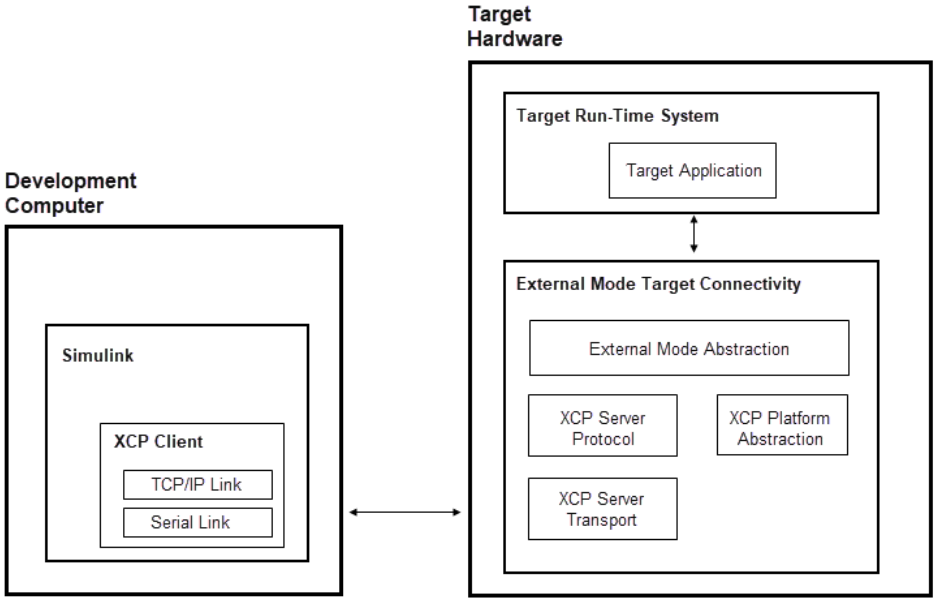
The XCP communication protocol for Simulink external mode simulations is a client-server communication protocol. By default, the software supports XCP external mode simulations:

- On your development computer for code that is generated by using ERT (`ert.tlc`) and GRT (`grt.tlc`) system target files.
- For some support packages on page 60-2.

If your system target file for custom target hardware is derived from the ERT or GRT system target files, you can use supplied APIs to provide XCP target connectivity. XCP external mode limitations on page 60-21 apply.

The external mode target connectivity software comprises:

- External mode abstraction layer
- XCP server protocol layer
- XCP server transport layer
- XCP platform abstraction layer



External Mode Abstraction Layer

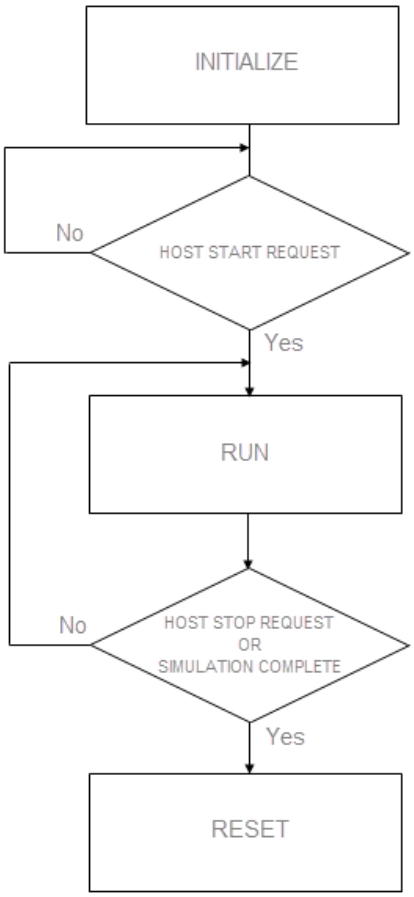
To communicate with Simulink during an external mode simulation, your target application must invoke functions from the external mode abstraction layer.

The parent folder for source code files is:

matlabroot`\toolbox\coder\xcp\src\target\ext_mode`

The layer APIs are declared in `include\ext_mode.h` and implemented in `src\xcp_ext_mode.c`.

To communicate with Simulink, the target application uses services exposed by the external mode abstraction layer. This flow chart shows the required target application steps to establish communication with Simulink.



This tables lists the functions that your target application must invoke at each stage.

Stage	Function	Purpose
INITIALIZE	extmodeParseArgs	Extract external mode command-line arguments.
	*modelName*_initialize	Initialize generated Simulink model code.
	extmodeInit	Initialize external mode target connectivity.
HOST START REQUEST	extmodeWaitForHostRequest	Wait for a start request from development computer, which is created when you click Run button.
RUN	*modelName*_step	Run single step of generated Simulink model code.
	extmodeEvent	For sample time ID of the model, sample signals generated by model step function and pass packet content to transport layer of communication protocol for transmission to development computer.
	extmodeBackgroundRun	Transmit and receive packet from physical communication interface and process packet content in accordance with selected communication protocol.
HOST STOP REQUEST OR SIMULATION COMPLETE	extmodeStopRequested	Check whether a request to stop external mode simulation is received from the model on the development computer. The request is created when you click the Stop button.
	extmodeSimulationComplete	Check whether execution of generated model code is complete.
RESET	*modelName*_terminate	Terminate generated Simulink model code.
	extmodeReset	Reset the external mode target connectivity to initial state.

This pseudo-code example shows how you can implement the various flowchart stages in your target application. During the RUN stage, the external mode run-time environment requires at least two threads:

- A periodic thread that is responsible for the execution of the generated model code.
- A background thread that is responsible for running the external mode communication stack and transmitting and receiving packets.

The pseudo-code simulates multithreading by running `periodicThread` and `extmodeBackgroundRun` sequentially within the same `while()` loop.

```
/*------------- Pseudo-Code Example -------------*/
/* Define periodic thread */
void periodicThread(void)
{
  /* Run model step function */
```

```
  modelName_step();
  /* Notify external mode abstraction layer about periodic event */
  extmodeEvent(PERIODIC_EVENT_ID, currentSimulationTime);
}

/* Main function for target application */
main(int argc, char *argv[])
{
  /*------------- INITIALIZE -------------*/
  /* Parse external mode command line arguments */
  extmodeParseArgs(argc, argv);

  /* Initialize model */
  modelName_initialize();

  /* Initialize external mode target connectivity */
  extmodeInit(extModeInfo, finalSimulationTime);

  /*------------- HOST START REQUEST -------------*/
  /* Wait until a start request is received from development computer */
  extmodeWaitForHostRequest(EXTMODE_WAIT_FOREVER);


  /*------- HOST STOP REQUEST OR SIMULATION COMPLETE -------*/
  /* While simulation is not complete and stop request is not received */
  while (!extmodeSimulationComplete() && !extmodeStopRequested()) {

    /*------------- RUN -------------*/
    periodicThread();
    extmodeBackgroundRun();
  }

  /*------------- RESET -------------*/
  /* Terminate model */
  modelName_terminate();

  /* Reset external mode target connectivity */
  extmodeReset();
  return 0;
}
```

To see code that invokes the functions, complete the example in “External Mode Simulation by Using XCP Communication” on page 60-8 with **System target file** set to `ert.tlc`. Then, from the code generation folder, open `ert_main.c`.

Note If your system target file is `grt.tlc` or GRT-based, the code generator creates `rt_main.c`, and, by default, uses API defined by the internal file `ext_work.h`. To enable the use of `ext_mode.h` API, before building the target application, run:

```
set_param(model, 'TLCOptions', '-aExtModeXCPClassicInterface=0');
```

If your system target file is `ert.tlc` or ERT-based and `GenerateSampleERTMain` is set to `'on'`, the code generator creates `ert_main.c` and, by default, uses API defined by `ext_mode.h`. To enable the use of `ext_work.h` API, before building the target application, run:

```
set_param(model, 'TLCOptions', '-aExtModeXCPClassicInterface=1');
```

XCP Server Protocol Layer

The XCP server protocol layer interprets XCP commands and data according to the Association for Standardisation of Automation and Measuring Systems (ASAM) standard, ASAM MCD-1 XCP.

The source code folder is:

matlabroot`\toolbox\coder\xcp\src\target\slave\protocol\src`

In an external mode simulation, the build process automatically adds the required files to the build information object.

XCP Server Transport Layer

The XCP server transport layer transmits and receives messages from the communication medium according to ASAM specifications.

The source folder is:

matlabroot\toolbox\coder\xcp\src\target\slave\transport\src

In an external mode simulation, the build process automatically adds the required files to the build information object.

XCP Platform Abstraction Layer

The XCP platform abstraction layer provides:

- An XCP driver on page 60-49 for sending and receiving raw data through the physical communication interface.
- The implementation of a static memory allocator on page 60-50.
- Other target hardware-specific functionality.

You can use the `target` package to provide connectivity between Simulink and your target hardware. To specify:

- The XCP driver, use the `target.CommunicationInterface` class.
- Other parts of the XCP platform abstraction layer, use the `target.XCPPlatformAbstraction` class.

If you do not specify the XCP platform abstraction layer, the software uses the default platform abstraction layer, which supports Linux, Windows, and Mac platforms. For more information, see "Customise Connectivity for XCP External Mode Simulations" on page 60-36.

For a customization example, see "Create Custom Abstraction Layer" on page 60-53.

XCP Driver

The XCP driver sends and receives XCP messages through the communication channel. In an external mode simulation, the build process automatically adds the driver files to build information.

The XCP driver is based on the `rtiostream` API. For example, host-based external mode simulations use these `rtiostream` files:

- *matlabroot*\toolbox\coder\rtiostream\src\rtiostreamtcpip\rtiostream_tcpip.c
- *matlabroot*\toolbox\coder\rtiostream\src\rtiostreamserial\rtiostream_serial.c

For details on implementing and testing an `rtiostream` communications channel, see:

- "Communications rtiostream API" on page 88-42

- `rtiostreamtest`

To provide `rtiostream` information:

- If you use the `target` package for connectivity, use the `target.CommunicationInterface` class. See “Customise Connectivity for XCP External Mode Simulations” on page 60-36.
- If you do not use the `target` package, add your `rtiostream` files to a build information object.

For more information, see “Create Custom Abstraction Layer” on page 60-53.

Memory Allocator

The XCP server software requires the dynamic allocation of variable-size, contiguous memory blocks to hold internal data structures. In an external mode simulation, the build process automatically adds memory allocator files to the build information object.

Configure alignment for the memory allocator through the `XCP_MEM_ALIGNMENT` preprocessor macro. For example:

```
#define XCP_MEM_ALIGNMENT 8
```

If you use the `target` package for connectivity, use the `BuildDependencies` property of the `target.XCPPlatformAbstraction` class. For example:

```
xcpPlatformAbstraction.BuildDependencies.Defines = {'XCP_MEM_ALIGNMENT=8'};
```

See “Customise Connectivity for XCP External Mode Simulations” on page 60-36.

Other Platform Abstraction Layer Functionality

This file defines the platform abstraction layer interface:

```
matlabroot\toolbox\coder\xcp\src\target\slave\platform\include\xcp_platform.h
```

This table describes the functionality that you must provide for the XCP server software that you deploy on your target hardware.

Functionality	Details

Mutual exclusion	To access system data structures with *mutual exclusion* support, the XCP driver depends on basic APIs for definition, initialization, lock, and unlock.
	
	To support mutual exclusion for your target hardware, provide a custom implementation.
	
	The default implementation for Linux and Windows supports mutual exclusion.
	
	`#if defined(_WIN32) || defined(__WIN32__) || defined(WIN32)`
	`...`
	`#define XCP_MUTEX_DEFINE(lock) HANDLE lock`
	`#define XCP_MUTEX_INIT(lock) lock = CreateMutex(0, FALSE, 0)`
	`#define XCP_MUTEX_LOCK(lock) WaitForSingleObject((lock), INFINITE)`
	`#define XCP_MUTEX_UNLOCK(lock) ReleaseMutex(lock)`
	
	`#else`
	`...`
	`#include <pthread.h>`
	`#define XCP_MUTEX_DEFINE(lock) pthread_mutex_t lock`
	`#define XCP_MUTEX_INIT(lock) pthread_mutex_init(&(lock), NULL)`
	`#define XCP_MUTEX_LOCK(lock) pthread_mutex_lock(&(lock))`
	`#define XCP_MUTEX_UNLOCK(lock) pthread_mutex_unlock(&(lock))`
	`...`
	`#endif`

Sleep	Provide a sleep API that makes the calling thread sleep until a specified time has elapsed. The default implementation for Linux and Windows systems is:
	
	`#if defined(_WIN32) || defined(__WIN32__) || defined(WIN32)`
	`...`
	`#define XCP_SLEEP(seconds,microseconds)  Sleep((seconds) * 1000 \`
	`+ (((microseconds) + 1000 - 1) / 1000))`
	
	`#else`
	`...`
	`#if _POSIX_C_SOURCE >= 199309L`
	`#define XCP_SLEEP(seconds,microseconds)  do {struct timespec t;\`
	`  t.tv_sec = seconds; t.tv_nsec = microseconds * 1000; nanosleep(&t, NULL);} while(0)`
	`#else`
	`/* nanosleep is not available. Use select instead. */`
	`#define XCP_SLEEP(seconds,microseconds)  do {struct timeval t; t.tv_sec = seconds;\`
	`  t.tv_usec = microseconds; select(0, NULL, NULL, NULL, &t);} while(0)`
	`#endif /* _POSIX_C_SOURCE >= 199309L */`
	`...`
	`#endif`

Logging	To generate diagnostic messages, the XCP server software requires a custom print API that supports logging services provided by the target hardware. If you do not define the `XCP_PRINTF` preprocessor macro, the default implementation is empty.

Functionality	Details

Address conversion	The XCP standard expresses the address of a variable in memory as a 32-bit address with an 8-bit extension.
	
	The XCP client extracts the addresses of signals and parameters of the model by parsing the debug information that the build process creates. The debug information is in DWARF or PDB format.
	
	The XCP client requests access to parameters and signals by sending an XCP command that contains the arguments `addressExtension` and `address`.
	
	When `addressExtension` and `address` information is received by the target hardware, the XCP server must convert the address to the location of the variable in the target hardware memory. The variable location is assigned by the loader and is target-specific.
	
	Use the `XCP_ADDRESS_GET()` macro to specify the conversion logic. The default implementation for Linux and Windows systems is:
	
	`#if defined(__MINGW32__) || defined(__MINGW64__)`
	`#define XCP_ADDRESS_GET(addressExtension, address) \`
	`(uint8_T*) ((uintptr_t) address)`
	`#else`
	`#define XCP_ADDRESS_GET(addressExtension, address) \`
	`(uint8_T*) ((address) + (uint8_T*)&__ImageBase)`
	`#endif`
	
	Currently, only 32-bit hardware architectures are supported, so `addressExtension` is 0.

Set and copy memory	You can specify an optimized version of copy memory and set memory operations.
	
	If you do not define the `XCP_MEMCPY` and `XCP_MEMSET` preprocessor macros, the default implementation specifies the standard C functions, `memcpy` and `memset`.
	
	`#ifndef XCP_MEMCPY`
	`#define XCP_MEMCPY memcpy`
	`#endif`
	
	`#ifndef XCP_MEMSET`
	`#define XCP_MEMSET memset`
	`#endif`

Parsing command-line arguments	If your target hardware does not support the parsing of command-line arguments, define the preprocessor macro `EXTMODE_DISABLE_ARGS_PROCESSING`. To replace the `-w` option, you can use this command to specify that the target application enters and stays in a wait state until it receives a Connect message from Simulink: `set_param(modelName, 'OnTargetWaitForStart', 'on');` The build process provides the required option (`-DON_TARGET_WAIT_FOR_START=1`) to the compiler.

Functionality	Details

Structure packing and alignment	The XCP software stack implementation requires the packing and alignment of some internal C structures. In the platform abstraction layer, provide packing and alignment information by using preprocessor directives that are compiler-specific.
	
	To pack and align C structures, provide these preprocessor directives:
	
	`#define XCP_PRAGMA_PACK_BEGIN(n)`
	`#define XCP_PRAGMA_PACK_END()`
	`#define XCP_ATTRIBUTE_ALIGNED(n)`
	`#define XCP_ATTRIBUTE_PACKED`
	
	For example:
	
	`XCP_PRAGMA_PACK_BEGIN(1)`
	`typedef struct XCP_ATTRIBUTE_PACKED XcpSetDaqListModeCmdPacketFrame {`
	`  uint8_T  PID;  /* Packet identifier, always XCP_PID_SET_DAQ_LIST_MODE */`
	`  uint8_T  mode;  /* DAQ list mode (XCP DAQ Mode bit masks) */`
	`  uint16_T  daqListId; /* DAQ list Id */`
	`  uint16_T  eventId;  /* Event Channel Id */`
	`  uint8_T  prescaler; /* Transmission rate prescaler */`
	`  uint8_T  priority;  /* DAQ list priority (0xFF -> highest) */`
	`}  XcpSetDaqListModeCmdPacketFrame;`
	`XCP_PRAGMA_PACK_END()`
	
	`XCP_PRAGMA_PACK_BEGIN(XCP_MEM_ALIGNMENT)`
	`struct XCP_ATTRIBUTE_ALIGNED(XCP_MEM_ALIGNMENT) xcpFifoEntry {`
	`  struct xcpFifoEntry *next;`
	`  size_t msgFrameSize;  /* in AG units */`
	`};`
	`XCP_PRAGMA_PACK_END()`
	
	For a GCC compiler, use these directives:
	
	`#define XCP_PRAGMA_PACK_BEGIN(n)`
	`#define XCP_PRAGMA_PACK_END()`
	`#define XCP_ATTRIBUTE_ALIGNED(n)  __attribute__((aligned(n)))`
	`#define XCP_ATTRIBUTE_PACKED  __attribute__((packed))`
	
	For a Microsoft Visual C++ compiler, use these directives:
	
	`#define PRAGMA(n)  _Pragma(#n)`
	`#define XCP_PRAGMA_PACK_BEGIN(n)  PRAGMA(pack(push, n))`
	`#define XCP_PRAGMA_PACK_END()  PRAGMA(pack(pop))`
	`#define XCP_ATTRIBUTE_ALIGNED(n)`
	`#define XCP_ATTRIBUTE_PACKED`

Create Custom Abstraction Layer

To create a custom platform abstraction layer:

1. Specify the header file `xcp_platform_custom.h`. This Linux example defines the required functions.

```
#ifndef XCP_PLATFORM_CUSTOM_H
#define XCP_PLATFORM_CUSTOM_H

/* XCP_ADDRESS_GET */
#include <stdint.h>
#define XCP_ADDRESS_GET(addressExtension, address)  (uint8_T*) ((uintptr_t) address)

/* XCP_MUTEX */
#include <pthread.h>
#define XCP_MUTEX_DEFINE(lock)    pthread_mutex_t lock
```

```
#define XCP_MUTEX_INIT(lock)      pthread_mutex_init(&(lock), NULL)
#define XCP_MUTEX_LOCK(lock)      pthread_mutex_lock(&(lock))
#define XCP_MUTEX_UNLOCK(lock)    pthread_mutex_unlock(&(lock))

/* XCP_SLEEP */
#include <sys/time.h>    /* gettimeofday */
#if _POSIX_C_SOURCE >= 199309L
#include <time.h>       /* for nanosleep */
#else
#include <stddef.h>
#include <sys/select.h> /* for select */
#endif

/* _POSIX_C_SOURCE >= 199309L */
#if _POSIX_C_SOURCE >= 199309L
#define XCP_SLEEP(seconds,microseconds)  do {struct timespec t;\
 t.tv_sec = seconds; t.tv_nsec = microseconds * 1000; nanosleep(&t, NULL);} while(0)
#else
/* nanosleep is not available. Use select instead. */
#define XCP_SLEEP(seconds,microseconds)  do {struct timeval t; t.tv_sec = seconds;\
 t.tv_usec = microseconds; select(0, NULL, NULL, NULL, &t);} while(0)
#endif /* _POSIX_C_SOURCE >= 199309L */
#endif
```

2 Register the header file:

- If you use the `target` package for connectivity, use the `BuildDependencies` property of the `target.XCPPlatformAbstraction` class to specify the required information. See “Customise Connectivity for XCP External Mode Simulations” on page 60-36.
- If you do not use the `target` package for connectivity, define a post-code generation command:

```
function myXCPTargetPostCodeGenCommand(buildInfo)

  buildInfo.addDefines('-DXCP_CUSTOM_PLATFORM', 'OPTS');

  % Add my rtiostream
  buildInfo.addSourceFiles(customRtIOStreamFileName, ...
                           customRtIOStreamSrcPath);

  % If the target hardware does not support parsing of command
  % line arguments
    buildInfo.addDefines('-DEXTMODE_DISABLE_ARGS_PROCESSING', 'OPTS');

end
```

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

More About

- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2
- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35

External Websites

- https://www.asam.net/standards/detail/mcd-1-xcp/wiki/

External Mode Simulation with TCP/IP or Serial Communication

Set up and run an external mode simulation that uses a TCP/IP or serial (RS-232) communication channel.

1. Create and configure a simple model.
2. Build the target executable file.
3. Run the target application.
4. Tune parameters.

The example, which uses the GRT target, does not require external hardware. The generated executable file runs:

- On the development computer that hosts Simulink and Simulink Coder.
- As a separate process from MATLAB and Simulink.

Create and Configure Model

In this part of the example, you create a simple model, `ex_extModeExample`. You also create a folder called `ext_mode_example` to store the model and the generated executable.

To create the folder and the model:

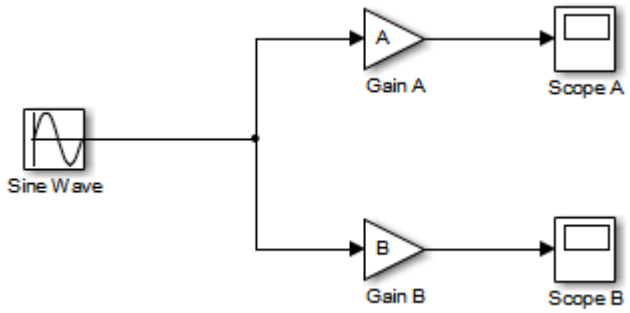
1. From the MATLAB command line, type:

   ```
   mkdir ext_mode_example
   ```

2. Make `ext_mode_example` your working folder:

   ```
   cd ext_mode_example
   ```

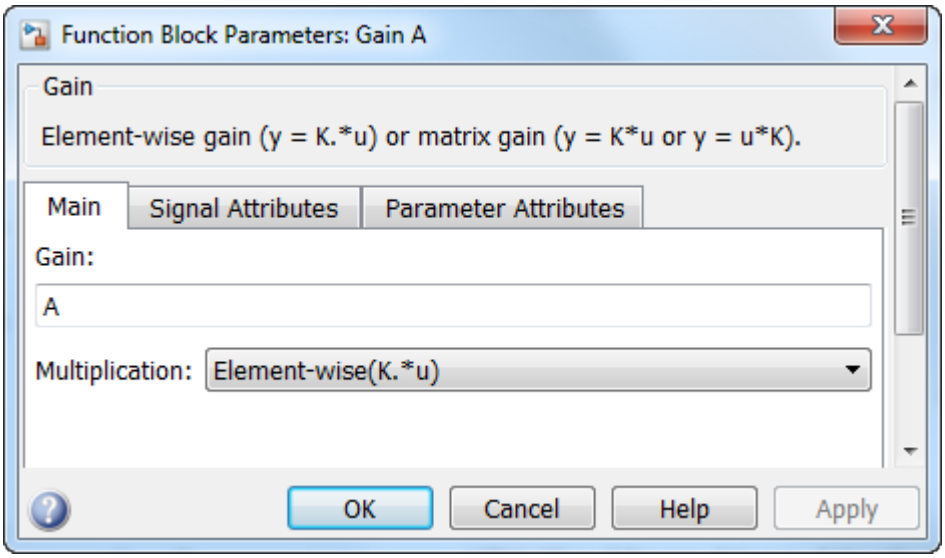
3. Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. Be sure to label the Gain and Scope blocks as shown.



4. Define and assign two MATLAB workspace variables, `A` and `B`:

   ```
   A = 2;
   B = 3;
   ```

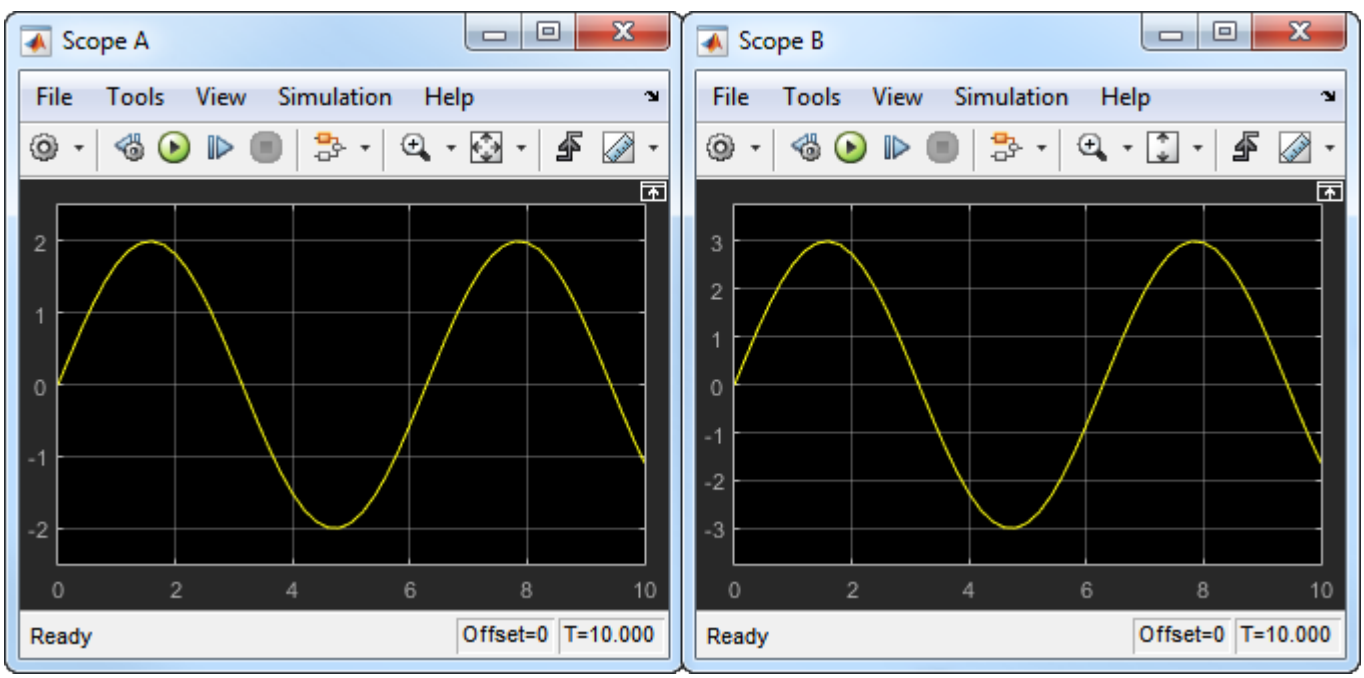
5. Open Gain block A and set its **Gain** parameter to the variable `A`.



6 Open Gain block B and set its **Gain** parameter to the variable `B`.

When the target application is built and connected to Simulink in external mode, you can download new gain values to the executing target application. To do this, you can assign new values to workspace variables `A` and `B`, or edit the values in the block parameters dialog box. For more information, see "Tune Parameters" on page 60-58.

7 Verify operation of the model. Open the Scope blocks and run the model. When `A = 2` and `B = 3`, the output appears as shown.

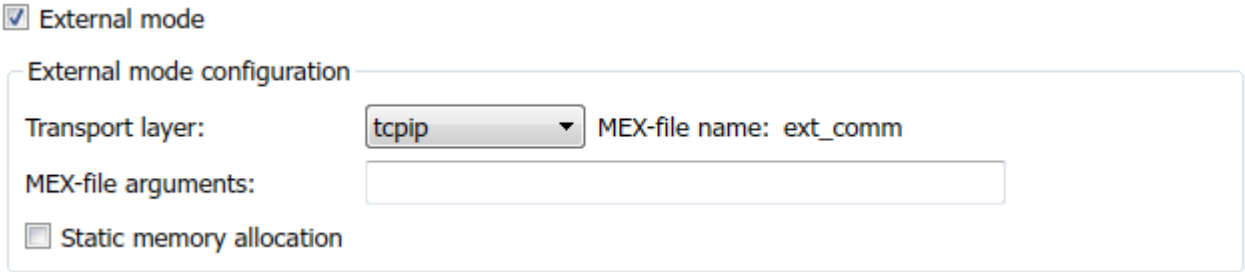


8 Save the model as `ex_extModeExample`.

Build Target Executable

Set up the model and code generation parameters required for an external mode target application. Then, generate code and build the target application.

1. From the **Apps** tab on the Simulink toolstrip, in the **Setup to Run on Hardware** section, click **Run on Custom Hardware**.
2. In the **Hardware** section of the **Hardware** tab, in the **System Target File Description** list, select `Generic Real-Time Target (grt.tlc)`.
3. In the **Prepare** section, click **Hardware Settings**. The Configuration Parameters dialog box opens, displaying **Hardware Implementation** settings that are determined by the system target file.
4. On the **Solver** pane:
   a. In the **Type** field, select `Fixed-step`.
   b. In the **Solver** field, select `discrete (no continuous states)`.
   c. Click **Solver details**. In the **Fixed-step size** field, specify `0.1`. (Otherwise, when you generate code, the Simulink Coder build process produces a warning and supplies a value.)
   d. Click **Apply**.
5. On the **Data Import/Export** pane, clear the **Time** and **Output** check boxes. In this example, data is not logged to the workspace or to a MAT-file. Click **Apply**.
6. On the **Code Generation** > **Optimization** pane, check that **Default parameter behavior** is set to `Tunable`. If you make a change, click **Apply**.
7. On the **Code Generation** > **Interface** pane, in the **Data exchange interface** section, select **External mode**.
8. In the **External mode configuration** section, make sure that the default value `tcpip` is selected for the **Transport layer** parameter.



External mode supports communication via TCP/IP, serial, and custom transport protocols. The **MEX-file name** specifies the name of a MEX-file that implements host-target communication. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Simulink Coder software. You can override this default by supplying other files. If you need to support other transport layers, see “Create a Transport Layer for TCP/IP or Serial External Mode Communication”.

The **MEX-file arguments** field lets you specify arguments, such as a TCP/IP server port number, to be passed to the external interface program. These arguments are specific to the external interface that you are using. For information on setting these arguments, see “MEX-File Optional Arguments for TCP/IP Transport” and “MEX-File Optional Arguments for Serial Transport”.

This example uses the default arguments. Leave the **MEX-file arguments** field blank.

The **Static memory allocation** check box controls how memory is allocated for external mode communication buffers in the target. For this example, do not select the check box. For more information, see “Control Memory Allocation for Communication Buffers in Target” on page 60-61.

9. Click **Apply** to save the external mode settings.
10. Save the model.
11. Select the **Code Generation** pane. Make sure that **Generate code only** is cleared.
12. To generate code and create the target application, in the model window, press **Ctrl+B**. Or, on the **Hardware** tab, in the **Run on Hardware** section, click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Build for Monitoring**.

    The software creates the `ex_extModeExample` executable file in your working folder.

Run Target Application

You now run the `ex_extModeExample` target executable and use Simulink as an interactive front end to the running target application. The executable file is in your working folder. Run the target application and establish communication between Simulink and the target.

Note An external mode program like `ex_extModeExample` is a host-based executable. Its execution is not tied to a real-time operating system (RTOS) or a periodic timer interrupt, and it does not run in real time. The program just runs as fast as possible, and the time units it counts off are simulated time units that do not correspond to time in the world outside the program.

To run the target application:

1. On the **Hardware** tab, in the **Run on Hardware** section:

   a. In the **Stop Time** field, specify `inf`, which makes the model run until the target application receives a stop message from Simulink

   b. Click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Deploy**.

   The target application begins execution, and enters a wait state.
2. Open the Scope blocks in the model. Signals are not visible on the scopes until you begin model execution.
3. On the **Hardware** tab, in the **Run on Hardware** section, click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Connect**. This action initiates a handshake between Simulink and the target application. When Simulink and the target are connected, the **Connect** button changes to **Disconnect**.
4. In the **Run on Hardware** section, click , which starts execution of the generated model code. The two scopes in your model display the outputs of Gain blocks A and B.

You have established communication between Simulink and the running target application. You can now tune block parameters in Simulink and observe the effects of parameter changes.

Tune Parameters

This table describes how you can tune tunable block parameters during a simulation.

Approach	Details
Model Data Editor	To tune parameters through the Model Data Editor: **1** On the **Hardware** tab of the Simulink Editor, in the **Prepare** section, click **Tune Parameters**, which opens the Model Data Editor. **2** If you want to update multiple tunable parameters simultaneously, in the **Prepare** section, toggle on **Hold Updates**. **3** On the **Parameters** tab of the Model Data Editor, in the **Value** column, specify new values for your tunable parameters. **4** Toggle off **Hold Updates** or click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application simultaneously. If **Hold Updates** is off, then immediately after you specify a new value, Simulink downloads the new value to the target application. For more information, see **Model Data Editor**.
Block Parameter dialog box	To tune parameters through the Block Parameter dialog box: **1** For each block that you want to update: **a** Double-click the block, which opens the Block Parameter dialog box. **b** In the parameter fields, specify new parameter values. **c** Click **Apply** or **OK**. **2** If **Hold Updates** is on, toggle off **Hold Updates** or click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application simultaneously. If **Hold Updates** is off, then immediately after you click **Apply** or **OK**, Simulink downloads the new block values to the target application.

Approach	Details
MATLAB workspace	If block parameters are MATLAB workspace variables: **1** In the Command Window, assign new values to the variables. **2** On the **Hardware** tab of the Simulink Editor, in the **Prepare** section, click **Update All Parameters** (**Ctrl+D**). Simulink downloads the new values to the target application. For more information, see “Create and Edit Variables”.

You cannot change the sample time of the Sine Wave block during simulation. Block sample times are part of the structural definition of the model and are part of the generated code. Therefore, if you want to change a block sample time, you must stop the external mode simulation, reset the sample time of the block, and rebuild the executable.

Block parameter tunability during external mode simulation depends on the way that the generated code represents block parameters.

For example, in the Gain A block dialog box, you cannot change the expression `A` in the **Gain** parameter during simulation. Instead, you must change the value of the variable `A` in the base workspace. You cannot change the expression because the generated code does not allocate storage in memory for the **Gain** parameter. Instead, the code creates a field `A` in a structure:

```
/* Parameters (auto storage) */
struct P_ex_extModeExample_T_ {
  real_T A;                           /* Variable: A
                                       */
  real_T B;                           /* Variable: B
                                       */
  real_T SineWave_Amp;                /* Expression: 1
                                       */
  real_T SineWave_Bias;               /* Expression: 0
                                       */
  real_T SineWave_Freq;               /* Expression: 1
                                       */
  real_T SineWave_Phase;              /* Expression: 0
                                       */
};
```

The generated code algorithm uses that field in the code that represents the block Gain A. In this case, the global structure variable `ex_extModeExample_P` uses the type `P_ex_extModeExample_T_`:

```
ex_extModeExample_B.GainA = ex_extModeExample_P.A * rtb_SineWave;
```

When you change the value of `A` in the base workspace, the simulation downloads the new value to the field `A` in the target application.

You can change the expressions in the Sine Wave block parameters during simulation because the generated code creates a field in the global structure `ex_extModeExample_P` to represent each

parameter in the block. When you change an expression in the block dialog box, the simulation first evaluates the new expression. The simulation then downloads the resulting numeric value to the corresponding structure field in the target application.

See “Create Tunable Calibration Parameter in the Generated Code” on page 36-112.

Stop Target Application

To simultaneously disconnect Simulink from the host/target communication and end execution of the target application, on the **Hardware** tab, in the **Run on Hardware** section, click **Stop**.

Control Memory Allocation for Communication Buffers in Target

If you select the **Code Generation > Interface > Static memory allocation** check box (for GRT and ERT targets), the code generator produces code for external mode that uses only static memory allocation (“malloc-free” code). Selecting **Static memory allocation** enables the **Static memory buffer size** parameter. Use this parameter to specify the size of the external mode static memory buffer. The default value is 1,000,000 bytes. If you enter too small a value for your program, the external mode simulation issues an out-of-memory error when it tries to allocate more memory than specified. In such cases, increase the value in the **Static memory buffer size** field and regenerate the code.

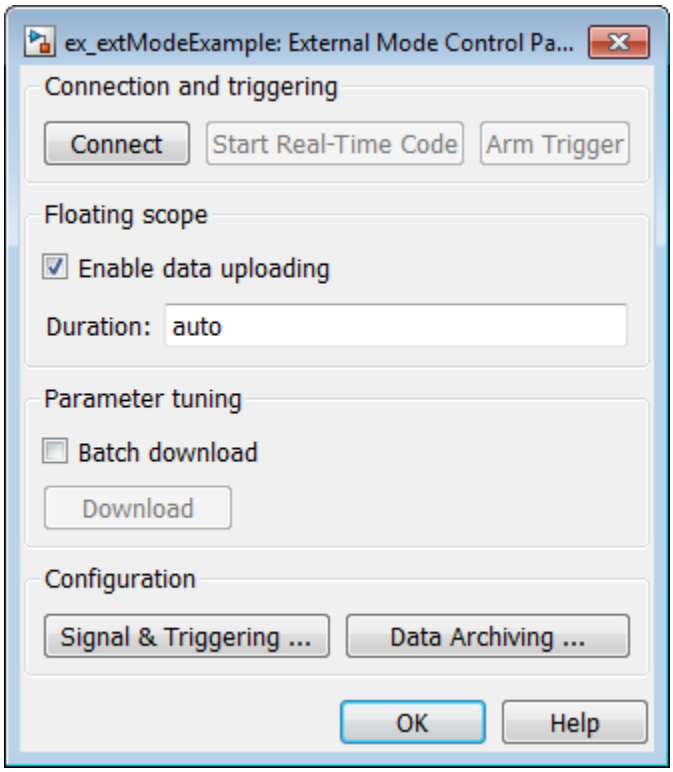
To determine how much memory to allocate, enable verbose mode on the target (by including `OPTS="-DVERBOSE"` on the `make` command line). As it executes, external mode displays the amount of memory it tries to allocate and the amount of memory available to it each time it attempts an allocation. If an allocation fails, you can use this console log to determine the required value for the **Static memory buffer size** field.

TCP/IP or Serial External Mode Control Panel

The External Mode Control Panel provides controls for TCP/IP or serial external mode operations, including:

- “Connect, Start, and Stop” on page 60-62
- “Upload Target Application Signal Data to Host” on page 60-62
- “Download Parameters to Target Application” on page 60-63
- “Configure Host Monitoring of Target Application Signal Data” on page 60-64
- “Configure Host Archiving of Target Application Signal Data” on page 60-70

To open the External Mode Control Panel dialog box, on the **Hardware** tab, in the **Prepare** section, click **Control Panel**.



Connect, Start, and Stop

The External Mode Control Panel performs the same connect/disconnect and start/stop functions found on the **Hardware** tab on the Simulink toolstrip (see “Summary of Graphical Controls for TCP/IP or Serial External Mode Simulation” on page 60-72).

Clicking the **Connect** button connects your model to a waiting or running target application. While you are connected, the button changes to a **Disconnect** button. **Disconnect** disconnects your model from the target environment, but does not halt real-time code running in the target environment.

Clicking the **Start Real-Time Code** button commands the target to start running real-time code. While real-time code is running in the target environment, the button changes to a **Stop Real-Time Code** button. **Stop Real-Time Code** stops target application execution and disconnects your model from the target environment.

Upload Target Application Signal Data to Host

The External Mode Control Panel allows you to trigger and cancel data uploads to the host. The destination for the uploaded data can be a scope block, Display block, To Workspace block, or another block or subsystem listed in “Blocks and Subsystems Compatible with External Mode” on page 60-73.

The **Arm Trigger** and **Cancel Trigger** buttons provide manual control of data uploading to compatible blocks or subsystems, except floating scopes. (For floating scopes, use the **Floating scope** section of the External Mode Control Panel.)

- To trigger data uploading to compatible blocks or subsystems, click the **Arm Trigger** button. The button changes to **Cancel Trigger**.

- To cancel data uploading, click the **Cancel Trigger** button. The button reverts to **Arm Trigger**.

You can trigger data uploads manually or automatically. To configure signals and triggers for data uploads, see “Configure Host Monitoring of Target Application Signal Data” on page 60-64.

A subset of external mode compatible blocks, including Scope, Time Scope, and To Workspace, allow you to log uploaded data to disk. To configure data archiving, see “Configure Host Archiving of Target Application Signal Data” on page 60-70.

The **Floating scope** section of the External Mode Control Panel controls when and for how long data is uploaded to Floating Scope blocks. When used in external mode, floating scopes:

- Do not appear in the External Signal & Triggering dialog box.
- Do not log data to external mode archiving.
- Support manual triggering only.

The **Floating scope** section contains the following parameters:

- **Enable data uploading** option, which functions as an **Arm Trigger** button for floating scopes. When the target is disconnected, the option controls whether to arm the trigger when connecting the floating scopes. When the target is connected, the option acts as a toggle button to arm or cancel the trigger.
  - To trigger data uploading to floating scopes, select **Enable data uploading**.
  - To cancel data uploading to floating scopes, clear **Enable data uploading**.
- **Duration** edit field, which specifies the number of base-rate steps for which external mode logs floating scopes data after a trigger event. By default, it is set to `auto`, which causes the duration value set in the External Signal & Triggering dialog box to be used. The default duration value is 1000 base rate steps.

Download Parameters to Target Application

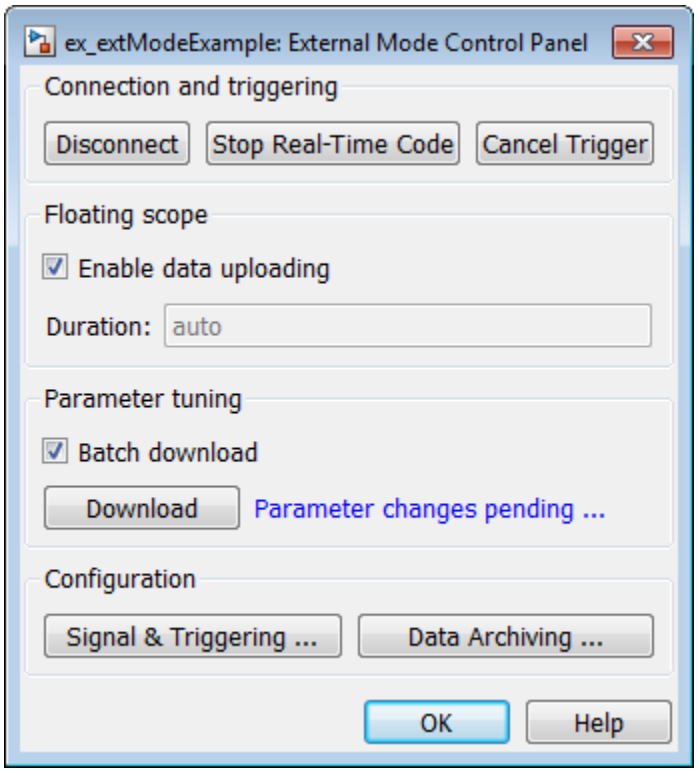
The **Batch download** option on the External Mode Control Panel enables or disables batch parameter changes.

By default, batch download is disabled. If batch download is disabled, when you click **OK** or **Apply**, changes made directly to block parameters by editing block parameter dialog boxes are sent to the target. When you perform an **Update Diagram**, changes to MATLAB workspace variables are sent.

If you select **Batch download**, the **Download** button is enabled. Until you click **Download**, changes made to block parameters are stored locally. When you click **Download**, the changes are sent in a single transmission.

When parameter changes are awaiting batch download, the External Mode Control Panel displays the message `Parameter changes pending...` to the right of the **Download** button. This message remains visible until the Simulink engine receives notification that the new parameters have been installed in the parameter vector of the target system.

The next figure shows the External Mode Control Panel with the **Batch download** option activated and parameter changes pending.



Configure Host Monitoring of Target Application Signal Data

- “Role of Trigger in Signal Data Uploading” on page 60-64
- “Configure Signal Data Uploading” on page 60-65
- “Default Trigger Options” on page 60-66
- “Select Signals to Upload” on page 60-66
- “Configure Trigger Options” on page 60-66
- “Select Trigger Signal” on page 60-68
- “Set Trigger Conditions” on page 60-68
- “Modify Signal and Triggering Options While Connected” on page 60-69

Role of Trigger in Signal Data Uploading

In external mode, uploading target application signal data to the host depends on a *trigger*. The trigger is a set of conditions that must be met for data uploading to begin. The trigger can be armed or not armed.

- When the trigger is armed, the software checks for the trigger conditions that allow data uploading to begin.
- If the trigger is not armed, the software does not check for the trigger conditions and data uploading cannot begin.
- The trigger can be armed automatically, when the host connects to the target, or manually, by clicking the **Arm Trigger** button on the External Mode Control Panel.

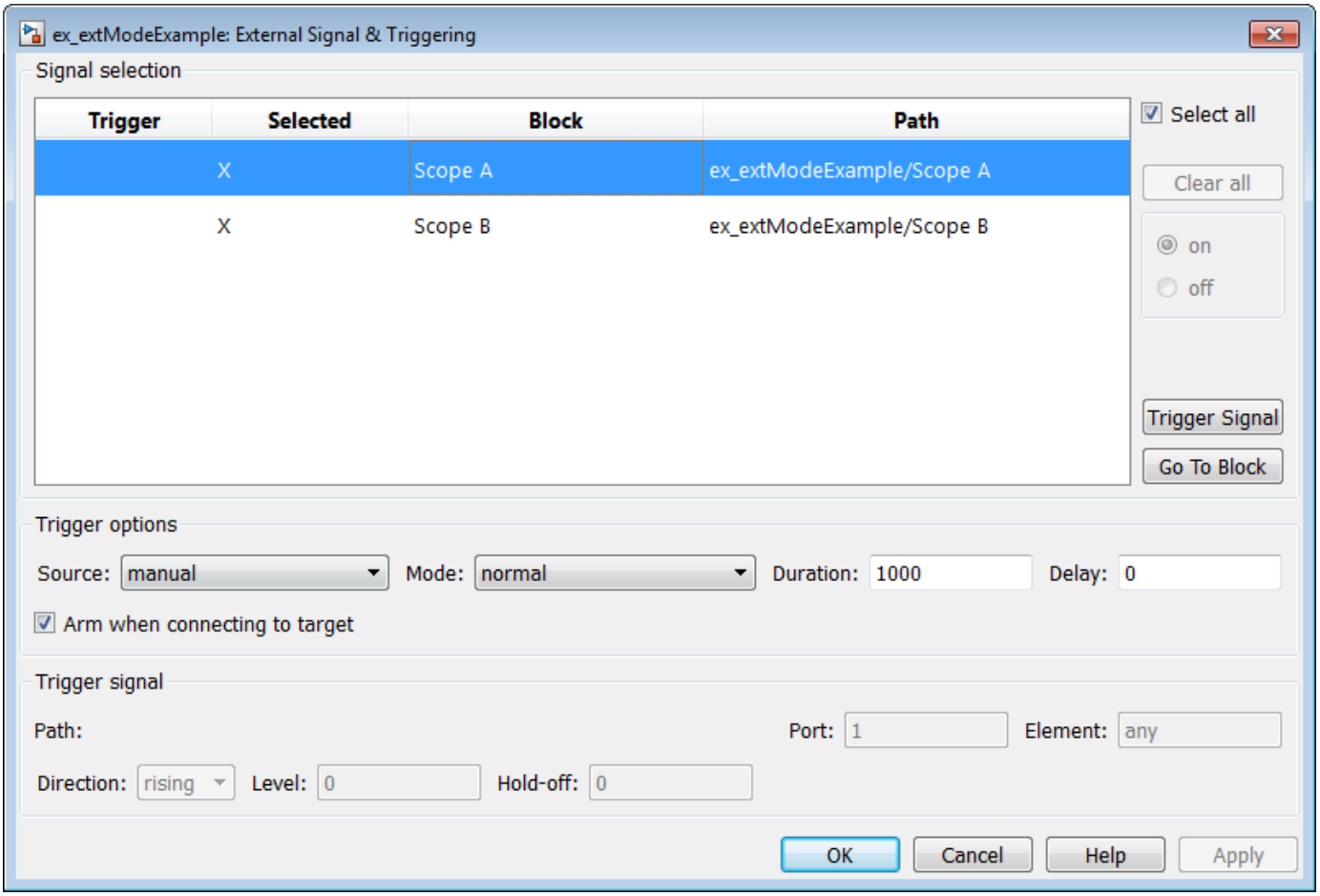
When the trigger is armed and the trigger conditions are met, the trigger fires and data uploading begins.

When data has been collected for a defined duration, the trigger event completes and data uploading stops. The trigger can then rearm, or remain unarmed until you click the **Arm Trigger** button.

To select the target application signals to upload and configure how uploads are triggered, see “Configure Signal Data Uploading” on page 60-65.

Configure Signal Data Uploading

Clicking the **Signal & Triggering** button of the External Mode Control Panel opens the External Signal & Triggering dialog box.



The External Signal & Triggering dialog box displays a list of blocks and subsystems in your model that support external mode signal uploading. For information on which types of blocks are external mode compatible, see “Blocks and Subsystems Compatible with External Mode” on page 60-73.

In the External Signal & Triggering dialog box, you can select the signals that are collected from the target system and viewed in external mode. You can also select a trigger signal, which triggers uploading of data based on meeting certain signal conditions, and define the triggering conditions.

Default Trigger Options

The preceding figure shows the default settings of the External Signal & Triggering dialog box. The default operation of the External Signal & Triggering dialog box simplifies monitoring the target application. If you use the default settings, you do not need to preconfigure signals and triggers. You start the target application and connect the Simulink engine to it. External mode compatible blocks are selected and the trigger is armed. Signal uploading begins immediately upon connection to the target application.

The default configuration of trigger options is:

- `Select all`: on
- `Source`: manual
- `Mode`: normal
- `Duration`: 1000
- `Delay`: 0
- `Arm when connecting to target`: on

Select Signals to Upload

External mode compatible blocks in your model appear in the **Signal selection** list of the External Signal & Triggering dialog box. You use this list to select signals that you want to view. In the **Selected** column, an X appears for each selected block.

The **Select all** check box selects all signals. By default, **Select all** is selected.

If **Select all** is cleared, you can select or clear individual signals using the **on** and **off** options. To select a signal, click its list entry and select the **on** option. To clear a signal, click its list entry and select the **off** option.

The **Clear all** button clears all signals.

Configure Trigger Options

As described in “Role of Trigger in Signal Data Uploading” on page 60-64, signal data uploading depends on a trigger. The trigger defines conditions that must be met for uploading to begin. Also, the trigger must be armed for data uploading to begin. When the trigger is armed and trigger conditions are met, the trigger fires and uploading begins. When data has been collected for a defined duration, the trigger event completes and data uploading stops.

To control when and how signal data is collected (uploaded) from the target system, configure the following **Trigger options** in the External Signal & Triggering dialog box.

- **Source**: `manual` or `signal`. Controls whether a button or a signal triggers data uploading.

  Selecting `manual` directs external mode to use the **Arm Trigger** button on the External Mode Control Panel as the trigger to start uploading data. When you click **Arm Trigger**, data uploading begins.

  Selecting `signal` directs external mode to use a trigger signal as the trigger to start uploading data. When the trigger signal satisfies trigger conditions (that is, the signal crosses the trigger level in the specified direction), a *trigger event* occurs. (Specify trigger conditions in the **Trigger signal** section.) If the trigger is *armed*, external mode monitors for the occurrence of a trigger event. When a trigger event occurs, data uploading begins.

- **Mode**: `normal` or `one-shot`. Controls whether the trigger rearms after a trigger event completes.

  In `normal` mode, external mode automatically rearms the trigger after each trigger event. The next data upload begins when the trigger fires.

  In `one-shot` mode, external mode collects only one buffer of data each time you arm the trigger.

  For more information on the **Mode** setting, see "Configure Host Archiving of Target Application Signal Data" on page 60-70.
- **Duration**: Specifies the number of base rate steps for which external mode uploads data after a trigger event (default is 1000). For example, if **Duration** is set to 1000, and the base (fastest) rate of the model is one second:

  - For a signal sampled at the base rate, one second (1.0 Hz), external mode collects 1000 contiguous samples during a trigger event.
  - For a signal sampled at two seconds (0.5 Hz), external mode collects 500 samples during a trigger event.
- **Delay**: Specifies a delay to be applied to data collection. The delay represents the amount of time that elapses between a trigger event and the start of data collection. The delay is expressed in base rate steps. It can be positive or negative (default is 0). A negative delay corresponds to pretriggering. When the delay is negative, data from the time preceding the trigger event is collected and uploaded.
- **Arm when connecting to target**: Selected or cleared. Whether a button or a signal triggers data uploading (as defined by **Source**), the trigger must be armed to allow data uploading to begin.

  If you select this option, connecting to the target arms the trigger.

  - If the trigger **Source** is `manual`, data uploading begins immediately.
  - If the trigger **Source** is `signal`, monitoring of the trigger signal begins immediately. Data uploading begins when the trigger signal satisfies trigger conditions (as defined in the **Trigger signal** section).

  If you clear **Arm when connecting to target**, manually arm the trigger by clicking the **Arm Trigger** button on the External Mode Control Panel.

When simulating in external mode, each rate in the model creates a buffer on the target. Each entry in the buffer is big enough to hold all of the data required of every signal in that rate for one time step (time plus data plus external mode indices identifying the signal). The number of entries in the circular buffer is determined by the external mode trigger **Duration** parameter (`ExtModeTrigDuration`). The memory allocated on the target for buffering signals is proportional to the **Duration** and the number of signals uploading. The **Duration** also provides an indication of the number of base rate steps with log data after a trigger event in external mode.

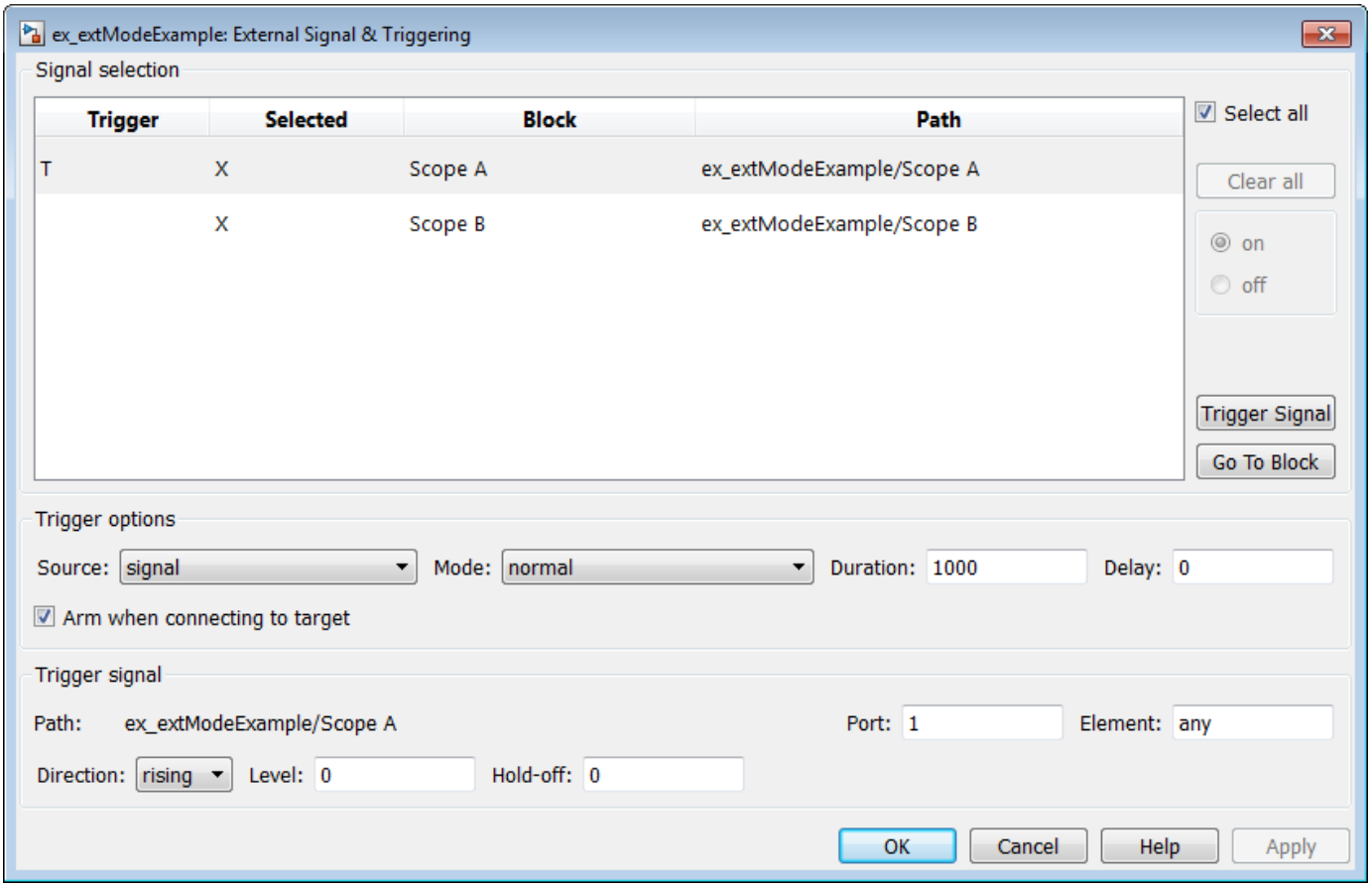
The **Duration** value specifies the number of contiguous points of data to be collected in each buffer of data. You should enter a **Duration** value equal to the number of continuous sample points that you need to collect rather than relying on a series of buffers to be continuous. If you enter a value less than the total number of sample points, you may lose sample points during the time spent transferring values from the data buffer to the MATLAB workspace. The Simulink software maintains point continuity only within one buffer. Between buffers, because of transfer time, some samples may be omitted.

The **Duration** value can affect the **Limit data points to last** value of Scope and To Workspace blocks. The number of sample points that the blocks save to the MATLAB workspace is the smaller of the two values. To set the number of sample points that the blocks save, clear **Limit data points to last**. Then, use **Duration** to specify the number of sample points saved.

Select Trigger Signal

You can designate one signal as a trigger signal. To select a trigger signal, from the **Source** menu in the **Trigger options** section, select `signal`. This action enables the parameters in the **Trigger signal** section. Then, select a signal in the **Signal selection** list, and click the **Trigger Signal** button.

When you select a signal to be a trigger, a `T` appears in the **Trigger** column of the **Signal selection** list. In the next figure, the `Scope A` signal is the trigger. `Scope B` is also selected for viewing, as indicated by the `X` in the **Selected** column.



After selecting the trigger signal, you can use the **Trigger signal** section to define the trigger conditions and set the trigger signal **Port** and **Element** parameters.

Set Trigger Conditions

Use the **Trigger signal** section of the External Signal & Triggering dialog box to set trigger conditions and attributes. **Trigger signal** parameters are enabled only when the trigger parameter **Source** is set to `signal` in the **Trigger options** section.

By default, any element of the first input port of a specified trigger block can cause the trigger to fire (that is, Port 1, any element). You can modify this behavior by adjusting the **Port** and **Element** values in the **Trigger signal** section. The **Port** field accepts a number or the keyword `last`. The **Element** field accepts a number or the keywords `any` or `last`.

In the **Trigger signal** section, you also define the conditions under which a trigger event occurs.

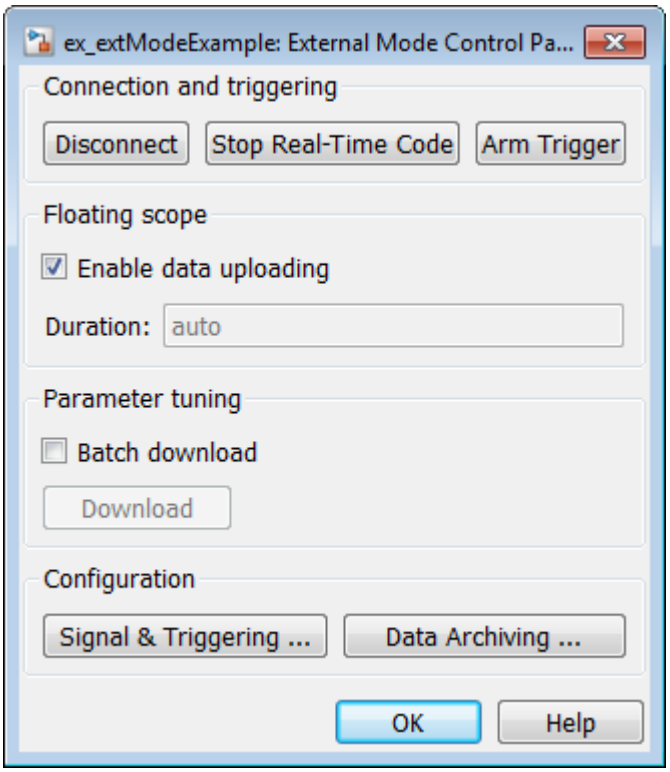
- **Direction**: `rising`, `falling`, or `either`. The direction in which the signal must be traveling when it crosses the threshold value. The default is `rising`.
- **Level**: A value indicating the threshold the signal must cross in a designated direction to fire the trigger. By default, the level is 0.
- **Hold-off**: Applies only to `normal` mode. Expressed in base rate steps, **Hold-off** is the time between the termination of one trigger event and the rearming of the trigger.

Modify Signal and Triggering Options While Connected

After you configure signal data uploading, and connect Simulink to a running target executable, you can modify signal and triggering options without disconnecting from the target.

If the trigger is armed (for example, if the trigger option **Arm when connecting to the target** is selected, which is the default), the External Signal & Triggering dialog box cannot be modified. To modify signal and triggering options:

1. Open the External Mode Control Panel.
2. Click **Cancel Trigger**. Triggering and display of uploaded data stops.
3. Open the External Signal & Triggering dialog box and modify signal and trigger options as required. For example, in the **Signal selection** section, you can enable or disable a scope, and in the **Trigger options** section, change the trigger **Mode**, for example, from `normal` to `one-shot`.
4. Click **Arm Trigger**. Triggering and display of uploaded data resumes, with your modifications.



Configure Host Archiving of Target Application Signal Data

In external mode, you can use the Simulink Scope and To Workspace blocks to archive data to disk.

To understand how the archiving features work, consider the handling of data when archiving is not enabled. There are two cases, one-shot mode and normal mode.

- In one-shot mode, after a trigger event occurs, each selected block writes its data to the workspace, as it would at the end of a simulation. If another one-shot is triggered, the existing workspace data is overwritten.
- In normal mode, external mode automatically rearms the trigger after each trigger event. Consequently, you can think of normal mode as a series of one-shots. Each one-shot in this series, except for the last, is referred to as an *intermediate result*. Because the trigger can fire at any time, writing intermediate results to the workspace can result in unpredictable overwriting of the workspace variables. For this reason, the default behavior is to write only the results from the final one-shot to the workspace. The intermediate results are discarded. If you know that enough time exists between triggers for inspection of the intermediate results, you can override the default behavior by selecting the **Write intermediate results to workspace** option. This option does not protect the workspace data from being overwritten by subsequent triggers.

If you use a Simulink Scope block to archive data to disk, open the Scope parameters dialog box and select the option **Log data to workspace**. The option is required for these reasons:

- The data is first transferred from the scope data buffer to the MATLAB workspace, before being written to a MAT-file.

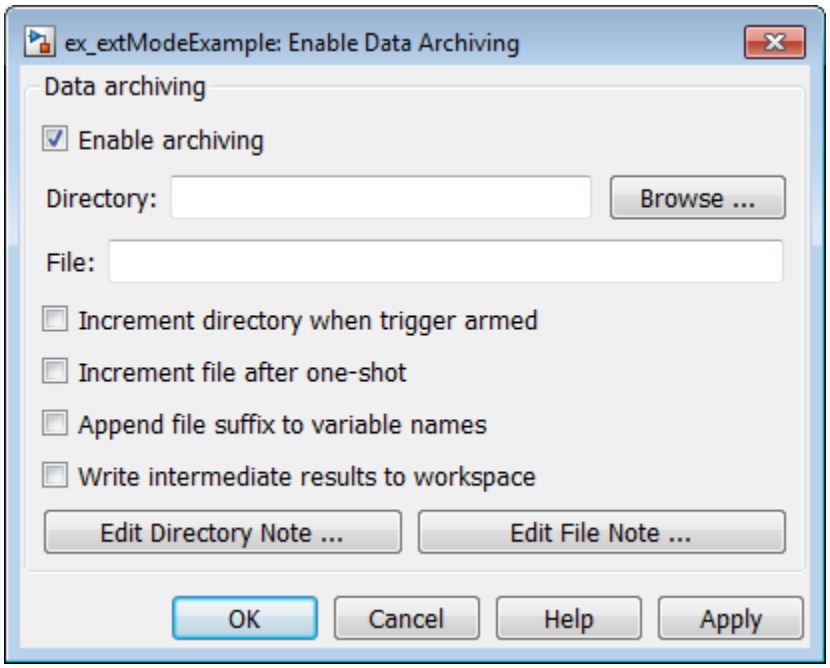
- The **Variable name** entered in the Scope parameters dialog box is the same as the one in the MATLAB workspace and the MAT-file. Enabling the data to be saved enables a variable named with the **Variable name** parameter to be saved to a MAT-file.

Note If you do not select the Scope block option **Log data to workspace**, the MAT-files for data logging are created, but they are empty.

The Enable Data Archiving dialog box supports:

- Folder notes
- File notes
- Automated data archiving

On the External Mode Control Panel, click the **Data Archiving** button to open the Enable Data Archiving dialog box. If your model is connected to the target environment, disconnect it while you configure data archiving. To enable the other controls in the dialog box, select **Enable archiving**.



These operations are supported by the Enable Data Archiving dialog box.

Folder Notes

To add annotations for a collection of related data files in a folder, in the Enable Data Archiving dialog box, click **Edit Directory Note**. The MATLAB editor opens. Place comments that you want saved to a file in the specified folder in this window. By default, the comments are saved to the folder last written to by data archiving.

File Notes

To add annotations for an individual data file, in the Enable Data Archiving dialog box, click **Edit File Note**. A file finder window opens, which by default is set to the last file to which you have written.

Selecting a MAT-file opens an edit window. In this window, add or edit comments that you want saved with your individual MAT-file.

Automated Data Archiving

To configure automatic writing of logging results to disk, optionally including intermediate results, use the **Enable archiving** option and the controls it enables . The dialog box provides the following related controls:

- **Directory**: Specifies the folder in which data is saved. If you select **Increment directory when trigger armed**, external mode appends a suffix.
- **File**: Specifies the name of the file in which data is saved. If you select **Increment file after one-shot**, external mode appends a suffix.
- **Increment directory when trigger armed**: Each time that you click the **Arm Trigger** button, external mode uses a different folder for writing log files. The folders are named incrementally, for example, `dirname1`, `dirname2`, and so on.
- **Increment file after one-shot**: New data buffers are saved in incremental files: `filename1`, `filename2`, and so on. File incrementing happens automatically in normal mode.
- **Append file suffix to variable names**: Whenever external mode increments file names, each file contains variables with identical names. Selecting **Append file suffix to variable name** results in each file containing unique variable names. For example, external mode saves a variable named `xdata` in incremental files (`file_1`, `file_2`, and so on) as `xdata_1`, `xdata_2`, and so on. This approach supports loading the MAT-files into the workspace and comparing variables at the MATLAB command prompt. Without the unique names, each instance of `xdata` would overwrite the previous one in the MATLAB workspace.
- **Write intermediate results to workspace**: If you want the Simulink Coder software to write intermediate results to the workspace, select this option.

Summary of Graphical Controls for TCP/IP or Serial External Mode Simulation

You can control a TCP/IP or serial external mode simulation through:

- The **Hardware** tab on the Simulink toolstrip. To display the **Hardware** tab, from the **Apps** tab on the Simulink toolstrip, click **Run on Custom Hardware**.
- The External Mode Control Panel. To open this dialog box, in Simulink Editor, on the **Hardware** tab, in the **Prepare** section, click ▾. Then under **Signal Monitoring & Tracing**, click **Control Panel**.

This table lists external mode actions that you can perform by using controls that are available in the **Hardware** tab and the External Mode Control Panel.

External Mode Action	Hardware Tab	External Mode Control Panel
Build target application.	**Run on Hardware** > **Build for Monitoring**	N/A

External Mode Action	Hardware Tab	External Mode Control Panel
Run application on target hardware.	**Run on Hardware** > **Deploy**	N/A
Connect Simulink to a waiting or running target application.	**Run on Hardware** > **Connect** When Simulink is connected to the target application, **Connect** appears dimmed and **Disconnect** is active.	**Connect** When Simulink is connected to the target application, **Connect** changes to **Disconnect**.
Start real-time execution of generated code in the target environment.	**Run on Hardware** > **Start**	**Start Real-Time Code** When the generated code starts executing, the button changes to **Stop Real-Time Code**.
Disconnect Simulink from the target environment, but do not stop real-time execution of code.	**Run on Hardware** > **Disconnect** When Simulink is disconnected from the target application, **Disconnect** appears dimmed and **Connect** is active.	**Disconnect**
Stop target application execution and disconnect Simulink from the target environment.	In **Run on Hardware** section, **Stop** button .	**Stop Real-Time Code**
Tune batch of block parameters.	In **Prepare** section, **Hold Updates** and **Update All Parameters** buttons.	**Batch download** and **Download**

Blocks and Subsystems Compatible with External Mode

- “Compatible Blocks” on page 60-73
- “Signal Viewing Subsystems” on page 60-74
- “Supported Blocks for Data Archiving” on page 60-75

Compatible Blocks

In external mode, you can use the following types of blocks to receive and view signals uploaded from the target application:

- Floating Scope and Scope blocks
- Spectrum Analyzer and Time Scope blocks from the DSP System Toolbox product
- Display blocks
- To Workspace blocks
- User-written S-Function blocks

  An external mode method is built into the S-function API. This method allows user-written blocks to support external mode. See *matlabroot*/simulink/include/simstruc.h.
- Record, XY Graph blocks

You can designate certain subsystems as Signal Viewing Subsystems and use them to receive and view signals uploaded from the target application. See “Signal Viewing Subsystems” on page 60-74 for more information.

You select external mode compatible blocks and subsystems, and arm the trigger, by using the External Signal & Triggering dialog box. By default, such blocks in a model are selected, and a manual trigger is set to be armed when connected to the target application.

Signal Viewing Subsystems

A Signal Viewing Subsystem is an atomic subsystem that encapsulates processing and viewing of signals received from the target system. A Signal Viewing Subsystem runs only on the host, and does not generate code in the target system. Signal Viewing Subsystems run in normal, accelerator, rapid accelerator, and external simulation modes.

Note Signal Viewing Subsystems are inactive if placed inside a SIL or PIL component, such as a top model in SIL or PIL mode, a Model block in SIL or PIL mode, or a SIL or PIL block. However, a SIL or PIL component can feed a Signal Viewing Subsystem running in a supported mode.

Signal Viewing Subsystems are useful in situations where you want to process or condition signals before viewing or logging them, but you do not want to perform these tasks on the target system. By using a Signal Viewing Subsystem, you can generate smaller and more efficient code on the target system.

Like other external mode compatible blocks, Signal Viewing Subsystems are displayed in the External Signal & Triggering dialog box.

To declare a subsystem to be a Signal Viewing Subsystem:

1. In the Block Parameters dialog box, select the **Treat as atomic unit** option.

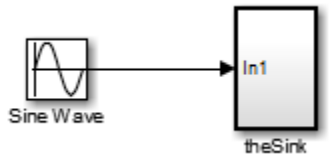
   For more information on atomic subsystems, see “Simulink Modeling Components”“Simulink Modeling Components”.
2. To turn the SimViewingDevice property on, use the `set_param` command:

   ```
   set_param('blockname', 'SimViewingDevice','on')
   ```

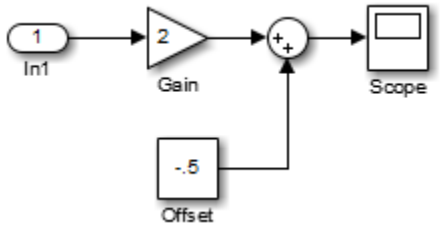
   `'blockname'` is the name of the subsystem.
3. Make sure the subsystem meets the following requirements:

- It must be a pure Sink block. That is, it must not contain Outport blocks or Data Store blocks. It can contain Goto blocks only if the corresponding From blocks are contained within the subsystem boundaries.
- It must not have continuous states.

The following model, `sink_examp`, contains an atomic subsystem, `theSink`.



The subsystem `theSink` applies a gain and an offset to its input signal and displays it on a Scope block.



If `theSink` is declared as a Signal Viewing Subsystem, the generated target application includes only the code for the Sine Wave block. If `theSink` is selected and armed in the External Signal & Triggering dialog box, the target application uploads the sine wave signal to `theSink` during simulation. You can then modify the parameters of the blocks within `theSink` and observe the uploaded signal.

If `theSink` were not declared as a Signal Viewing Subsystem, its Gain, Constant, and Sum blocks would run as subsystem code on the target system. The Sine Wave signal would be uploaded to the Simulink engine after being processed by these blocks, and viewed on `sink_examp/theSink/Scope2`. Processing demands on the target system would be increased by the additional signal processing, and by the downloading of changes in block parameters from the host.

Supported Blocks for Data Archiving

In external mode, you can use the following types of blocks to archive data to disk:

- Scope blocks
- To Workspace blocks

You configure data archiving by using the Enable Data Archiving dialog box, as described in “Configure Host Archiving of Target Application Signal Data” on page 60-70.

External Mode Mechanism for Downloading Tunable Parameters

- “Download Mechanism” on page 60-76

- “Inlined and Tunable Parameters” on page 60-77

Depending on the setting of the **Default parameter behavior** option when the target application is generated, there are differences in the way parameter updates are handled. “Download Mechanism” on page 60-76 describes the operation of external mode communication with **Default parameter behavior** set to `Tunable`. “Inlined and Tunable Parameters” on page 60-77 describes the operation of external mode with **Default parameter behavior** set to `Inlined`.

Download Mechanism

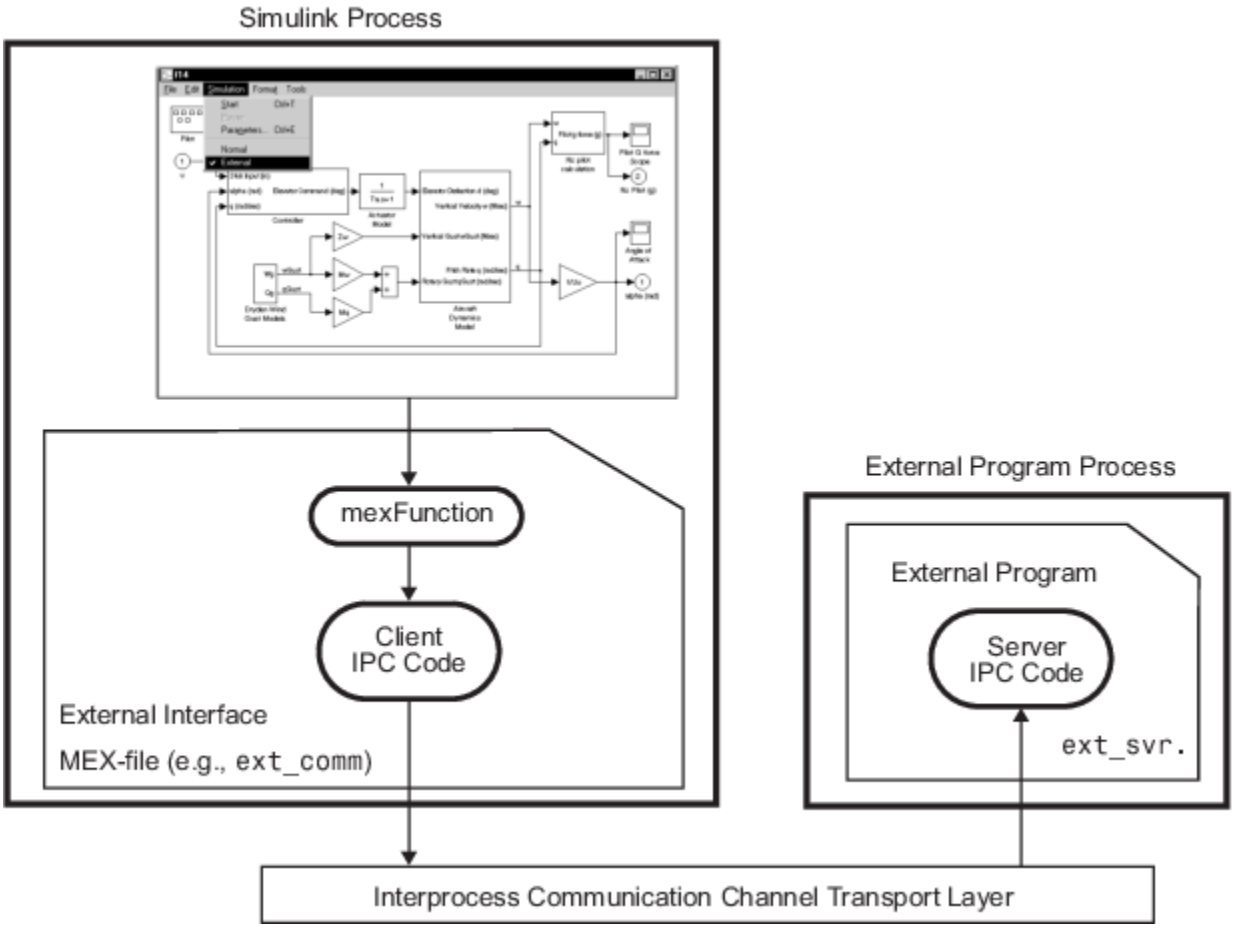
In external mode, the Simulink engine does not simulate the system represented by the block diagram. By default, when external mode is enabled, the Simulink engine downloads parameters to the target system. After the initial download, the engine remains in a waiting mode until you change parameters in the block diagram or until the engine receives data from the target.

When you change a parameter in the block diagram, the Simulink engine calls the external interface MEX-file, passing new parameter values (along with other information) as arguments. The external interface MEX-file contains code that implements one side of the interprocess communication (IPC) channel. This channel connects the Simulink process (where the MEX-file executes) to the process that is executing the external program. The MEX-file transfers the new parameter values by using this channel to the external program.

The other side of the communication channel is implemented within the external program. This side writes the new parameter values into the target's parameter structure (*`model`*`_P`).

The Simulink side initiates the parameter download operation by sending a message containing parameter information to the external program. In the terminology of client/server computing, the Simulink side is the client and the external program is the server. The two processes can be remote, or they can be local. Where the client and server are remote, a protocol such as TCP/IP is used to transfer data. Where the client and server are local, a serial connection or shared memory can be used to transfer data.

The next figure shows this relationship. The Simulink engine calls the external interface MEX-file whenever you change parameters in the block diagram. The MEX-file then downloads the parameters to the external program by using the communication channel.



External Mode Architecture

Inlined and Tunable Parameters

By default, parameters (except those listed in “TCP/IP and Serial External Mode Limitations” on page 60-90) in an external mode program are tunable; that is, you can change them by using the download mechanism described in this section.

If you set **Default parameter behavior** to `Inlined` (on the **Optimization** pane of the Configuration Parameters dialog box), the Simulink Coder code generator embeds the numerical values of model parameters (constants), instead of symbolic parameter names, in the generated code. Inlining parameters generates smaller and more efficient code. However, inlined parameters, because they effectively become constants, are not tunable.

The Simulink Coder software lets you improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters that are important to your application. When you inline parameters, you can use `Simulink.Parameter` objects to remove individual parameters from inlining and declare them to be tunable. In addition, you can use these objects to control how parameters are represented in the generated code.

For more information on tunable parameters, see “Create Tunable Calibration Parameter in the Generated Code” on page 36-112.

Automatic Parameter Uploading on Host/Target Connection

Each time the Simulink engine connects to a target application that was generated with **Default parameter behavior** set to `Inlined`, the target application uploads the current value of its tunable parameters to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure synchronizes the host and target with respect to parameter values. If the tunable parameters are stored in the model workspace or a data dictionary, the values are not uploaded and Simulink produces a warning.

Workspace variables required by the model must be initialized at the time of host/target connection. Otherwise the uploading cannot proceed and an error results. Once the connection is made, these variables are updated to reflect the current parameter values on the target system.

Automatic parameter uploading takes place only if the target application was generated with **Default parameter behavior** set to `Inlined`. “Download Mechanism” on page 60-76 describes the operation of external mode communication with **Default parameter behavior** set to `Tunable`.

Choose Communication Protocol for Client and Server

- “Introduction” on page 60-78
- “Using the TCP/IP Implementation” on page 60-78
- “Using the Serial Implementation” on page 60-81
- “Run the External Program” on page 60-82
- “Implement an External Mode Protocol Layer” on page 60-84

Introduction

The Simulink Coder product provides code to implement both the client and server side of external mode communication using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by the Simulink Coder product with the generated code, provided that your target system supports TCP/IP. If not, use or customize the serial transport layer option provided.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

Using the TCP/IP Implementation

You can use TCP/IP-based client/server implementation of external mode with real-time programs on The Open Group UNIX or PC systems. For help in customizing external mode transport layers, see “Create a Transport Layer for TCP/IP or Serial External Mode Communication”.

To use Simulink external mode over TCP/IP:

- Make sure that the external interface MEX-file for your target's TCP/IP transport is specified.

  Targets provided by MathWorks specify the name of the external interface MEX-file in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`. The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The TCP/IP default is `ext_comm`.

To specify a TCP/IP transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
%end function
```

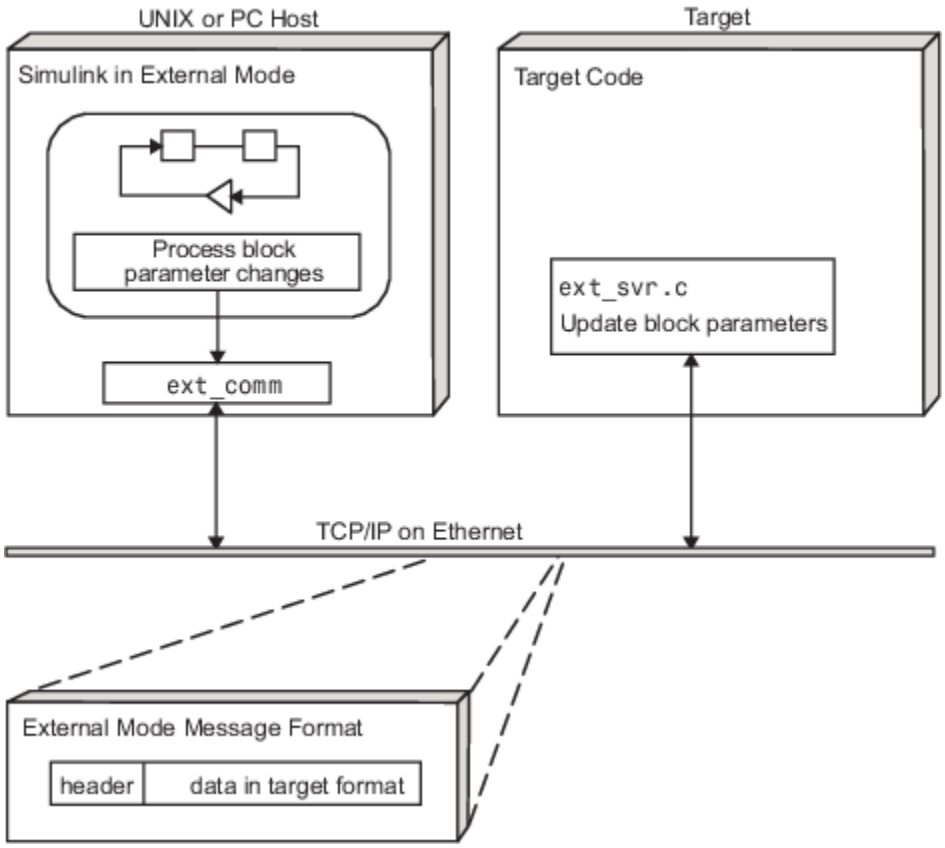
- *`stf`*`.tlc` is the name of the system target file for which you are registering the transport (for example, `'mytarget.tlc'`)
- *`transport`* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'tcpip'`)
- *`mexfile`* is the name of the transport's associated external interface MEX-file (for example, `'ext_comm'`)

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
  cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
                           'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the source files for the TCP/IP server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

The next figure shows the structure of the TCP/IP-based implementation.



TCP/IP-Based Client/Server Implementation for External Mode

MEX-File Optional Arguments for TCP/IP Transport

In the External Target Interface dialog box, you can specify optional arguments that are passed to the external mode interface MEX-file for communicating with executing targets.

- Target network name: the network name of the computer running the external program. By default, this is the computer on which the Simulink product is running, for example, `'myComputer'`. You can also use the IP address, for example, `'148.27.151.12'`.
- Verbosity level: controls the level of detail of the information displayed during the data transfer. The value is either `0` or `1` and has the following meaning:

  `0` — No information
  `1` — Detailed information
- TCP/IP server port number: The default value is `17725`. You can change the port number to a value between `256` and `65535` to avoid a port conflict.

The arguments are positional and must be specified in the following order:

```
<TargetNetworkName> <VerbosityLevel> <ServerPortNumber>
```

For example, if you want to specify the verbosity level (the second argument), then you must also specify the target network name (the first argument). Arguments can be delimited by white space or commas. For example:

```
'148.27.151.12' 1 30000
```

You can specify command-line options to the external program when you launch it. See “Run the External Program” on page 60-82.

Using the Serial Implementation

Controlling host/target communication on a serial channel is similar to controlling host/target communication on a TCP/IP channel.

To use Simulink external mode over a serial channel, you must:

- Make sure that the external interface MEX-file for your target's serial transport is specified.

  Targets provided by MathWorks specify the name of the external interface MEX-file in *matlabroot*/toolbox/simulink/simulink/extmode_transports.m. The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The serial default is `serial`.

  To specify a serial transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

  ```
  function sl_customization(cm)
    cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
  %end function
  ```

  - *stf*.tlc is the name of the system target file for which you are registering the transport (for example, `'mytarget.tlc'`)
  - *transport* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'serial'`)
  - *mexfile* is the name of the transport's associated external interface MEX-file (for example, `'ext_serial_win32_comm'`)

  You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

  ```
  function sl_customization(cm)
    cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
    cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
                             'ext_serial_win32_comm', 'Level1');
  %end function
  ```

- Be sure that the template makefile is configured to link the source files for the serial server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

MEX-File Optional Arguments for Serial Transport

In the **MEX-file arguments** field of the **Interface** pane of the Configuration Parameters dialog box, you can specify arguments that are passed to the external mode interface MEX-file for communicating with the executing targets. For serial transport, the optional arguments to `ext_serial_win32_comm` are as follows:

- Verbosity level: This argument controls the level of detail of the information displayed during data transfer. The value of this argument is:

  - `0` (no information), or
  - `1` (detailed information)
- Serial port ID: The port ID of the host, specified as an integer or character vector. For example, specify the port ID of a USB to serial converter as `'/dev/ttyusb0'`. Simulink Coder prefixes integer port IDs with `\\.\COM` on Windows and by `/dev/ttyS` on Unix.

  When you start the target application using a serial connection, you must specify the port ID to use to connect it to the host. Do this by including the `-port` command-line option. For example:

  ```
  mytarget.exe -port 2 -w
  ```
- Baud rate: Specify an integer value. The default value is `57600`.

The MEX-file options arguments are positional and must be specified in the following order:

```
<VerbosityLevel> <SerialPortID> <BaudRate>
```

For example, if you want to specify the serial port ID (the second argument), then you must also specify the verbosity level (the first argument). Arguments can be delimited by white space or commas. For example:

```
1 '/dev/ttyusb0' 57600
```

When you launch the external program, you can specify command-line options.

Run the External Program

Before you can use the Simulink product in external mode, the external program must be running.

If the target application is executing on the same machine as the host and communication is through a loopback serial cable, the target's port ID must differ from that of the host (as specified in the **MEX-file arguments** edit field).

To run the external program, you type a command of the form:

```
model -opt1 ... -optN
```

`model` is the name of the external program and *`-opt1 ... -optN`* are options. (See “Command-Line Options for the External Program” on page 60-83.) In the examples in this section, the name of the external program is `ext_example`.

Running the External Program in the Windows Environment

In the Windows environment, you can run the external programs in either of the following ways:

- Open a Command Prompt window. At the command prompt, type the name of the target executable, followed by possible options, such as:

  ```
  ext_example -tf inf -w
  ```
- Alternatively, you can launch the target executable from the MATLAB Command Window. The command must be preceded by an exclamation point (`!`) and followed by an ampersand (&), as in the following example:

  ```
  !ext_example -tf inf -w &
  ```

The ampersand (&) causes the operating system to spawn another process to run the target executable. If you do not include the ampersand, the program still runs, but you cannot enter commands at the MATLAB command prompt or manually terminate the executable.

Running the External Program in the UNIX Environment

In the UNIX environment, you can run the external programs in either of the following ways:

- Open an `Xterm` window. At the command prompt, type the name of the target executable, followed by possible options, such as:

  ```
  ./ext_example -tf inf -w
  ```

- Alternatively, you can launch the target executable from the MATLAB Command Window. You must run it in the background so that you can still access the Simulink environment. The command must be preceded by an exclamation point (`!`), dot slash (./ indicating the current directory), and followed by an ampersand (&), as in the following example:

  ```
  !./ext_example -tf inf -w &
  ```

  The ampersand (&) causes the operating system to spawn another process to run the target executable.

Command-Line Options for the External Program

External mode target executables generated by the Simulink Coder code generator support the following command-line options:

- `-tf n`

  The `-tf` option overrides the stop time set in the Simulink model. The argument `n` specifies the number of seconds the program will run. The value `inf` directs the model to run indefinitely. In this case, the model code runs until the target application receives a stop message from the Simulink engine.

  The following example sets the stop time to 10 seconds.

  ```
  ext_example -tf 10
  ```

When integer-only ERT targets are built and executed in external mode, the stop time parameter (`-tf`) is interpreted by the target as the number of base rate steps rather than the number of seconds to execute.

- `-w`

  Instructs the target application to enter a wait state until it receives a message from the host. At this point, the target is running, but not executing the model code. The start message is sent when you select **Start Real-Time Code** from the **Simulation** menu or click the **Start Real-Time Code** button in the External Mode Control Panel.

  Use the `-w` option if you want to view target application execution data from time step 0, or if you want to modify parameters before the target application begins execution of model code.

- `-port n`

  Specifies the TCP/IP port number or the serial port ID, `n`, for the target application. The port number of the target application must match that of the host for TCP/IP transport. The port number depends on the type of transport.

- For TCP/IP transport: Port number is an integer between `256` and `65535`, with the default value being `17725`. The selected port must not be used by another TCP/IP service that runs on the target hardware.
- For serial transport: Port ID is an integer or a character vector. For example, specify the port ID of a USB to serial converter as `'/dev/ttyusb0'`

- `-baud r`

  Specified as an integer, this option is only available for serial transport.

Implement an External Mode Protocol Layer

If you want to implement your own transport layer for external mode communication, you must modify certain code modules provided by the Simulink Coder product and create a new external interface MEX-file. See “Create a Transport Layer for TCP/IP or Serial External Mode Communication”.

Use External Mode Programmatically

You can run external mode simulations from the MATLAB command line or programmatically in scripts. Use the `get_param` and `set_param` commands to retrieve and set the values of model simulation command-line parameters, such as `SimulationMode` and `SimulationCommand`, and external mode command-line parameters, such as `ExtModeCommand` and `ExtModeTrigType`.

The following model simulation commands assume that a Simulink model is open and that you have loaded a target application to which the model will connect using external mode.

1. Change the Simulink model to external mode:

   ```
   set_param(gcs,'SimulationMode','external')
   ```

2. Connect the open model to the loaded target application:

   ```
   set_param(gcs,'SimulationCommand','connect')
   ```

3. Start running the target application:

   ```
   set_param(gcs,'SimulationCommand','start')
   ```

4. Stop running the target application:

   ```
   set_param(gcs,'SimulationCommand','stop')
   ```

5. Disconnect the target application from the model:

   ```
   set_param(gcs,'SimulationCommand','disconnect')
   ```

The `set_param` commands that use the `'SimulationCommand'` argument are asynchronous. If you run the commands successively from a script, each command starts without waiting for the previous command to complete. To check that each command is complete, in the script, use the `get_param` command with the `'SimulationStatus'` argument. For example, for steps 1 to 3, specify these commands in the script:

```
set_param(gcs,'SimulationMode','external');
set_param(gcs,'SimulationCommand','connect');

isExternalSimulationActive = false;
while ~isExternalSimulationActive
```

```
    simStatus = get_param(gcs, 'SimulationStatus');
    isExternalSimulationActive = strcmp(simStatus, 'external');
end
set_param(gcs,'SimulationCommand','start');
```

For more information, see “Run Simulations Programmatically”.

The Diagnostic Viewer displays error messages produced by the `get_param` and `set_param` commands.

To tune a workspace parameter, change its value at the command prompt. If the workspace parameter is a `Simulink.Parameter` object, assign the new value to the `Value` property.

```
myVariable = 5.23;
myParamObj.Value = 5.23;
```

To download the workspace parameter in external mode, you update the model diagram. The following model simulation command initiates a model update:

```
set_param(gcs,'SimulationCommand','update')
```

To trigger or cancel data uploading to scopes, use the `ExtModeCommand` values `armFloating` and `cancelFloating`, or `armWired` and `cancelWired`. For example, to trigger and then cancel data uploading to wired (nonfloating) scopes:

```
set_param(gcs,'ExtModeCommand','armWired')
set_param(gcs,'ExtModeCommand','cancelWired')
```

The next table lists external mode command-line parameters that you can use in `get_param` and `set_param` commands. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to External Mode dialog box equivalents. For external mode parameters that are equivalent to **Interface** pane options in the Configuration Parameters dialog box, see “Model Configuration Parameters: Code Generation Interface”.

External Mode Command-Line Parameters

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeAddSuffixToVar` **`off`**, `on`	Enable Data Archiving: **Append file suffix to variable names** check box	Increment variable names for each incremented filename.
`ExtModeArchiveDirName` *`character vector`*	Enable Data Archiving: **Directory** text field	Save data in specified folder.
`ExtModeArchiveFileName` *`character vector`*	Enable Data Archiving: **File** text field	Save data in specified file.
`ExtModeArchiveMode` *`character vector`* - **`off`**, `auto`, `manual`	Enable Data Archiving: **Enable archiving** check box	Activate automated data archiving features. To specify `manual`, run `set_param(gcs, 'ExtModeArchiveMode', 'manual')`. Note that if you specify `auto`, `ExtModeAutoIncOneShot` is set to `on`.
`ExtModeArmWhenConnect` `off`, **`on`**	External Signal & Triggering: **Arm when connecting to target** check box	Arm the trigger as soon as the Simulink Coder software connects to the target.
`ExtModeAutoIncOneShot` **`off`**, `on`	Enable Data Archiving: **Increment file after one-shot** check box	Save new data buffers in incremental files.
`ExtModeAutoUpdateStatusClock` (Windows platforms only) `off`, **`on`**	Not available	Continuously upload and display target time on the model window status bar.
`ExtModeBatchMode` **`off`**, `on`	External Mode Control Panel: **Batch download** check box	Enable or disable downloading of parameters in batch mode.
`ExtModeChangesPending` **`off`**, `on`	Not available	When `ExtModeBatchMode` is enabled, indicates whether parameters remain in the queue of parameters to be downloaded to the target.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeCommand` *`character vector`* - `armFloating`, `armWired`, `cancelFloating`, `cancelWired`	• `armFloating` and `cancelFloating` are equivalent to selecting and clearing External Mode Control Panel check box **Floating scope** > **Enable data uploading** • `armWired` and `cancelWired` are equivalent to External Mode Control Panel buttons **Arm Trigger** and **Cancel Trigger**	Issue an external mode command to the target application.
`ExtModeConnected` **`off`**, `on`	External Mode Control Panel: **Connect/Disconnect** button	Indicate the state of the connection with the target application.
`ExtModeEnableFloating` `off`, **`on`**	External Mode Control Panel: **Enable data uploading** check box	Enable or disable the arming and canceling of triggers when a connection is established with floating scopes.
`ExtModeIncDirWhenArm` **`off`**, `on`	Enable Data Archiving: **Increment directory when trigger armed** check box	Write log files to incremental folders each time the trigger is armed.
`ExtModeLogAll` `off`, **`on`**	External Signal & Triggering: **Select all** check box	Upload available signals from the target to the host.
`ExtModeParamChangesPending` **`off`**, `on`	Not available	When the Simulink Coder software is connected to the target and `ExtModeBatchMode` is enabled, indicates whether parameters remain in the queue of parameters to be downloaded to the target. More efficient than `ExtModeChangesPending`, because it checks for a connection to the target.
`ExtModeSkipDownloadWhenConnect` **`off`**, `on`	Not available	Connect to the target application without downloading parameters.
`ExtModeTrigDelay` *`integer`* (**`0`**)	External Signal & Triggering: **Delay** text field	Specify the amount of time (expressed in base rate steps) that elapses between a trigger occurrence and the start of data collection.
`ExtModeTrigDirection` *`character vector`* - **`rising`**, `falling`, `either`	External Signal & Triggering: **Direction** menu	Specify the direction in which the signal must be traveling when it crosses the threshold value.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigDuration` *`integer`* (**`1000`**)	External Signal & Triggering: **Duration** text field	Specify the number of base rate steps for which external mode is to log data after a trigger event.
`ExtModeTrigDurationFloating` *`character vector`* - *`integer`* (**`auto`**)	External Mode Control Panel: **Duration** text field	Specify the duration for floating scopes. If `auto` is specified, the value of `ExtModeTrigDuration` is used.
`ExtModeTrigElement` *`character vector`* - *`integer`*, **`any`**, `last`	External Signal & Triggering: **Element** text field	Specify the elements of the input port of the specified trigger block that can cause the trigger to fire.
`ExtModeTrigHoldOff` *`integer`* (**`0`**)	External Signal & Triggering: **Hold-off** text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.
`ExtModeTrigLevel` *`integer`* (**`0`**)	External Signal & Triggering: **Level** text field	Specify the threshold value the trigger signal must cross to fire the trigger.
`ExtModeTrigMode` *`character vector`* - **`normal`**, `oneshot`	External Signal & Triggering: **Mode** menu	Specify whether the trigger is to rearm automatically after each trigger event or whether only one buffer of data is to be collected each time the trigger is armed.
`ExtModeTrigPort` *`character vector`* - *`integer`* (**`1`**), `last`	External Signal & Triggering: **Port** text field	Specify the input port of the specified trigger block for which elements can cause the trigger to fire.
`ExtModeTrigType` *`character vector`* - **`manual`**, `signal`	External Signal & Triggering: **Source** menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.
`ExtModeUploadStatus` *`character vector`* - `inactive`, `armed`, `uploading`	Not available	Return the status of the external mode upload mechanism — inactive, armed, or uploading.
`ExtModeWriteAllDataToWs` **`off`**, `on`	Enable Data Archiving: **Write intermediate results to workspace** check box	Write intermediate results to the workspace.

Animate Stateflow Charts in External Mode

If you have Stateflow, you can animate a chart in external mode. In external mode, you can animate states in a chart, and view test point signals in a floating scope or signal viewer.

- “Animate States During Simulation in External Mode” on page 60-89
- “View Test Point Data in Floating Scopes and Signal Viewers” on page 60-89

Animate States During Simulation in External Mode

To animate states in a chart in external mode:

1. Load the chart you want to animate to the target machine.
2. In the Configuration Parameters dialog box, select the **External mode** check box.
3. Open the External Mode Control Panel dialog box.
4. Click **Signal & Triggering**.
5. In the External Signal & Triggering dialog box, set these parameters.

In:	Select:
Signal selection pane	Chart you want to animate
Trigger options pane	**Arm when connecting to target** check box
Trigger options pane	`normal` from drop-down menu in **Mode** field

6. Build the model to generate an executable file.
7. Deploy the target application.
8. Connect Simulink to the target application.
9. Start execution of the generated model code. The chart highlights states as they execute.

View Test Point Data in Floating Scopes and Signal Viewers

When you simulate a chart in external mode, you can designate chart data of local scope to be test points and view the test point data in floating scopes and signal viewers.

To view test point data during simulation in external mode:

1. Open the Model Explorer and for each data you want to view, follow these steps:
   a. In the middle **Contents** pane, select the state or local data of interest.
   b. In the right **Dialog** pane, select the **Logging** tab and select **Test point** check box.
2. From a floating scope or signal viewer, click the signal selection button:

   The Signal Selector dialog box opens.
3. In the Signal Selector **Model hierarchy** pane, select the chart.
4. In the Signal Selector **List contents** menu, select **Testpointed/Logged signals only** and then select the signals you want to view.
5. Simulate the model in external mode as described in “Animate States During Simulation in External Mode” on page 60-89.

   The scope or viewer displays the values of the test point signals as the simulation runs.

For more information, see “Behavior of Scopes and Viewers with Rapid Accelerator Mode”.

TCP/IP and Serial External Mode Limitations

Feature	Details
Service code interface	Using external mode for tuning parameters and monitoring signals as generated code executes in a target environment is not supported for code generated from component models configured with an ERT-based system target file and service code interface.
Changing parameters	In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change • The number of states, inputs, or outputs of a block • The sample time or the number of sample times • The integration algorithm for continuous systems • The name of the model or of a block If you make these changes to the block diagram, then you must rebuild the program with newly generated code. You can change parameters in transfer function and state space representation blocks in specific ways: • The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change). • Zero entries in the State-Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (that is, the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started. • In the State-Space block, if you specify the matrices in the controllable canonical realization, then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed. If the Simulink block diagram does not match the external program, Simulink produces an error stating that the checksums do not match. The checksums take into account the top models, but not referenced models. Use the updated block diagram to rebuild the target application.
Uploading data	For parameters with fixed-point values, uploading of parameter values from the target application is not supported.
Uploading variable-size signals	Uploading of variable-size signals is not supported for these targets: • Simulink Real-Time • Texas Instruments C2000

Feature	Details
Signal value display in simulation	Graphical display of signal values in models (described in “Display Signal Values in Model Diagrams”) is not supported. For example, you cannot use the **Data Display in Simulation** menu selections **Show Value Labels When Hovering**, **Toggle Value Labels When Clicked**, and **Show Value Label of Selected Port**.
Tunable structure parameters	Uploading or downloading of tunable structure parameters is not supported.
C++ code	Generated C++ code is not supported. The simulation produces an error if you specify these configuration parameter settings: • **Language** — `C++` • **Code interface packaging** — `C++ class`
Pure integer code	Pure integer code is supported. If you do not specify `-tf` *`finalTime`* in the execution command, the target application runs the generated model code indefinitely, ignoring `StopTime`. If you specify `-tf` *`finalTime`* in the execution command: • The *`finalTime`* value represents base rate steps, not seconds. • The maximum value for *`finalTime`*, in ticks, is `MAX_int32_T`. • When the 16-bit or 32-bit tick counter overflows, the simulation time in Scope blocks returns to zero.
Half precision	Half-precision data types are not supported.
`Simulink.ImageType`	`Simulink.ImageType` data type is not supported.
Archiving data	For archiving data to disk, Scope and To Workspace blocks are supported. However, other scopes are not supported for data archiving. For example, you cannot use Floating Scope blocks or Viewers and Generators Manager viewer objects to archive data.
Scopes in referenced models	In a model hierarchy, if the top model simulates in external mode and a referenced model simulates in normal or accelerator mode, scopes in the referenced model are not displayed. However, if the top model is changed to simulate in normal mode, the behavior of scopes in the referenced models differs between normal and accelerator mode. Scopes in a referenced model simulating in normal mode are displayed, while scopes in a referenced model simulating in accelerator mode are not displayed.
Scope and Display blocks connected to Simulink message blocks	During external mode simulations, Scope and Display blocks connected to Simulink message blocks do not provide displays.
Simulation start time	Nonzero simulation start times are not supported. In the Configuration Parameters dialog box, **Solver** pane, leave **Start time** set to the default value of `0.0`.

Feature	Details
Intermediate step values	Some Simulink blocks can generate multiple values in a simulation time step. For example: • A Simulink Function or Function-Call Subsystem block that is invoked multiple times in a time step. • Iterator subsystem blocks that execute multiple times in a time step. For each time step in an external mode simulation, Simulink uploads from the target application only the final values of such blocks. Simulink does not upload the intermediate values generated during the step.
File-scoped data	File-scoped data are not supported, for example, data items to which you apply the built-in custom storage class, `FileScope`. File-scoped data are not externally accessible.
Signals with custom storage classes	Uploading of signals with custom storage classes (CSC) is not supported.
Use of `printf` Statements	To show target application error and information messages on the target hardware display, you can use `printf` calls. For some target hardware, the use of `printf` statements can increase the external mode binary file size. To disable `printf` calls, specify the preprocessor macro definition `EXTMODE_DISABLEPRINTF` for your target application compiler.
Command-line arguments	You can use command-line arguments for running target applications. These limitations apply: • Parsing of the command-line arguments requires the `sscanf` function, which increases the program size for some target hardware. • Some target applications do not accept command-line arguments. If your target hardware does not support the parsing of command-line arguments, specify the preprocessor macro definition `EXTMODE_DISABLE_ARGS_PROCESSING=1` for your target application compiler. To replace the `-w` option, you can use this command to specify that the target application enters and stays in a wait state until it receives a Connect message from Simulink: `set_param(modelName, 'OnTargetWaitForStart', 'on');` The build process provides the required option (`-DON_TARGET_WAIT_FOR_START=1`) to the compiler.
Row-major code generation	Code generated with the row-major format is not supported.
Simulation data export	The export of simulation output, signal, and state data is not supported.

See Also

More About

- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Create a Transport Layer for TCP/IP or Serial External Mode Communication” on page 60-94
- “Customize Connectivity for TCP/IP or Serial External Mode Simulations” on page 60-40

Create a Transport Layer for TCP/IP or Serial External Mode Communication

In this section...

"Design of External Mode" on page 60-94
"External Mode Communications Overview" on page 60-96
"External Mode Source Files" on page 60-97
"Implement a Custom Transport Layer" on page 60-100

This section helps you to connect your custom target by using external mode using your own low-level communications layer. The topics include:

- An overview of the design and operation of external mode
- A description of external mode source files
- Guidelines for modifying the external mode source files and building an executable to handle the tasks of the default `ext_comm` MEX-file

This section assumes that you are familiar with the execution of Simulink Coder programs, and with the basic operation of external mode.

Design of External Mode

External mode communication between the Simulink engine and a target system is based on a client/server architecture. The client (the Simulink engine) transmits messages requesting the server (target) to accept parameter changes or to upload signal data. The server responds by executing the request.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

This design makes it possible for different targets to use different transport layers. The GRT, ERT, and RSim targets support host/target communication by using TCP/IP and RS-232 (serial) communication. The Simulink Desktop Real-Time target supports shared memory communication. The Wind River Systems Tornado® target supports TCP/IP only.

The Simulink Coder product provides full source code for both the client and server-side external mode modules, as used by the GRT, ERT, Rapid Simulation, and Tornado targets, and the Simulink Desktop Real-Time and Simulink Real-Time products. The main client-side module is `ext_comm.c`. The main server-side module is `ext_svr.c`.

These two modules call the specified transport layer through the following source files.

Built-In Transport Layer Implementations

Protocol	Client or Server?	Source Files
TCP/IP	Client (host)	• *matlabroot*/toolbox/coder/simulinkcoder_core/ext_mode/host/common/rtiostream_interface.c • *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c
	Server (target)	• *matlabroot*/rtw/c/src/ext_mode/common/rtiostream_interface.c • *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c
Serial	Client (host)	• *matlabroot*/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_transport.c • *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c
	Server (target)	• *matlabroot*/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c • *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c

For serial communication, the modules `ext_serial_transport.c` and `rtiostream_serial.c` implement the client-side transport functions and the modules `ext_svr_serial_transport.c` and `rtiostream_serial.c` implement the corresponding server-side functions. For TCP/IP communication, the modules `rtiostream_interface.c` and `rtiostream_tcpip.c` implement both client-side and server-side functions. You can edit copies of these files (but do not modify the originals). You can support external mode using your own low-level communications layer by creating similar files using the following templates:

- Client (host) side: *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c (TCP/IP) or *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c (serial)
- Server (target) side: *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c (TCP/IP) or *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c (serial)

The `rtiostream_interface.c` file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing and testing an `rtiostream` communications channel, see:

- "Communications rtiostream API" on page 88-42
- `rtiostreamtest`

Implement your `rtiostream` communications channel by using the documented interface to avoid having to make changes to the file `rtiostream_interface.c` or other external mode related files.

Note Do not modify working source files. Use the templates provided in the `/custom` or `/rtiostream` folder as starting points, guided by the comments within them.

You need only provide code that implements low-level communications. You need not be concerned with issues such as data conversions between host and target, or with the formatting of messages. The Simulink Coder software handles these functions.

On the client (Simulink engine) side, communications are handled by `ext_comm` (for TCP/IP) and `ext_serial_win32_comm` (for serial) MEX-files.

On the server (target) side, external mode modules are linked into the target executable. This takes place automatically at code generation time if you select **External mode** and is based on the **Transport layer** option that you specify. The modules, called from the main program and the model execution engine, are independent of the generated model code.

The general procedure for implementing your own client-side low-level transport protocol is as follows:

1. Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls.
2. Generate a MEX-file executable for your custom transport.
3. Register your new transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box.

For more details, see “Create a Custom Client (Host) Transport Protocol” on page 60-100.

The general procedure for implementing your own server-side low-level transport protocol is as follows:

1. Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls. Typically this involves writing or integrating device drivers for your target hardware.
2. Modify template makefiles to support the new transport.

For more details, see “Create a Custom Server (Target) Transport Protocol” on page 60-103.

External Mode Communications Overview

This section gives a high-level overview of how a Simulink Coder generated program communicates with Simulink external mode. This description is based on the TCP/IP version of external mode that ships with the Simulink Coder product.

For communication to take place, both the server (target) program and the Simulink software must be executing. This does not mean that the model code in the server system must be executing. The server can be waiting for the Simulink engine to issue a command to start model execution.

The client and server communicate by using bidirectional sockets carrying packets. Packets consist either of *messages* (commands, parameter downloads, and responses) or *data* (signal uploads).

If the target application was invoked with the `-w` command-line option, the program enters a wait state until it receives a message from the host. Otherwise, the program begins execution of the model. While the target application is in a wait state, the Simulink engine can download parameters to the target and configure data uploading.

When the user chooses the **Connect to Target** option from the **Simulation** menu, the host initiates a handshake by sending an `EXT_CONNECT message`. The server responds with information about itself. This information includes

- Checksums. The host uses model checksums to determine that the target code is an exact representation of the current Simulink model.
- Data format information. The host uses this information when formatting data to be downloaded, or interpreting data that has been uploaded.

At this point, host and server are connected. The server is either executing the model or in the wait state. (In the latter case, the user can begin model execution by selecting **Start Real-Time Code** from the **Simulation** menu.)

During model execution, the message server runs as a background task. This task receives and processes messages such as parameter downloads.

Data uploading comprises both foreground execution and background servicing of the signal packets. As the target computes model outputs, it also copies signal values into data upload buffers. This occurs as part of the task associated with each task identifier (`tid`). Therefore, data collection occurs in the foreground. Transmission of the collected data, however, occurs as a background task. The background task sends the data in the collection buffers to the Simulink engine by using data packets.

The host initiates most exchanges as messages. The target usually sends a response confirming that it has received and processed the message. Examples of messages and commands are:

- Connection message / connection response
- Start target simulation / start response
- Parameter download / parameter download response
- Arm trigger for data uploading / arm trigger response
- Terminate target simulation / target shutdown response

Model execution terminates when the model reaches its final time, when the host sends a terminate command, or when a Stop Simulation block terminates execution. On termination, the server informs the host that model execution has stopped, and shuts down its socket. The host also shuts down its socket, and exits external mode.

External Mode Source Files

- “Client (Host) MEX-file Interface Source Files” on page 60-97
- “Server (Target) Source Files” on page 60-99
- “Other Files in the Server Folder” on page 60-100

Client (Host) MEX-file Interface Source Files

The source files for the MEX-file interface component are located in the folder *`matlabroot/`*`toolbox/coder/simulinkcoder_core/ext_mode/host` (open), except as noted:

- `common/ext_comm.c`

This file is the core of external mode communication. It acts as a relay station between the target and the Simulink engine. `ext_comm.c` communicates to the Simulink engine by using a shared data structure, `ExternalSim`. It communicates to the target by using calls to the transport layer.

Tasks carried out by `ext_comm.c` include establishment of a connection with the target, downloading of parameters, and termination of the connection with the target.

- `common/rtiostream_interface.c`

  This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see "Communications rtiostream API" on page 88-42. Implement your `rtiostream` communications channel using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.

- *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c`

  This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c`

  This file implements required serial transport layer functions. The version of `rtiostream_serial.c` shipped with the Simulink Coder software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.

- `serial/ext_serial_transport.c`

  This file implements required serial transport layer functions. `ext_serial_transport.c` includes `ext_serial_utils.c`, which is located in *`matlabroot`*`/rtw/c/src/ext_mode/serial` (open) and contains functions common to client and server sides.

- `common/ext_main.c`

  This file is a MEX-file wrapper for external mode. `ext_main.c` interfaces to the Simulink engine by using the standard `mexFunction` call. (See the `mexFunction` reference page and "Integrate MATLAB with External Programming Languages and Systems" for more information.) `ext_main.c` contains a function dispatcher, `esGetAction`, that sends requests from the Simulink engine to `ext_comm.c`.

- `common/ext_convert.c` and `ext_convert.h`

  This file contains functions used for converting data from host to target formats (and vice versa). Functions include byte-swapping (big to little- endian), conversion from non-IEEE floats to IEEE doubles, and other conversions. These functions are called both by `ext_comm.c` and directly by the Simulink engine (by using function pointers).

  Note You do not need to customize `ext_convert` to implement a custom transport layer. However, you might want to customize `ext_convert` for the intended target. For example, if the target represents the `float` data type in Texas Instruments format, `ext_convert` must be modified to perform a Texas Instruments to IEEE conversion.

- `common/extsim.h`

  This file defines the `ExternalSim` data structure and access macros. This structure is used for communication between the Simulink engine and `ext_comm.c`.

- `common/extutil.h`

  This file contains only conditionals for compilation of the `assert` macro.
- `common/ext_transport.h`

  This file defines functions that must be implemented by the transport layer.

Server (Target) Source Files

These files are linked into the *`model`*`.exe` executable. They are located within *`matlabroot`*`/rtw/c/src/ext_mode` (open) except as noted.

- `common/ext_svr.c`

  `ext_svr.c` is analogous to `ext_comm.c` on the host, but generally is responsible for more tasks. It acts as a relay station between the host and the generated code. Like `ext_comm.c`, `ext_svr.c` carries out tasks such as establishing and terminating connection with the host. `ext_svr.c` also contains the background task functions that either write downloaded parameters to the target model, or extract data from the target data buffers and send it back to the host.
- `common/rtiostream_interface.c`

  This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see “Communications rtiostream API” on page 88-42. Implement your `rtiostream` communications channel by using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.
- *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c`

  This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.
- *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c`

  This file implements required serial transport layer functions. The version of rtiostream_serial.c shipped with the software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.
- *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostream.h`

  This file defines the `rtIOStream*` functions implemented in `rtiostream_tcpip.c`.
- `serial/ext_svr_serial_transport.c`

  This file implements required serial transport layer functions. `ext_svr_serial_transport.c` includes `serial/ext_serial_utils.c`, which contains functions common to client and server sides.
- `common/updown.c`

  `updown.c` handles the details of interacting with the target model. During parameter downloads, `updown.c` does the work of installing the new parameters into the model's parameter vector. For data uploading, `updown.c` contains the functions that extract data from the model's `blockio` vector and write the data to the upload buffers. `updown.c` provides services both to `ext_svr.c` and to the model code (for example, `grt_main.c`). It contains code that is called by using the

background tasks of `ext_svr.c` as well as code that is called as part of the higher priority model execution.

- *`matlabroot`*`/rtw/c/src/dt_info.h` (included by generated model build file *`model`*`.h`)

  These files contain data type transition information that allows access to multi-data type structures across different computer architectures. This information is used in data conversions between host and target formats.

- `common/updown_util.h`

  This file contains only conditionals for compilation of the `assert` macro.

- `common/ext_svr_transport.h`

  This file defines the `Ext*` functions that must be implemented by the server (target) transport layer.

Other Files in the Server Folder

- `common/ext_share.h`

  Contains message code definitions and other definitions required by both the host and target modules.

- `serial/ext_serial_utils.c`

  Contains functions and data structures for communication, MEX link, and generated code required by both the host and target modules of the transport layer for serial protocols.

- The serial transport implementation includes the additional files
  - `serial/ext_serial_pkt.c` and `ext_serial_pkt.h`
  - `serial/ext_serial_port.h`

Implement a Custom Transport Layer

- “Requirements for Custom Transport Layers” on page 60-100
- “Create a Custom Client (Host) Transport Protocol” on page 60-100
- “MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files” on page 60-101
- “Register a Custom Client (Host) Transport Protocol” on page 60-102
- “Create a Custom Server (Target) Transport Protocol” on page 60-103
- “Serial Receive Buffer Smaller than 64 Bytes” on page 60-104

Requirements for Custom Transport Layers

- By default, `ext_svr.c` and `updown.c` use `malloc` to allocate buffers in target memory for messages, data collection, and other purposes, although there is also an option to preallocate static memory. If your target uses another memory allocation scheme, you must modify these modules.
- The target is assumed to support both `int32_T` and `uint32_T` data types.

Create a Custom Client (Host) Transport Protocol

To implement the client (host) side of your low-level transport protocol,

1. Edit the template file *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c to replace low-level communication calls with your own communication calls.

   a. Copy and rename the file to rtiostream_*name*.c (replacing *name* with a name meaningful to you).

   b. Replace the functions rtIOStreamOpen, rtIOStreamClose, rtIOStreamSend, and rtIOStreamRecv with functions (of the same name) that call your low-level communication primitives. These functions are called from other external mode modules via rtiostream_interface.c. For more information, see “Communications rtiostream API” on page 88-42.

   c. Build your rtiostream implementation into a shared library that exports the rtIOStreamOpen, rtIOStreamClose, rtIOStreamRecv and rtIOStreamSend functions.

2. Build the customized MEX-file executable using the MATLAB mex function. See “MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files” on page 60-101 for examples of mex invocations.

   Do not replace the existing ext_comm MEX-file if you want to preserve its functionality. Instead, use the -output option to name the new executable file, for example, my_ext_comm. For more information, see mex.

3. Register your new client transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box. For details, see “Register a Custom Client (Host) Transport Protocol” on page 60-102.

Sample commands for rebuilding external mode MEX-files are listed in “MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files” on page 60-101.

MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files

The following table lists the commands for building the standard ext_comm and ext_serial_win32 modules on PC and UNIX platforms.

Platform	Commands

Windows, TCP/IP	`cd (matlabroot)`
	`mex toolbox/coder/simulinkcoder_core/ext_mode/host/common/ext_comm.c ...`
	`toolbox/coder/simulinkcoder_core/ext_mode/host/common/ext_convert.c ...`
	`toolbox/coder/simulinkcoder_core/ext_mode/host/common/rtiostream_interface.c ...`
	`toolbox/coder/simulinkcoder_core/ext_mode/host/common/ext_util.c ...`
	`-R2018a ...`
	`-Itoolbox/coder/rtiostream/src ...`
	`-Itoolbox/coder/rtiostream/src/utils_host ...`
	`-Itoolbox/coder/simulinkcoder_core/ext_mode/host/common/include ...`
	`-Irtw/c/src/ext_mode/common ...`
	`-lmwrtiostreamutils -lmwsl_services ...`
	`-DEXTMODE_TCPIP_TRANSPORT ...`
	`-DSL_EXT_DLL -output my_ext_comm`
	
	Note The rtiostream_interface.c function defines RTIOSTREAM_SHARED_LIB as libmwrtiostreamtcpip and dynamically loads the MathWorks TCP/IP rtiostream shared library. Modify this file if you need to load a different rtiostream shared library.

Platform	Commands
Linux, TCP/IP	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_SO`. • Replace back slashes with forward slashes.
Mac, TCP/IP	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_DYLIB`. • Replace back slashes with forward slashes.

Windows, serial	`cd (matlabroot)`
	`mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_transport.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_pkt.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\rtiostream_serial_interface.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ...`
	`-R2018a ...`
	`-Itoolbox\coder\rtiostream\src ...`
	`-Itoolbox\coder\rtiostream\src\utils_host ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ...`
	`-Irtw\c\src\ext_mode\common ...`
	`-Irtw\c\src\ext_mode\serial ...`
	`-lmwrtiostreamutils -lmwsl_services ...`
	`-DEXTMODE_SERIAL_TRANSPORT -DSL_EXT_DLL ...`
	`-output my_ext_serial_comm`
	
	Note The `rtiostream_interface.c` function defines RTIOSTREAM_SHARED_LIB as `libmwrtiostreamserial` and dynamically loads the MathWorks serial `rtiostream` shared library. Modify this file if you need to load a different `rtiostream` shared library.

Linux, serial	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_SO`. • Replace back slashes with forward slashes.
Mac, serial	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_DYLIB`. • Replace back slashes with forward slashes.

Note `mex` requires a compiler supported by the MATLAB API. See the `mex` reference page and "Integrate MATLAB with External Programming Languages and Systems" for more information about the `mex` function.

Register a Custom Client (Host) Transport Protocol

To register a custom client transport protocol with the Simulink software, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
% -- end of sl_customization
```

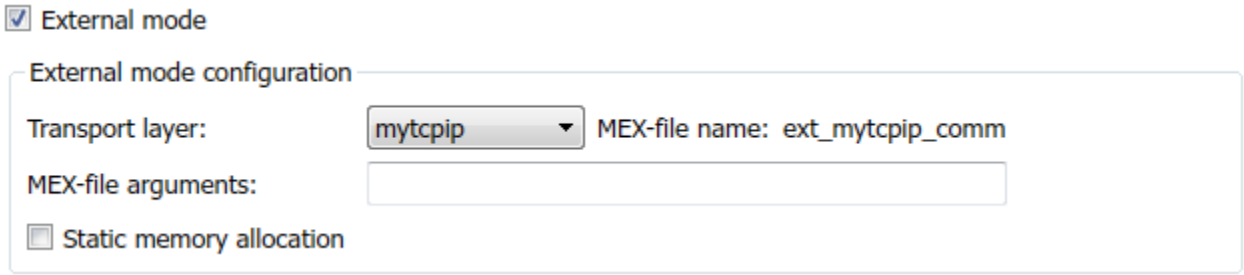
where

- *stf*`.tlc` is the name of the system target file for which the transport will be registered (for example, `'grt.tlc'`)
- *transport* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'mytcpip'`)
- *mexfile* is the name of the transport's associated external interface MEX-file (for example, `'ext_mytcpip_comm'`)

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('grt.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
  cm.ExtModeTransports.add('ert.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
% -- end of sl_customization
```

If you place the `sl_customization.m` file containing the transport registration information on the MATLAB path, your custom client transport protocol will be registered with each subsequent Simulink session. The name of the transport will appear in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box. When you select the transport for your model, the name of the associated external interface MEX-file will appear in the noneditable **MEX-file name** field, as shown in the following figure.



Create a Custom Server (Target) Transport Protocol

The `rtIOStream*` function prototypes in *matlabroot*`/toolbox/coder/rtiostream/src/rtiostream.h` define the calling interface for both the server (target) and client (host) side transport layer functions.

- The TCP/IP implementations are in *matlabroot*`/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c`.
- The serial implementations are in *matlabroot*`/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c`.

Note The `Ext*` function prototypes in *matlabroot*`/rtw/c/src/ext_mode/common/ext_svr_transport.h` are implemented in *matlabroot*`/rtw/c/src/ext_mode/common/rtiostream_interface.c` or *matlabroot*`/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c`. In most cases you will not need to modify `rtiostream_interface.c` or `rtiostream_serial_interface.c` for your custom TCP/IP or serial transport layer.

To implement the server (target) side of your low-level TCP/IP or serial transport protocol:

1. Edit the template *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c or *matlabroot*/toolbox/coder/rtiostream/src/rtiostreamserial/rtiostream_serial.c to replace low-level communication calls with your own communication calls.

   a. Copy and rename the file to rtiostream_*name*.c (replacing *name* with a name meaningful to you).

   b. Replace the functions rtIOStreamOpen, rtIOStreamClose, rtIOStreamSend, and rtIOStreamRecv with functions (of the same name) that call your low-level communication drivers.

      You must implement the functions defined in rtiostream.h, and your implementations must conform to the prototypes defined in that file. Refer to the original rtiostream_tcpip.c or rtiostream_serial.c for guidance.

2. Incorporate the external mode source files for your transport layer into the model build process. Use a build process mechanism such as a post code generation command or a before_make hook function to make the transport files available to the build process. For more information on the build process mechanisms, see "Customize Post-Code-Generation Build Processing", "Customize Build Process with STF_make_rtw_hook File", and "Customize Build Process with sl_customization.m".

   For example:

   - Add the file created in the previous step to the build information:

     ```
     path/rtiostream_name.c
     ```

   - For TCP/IP, add the following file to the build information:

     ```
     matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c
     ```

   - For serial, add the following files to the build information:

     ```
     matlabroot/rtw/c/src/ext_mode/serial/ext_serial_pkt.c
     matlabroot/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c
     matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c
     ```

Note For external mode, check that rtIOStreamRecv is not a blocking implementation. Otherwise, it might cause the external mode server to block until the host sends data through the comm layer.

Serial Receive Buffer Smaller than 64 Bytes

For serial communication, if the serial receive buffer of your target is smaller than 64 bytes:

1. Update the following macro with the actual target buffer size:

   ```
   #define TARGET_SERIAL_RECEIVE_BUFFER_SIZE 64
   ```

   Implement the change in the following files:

   ```
   matlabroot/rtw/c/src/ext_mode/serial/ext_serial_utils.c
   matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_utils.c
   ```

2. Run the command to rebuild the ext_serial_win32 MEX-file. See "MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files" on page 60-101.

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”
- “External Mode Simulation with TCP/IP or Serial Communication” on page 60-55
- “Customize Connectivity for TCP/IP or Serial External Mode Simulations” on page 60-40
- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2
- “Customize XCP Server Software” on page 60-45

Logging in Simulink Coder

Log Program Execution Results

Multiple techniques are available by which a program generated by the Simulink Coder software can save data to a MAT-file for analysis. A generated executable can save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *model*.mat, where *model* is the name of your model. See “Log Data for Analysis” on page 61-2 for a data logging tutorial.

Note Data logging is available only for system target files that have access to a file system. In addition, only the RSim target executables are capable of accessing MATLAB workspace data.

For MAT-file logging limitations, see the configuration parameter MAT-file logging.

Log Data for Analysis

- “Set Up and Configure Model” on page 61-2
- “Data Logging During Simulation” on page 61-3
- “Data Logging from Generated Code” on page 61-5

Set Up and Configure Model

This example shows how data generated by a copy of the model `slexAircraftExample` is logged to the file `myAircraftExample.mat`. Refer to “Build Process Workflow for Real-Time Systems” on page 59-39 for instructions on setting up a copy of `slexAircraftExample` as `myAircraftExample` in a working folder if you have not done so already.

Note When you configure the code generator to produce code that includes support for data logging during execution, the code generator can include text for block names in the block paths included in the log file. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see “Internationalization and Code Generation”.

To configure data logging, open the Configuration Parameters dialog box and select the **Data Import/Export** pane. The process is the same as configuring a Simulink model to save output to the MATLAB workspace. For each workspace return variable you define and enable, the Simulink Coder software defines a parallel MAT-file variable. For example, if you save simulation time to the variable `tout`, your generated program logs the same data to a variable named `rt_tout`. You can change the prefix `rt_` to a suffix (`_rt`), or eliminate it entirely. You do this by setting model configuration parameter **MAT-file variable name modifier**.

To log signals in generated code, use the **Data Import/Export** model configuration parameters described below or include To File or To Workspace blocks in your model.

Note MAT-file logging for code generation does not support signal logging. If you enable MAT-file and signal logging by using **Data Import/Export** model configuration parameters and select signals for logging in the Simulink Editor, when you build the model, the code generator displays this warning:

```
Warning: MAT-file logging does not support signal logging.
When your model code executes, the signal logging variable 'rt_logsout' will
not be saved to the MAT-file.
```

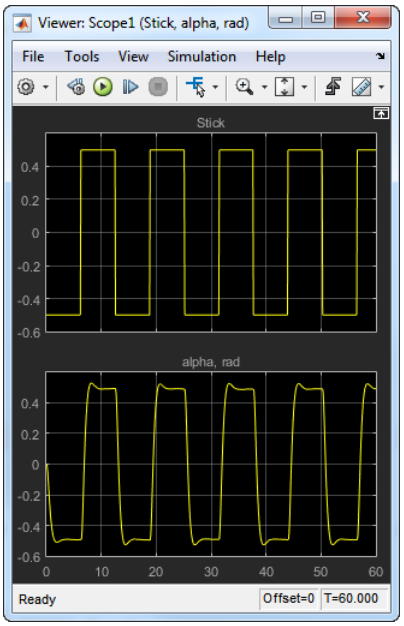
To avoid this warning, clear the model configuration parameter **Signal logging**.

In this example, you modify the `myAircraftExample` model so that the generated program saves the simulation time and system outputs to the file `myAircraftExample.mat`. Then you load the data into the base workspace and plot simulation time against one of the outputs. The `myAircraftExample` model should be configured as described in "Build Process Workflow for Real-Time Systems" on page 59-39.

Data Logging During Simulation

To use the data logging feature:

1. Open the `myAircraftExample` model if it is not already open.
2. Open the Configuration Parameters dialog box.
3. Select the **Data Import/Export** pane. The **Data Import/Export** pane lets you specify which outport data is to be saved to the workspace and what variable names to use for it.
4. Set parameter **Format** to `Structure with time`. When you select this format, Simulink saves the model states and outputs in structures that have their names specified in the **Save to workspace or file** area. By default, the structures are `xout` for states and `yout` for output. The structure used to save output has two top-level fields: `time` and `signals`. The `time` field contains a vector of simulation times and `signals` contains an array of substructures, each of which corresponds to a model output port.
5. Select **Output**. This tells Simulink to save output signal data during simulation as a variable named `yout`. Selecting **Output** enables the code generator to create code that logs the root Output block (`alpha, rad`) to a MAT-file.
6. Set **Decimation** to 1.
7. If other parameters are selected, clear them.
8. Click **Apply** and **OK** to register your changes and close the dialog box.
9. Save the model.
10. In the model window, double-click the scope symbol next to the Aircraft Dynamics Model block, then simulate the model. The resulting scope display is shown below.



11 Verify that the simulation time and outputs have been saved to the base workspace in MAT-files. At the MATLAB prompt, type:

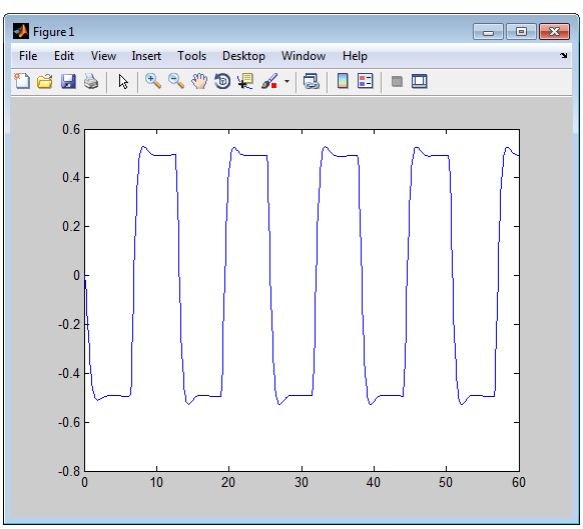
```
whos yout
```

Simulink displays:

```
  Name        Size            Bytes  Class     Attributes

  yout        1x1             10756  struct
```

12 Verify that `alpha, rad` was logged by plotting simulation time versus that variable. In the Command Window, type:

```
plot(yout.time,yout.signals.values)
```



Data Logging from Generated Code

In the second part of this example, you build and run an executable program built by the code generator for model `myAircraftExample`. The program outputs a MAT-file containing the simulation time and output you previously examined. Even though you have already generated code for the `myAircraftExample` model, you must now regenerate that code because you have changed the model by enabling data logging. The steps below explain this procedure.

To avoid overwriting workspace data with data from simulation runs, the code generator modifies identifiers for variables logged by Simulink. You can control these modifications.

1. Set model configuration parameter **MAT-file variable name modifier** to `_rt`. This adds the suffix `_rt` to each variable that you selected to be logged in the first part of this example.
2. Click **Apply** and **OK** to register your changes and close the dialog box.
3. Save the model.
4. Build an executable.
5. When the build concludes, run the executable program with the command:

   ```
   !myAircraftExample
   ```

6. The program now produces two message lines, indicating that the MAT-file has been written.

   ```
   ** starting the model **
   ** created myAircraftExample.mat **
   ```

7. Load the MAT-file data created by the executable and look at the workspace variables from simulation and the generated program by typing:

   ```
   load myAircraftExample.mat
   whos yout*
   ```

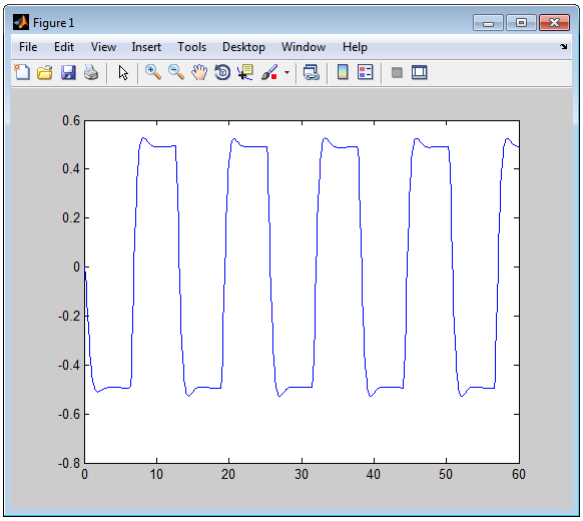
   Simulink displays:

   ```
     Name           Size            Bytes  Class     Attributes

     yout           1x1             10756  struct
     yout_rt        1x1             10756  struct
   ```

   Note the size and bytes of the structures resulting from the simulation run and generated code are the same.

8. Plot the generated code output by entering the following command in the Command Window:

   ```
   plot(yout_rt.time,yout_rt.signals.values)
   ```



The plot should be identical to the plot that you produced in the previous part of this example.

Tip For UNIX platforms, run the executable program in the Command Window with the syntax `!./executable_name`. If preferred, run the executable program from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs".

Configure State, Time, and Output Logging

The **Data Import/Export** pane enables a generated program to save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) `model.mat`.

Before using this data logging feature, you should learn how to configure a Simulink model to return output to the MATLAB workspace. This is discussed in "Save Simulation Data".

For each workspace return variable that you define and enable, the code generator defines a MAT-file variable. For example, if your model saves simulation time to the workspace variable `tout`, your generated program logs the same data to a variable named (by default) `rt_tout`.

The code generated by the code generator logs the following data:

- Root Outport blocks

  The default MAT-file variable name for system outputs is `rt_yout`.

  The sort order of the `rt_yout` array is based on the port number of the Outport block, starting with 1.
- Continuous and discrete states in the model

  The default MAT-file variable name for system states is `rt_xout`.
- Simulation time

  The default MAT-file variable name for simulation time is `rt_tout`.

- “Override Default MAT-File Variable Names” on page 61-7
- “Override Default MAT-File Name or Buffer Size” on page 61-7

Override Default MAT-File Variable Names

By default, the code generator prefixes the text `rt_` to the variable names for system outputs, states, and simulation time to form MAT-file variable names. To change this prefix for a model, set model configuration parameter **MAT-file variable name modifier** to (`rt_`), a suffix (`_rt`), or no modifier (`none`). Other system target files might not support this parameter.

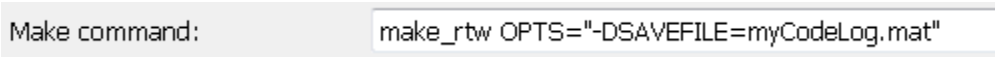
Override Default MAT-File Name or Buffer Size

You can specify compiler options to override the following MAT-file attributes in generated code:

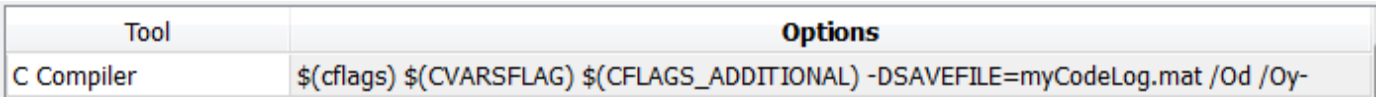
MAT-File Attribute	Default	Compiler Option
Name	*`model`*`.mat`	`-DSAVEFILE=`*`filename`*
Size of data logging buffer	1024 bytes	`-DDEFAULT_BUFFER_SIZE=`*`n`*

Note Valid option syntax can vary among compilers. For example, Microsoft Visual C++ compilers typically accept `/DSAVEFILE=`*`filename`* as well as `-DSAVEFILE=`*`filename`*.

For a template makefile (TMF) based target, set model configuration parameter **Make command** to the compiler option. For example:



For a toolchain-based system target file such as GRT or ERT, add the compiler option to model configuration. Set parameter **Build configuration** to `Specify`, and add the compiler option to the **C Compiler** row of the **Tool/Options** table. For example:



To add the compiler option to a custom toolchain, you can modify and reregister the custom toolchain using the procedures shown in the example “Add Custom Toolchains to MATLAB® Coder™ Build Process”. For example, to add the compiler option to the MATLAB source file for the custom toolchain, you could define `myCompilerOpts` as follows:

```
optimsOffOpts    = {'/c /Od'};
optimsOnOpts     = {'/c /O2'};
cCompilerOpts    = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts  = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
myCompilerOpts   = {' -DSAVEFILE=myCodeLog.mat '};
...
```

Then you can add `myCompilerOpts` to the flags for each configuration and compiler to which it applies, for example:

```
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, myCompilerOpts, optimsOffOpts));
```

As shown in “Add Custom Toolchains to MATLAB® Coder™ Build Process”, after modifying the custom toolchain, you save the configuration to a MAT-file and refresh the target registry.

Log Data with Scope and To Workspace Blocks

The code generated by the code generator also logs data from these sources:

- Scope blocks that have the block parameter **Log data to workspace** enabled

  You must specify the variable name and data format in each Scope block's dialog box.
- To Workspace blocks in the model

  You must specify the variable name and data format in each To Workspace block's dialog box.

The variables are written to `model.mat`, along with variables logged from the **Workspace I/O** pane.

Log Data with To File Blocks

You can also log data to a To File block. The generated program creates a separate MAT-file (distinct from `model.mat`) for each To File block in the model. The file contains the block time and input data. You must specify the file name, variable name, decimation, and sample time in the To File block dialog box.

Note Models referenced by Model blocks do not perform data logging in that context except for states, which you can include in the state logged for top models. Code generated by the Simulink Coder software for referenced models does not perform data logging to MAT-files.

Data Logging Differences Between Single- and Multitasking

When logging data in single-tasking and multitasking systems, you will notice differences in the logging of

- Noncontinuous root Outport blocks
- Discrete states

In multitasking mode, the logging of states and outputs is done after the first task execution (and not at the end of the first time step). In single-tasking mode, the code generated by the build procedure logs states and outputs after the first time step.

See Data Logging in Single-Tasking and Multitasking Model Execution for more details on the differences between single-tasking and multitasking data logging.

Note The rapid simulation target (RSim) provides enhanced logging options. See “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” for more information.

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment”

- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Data Interchange Using the C API in Simulink Coder

- “Exchange Data Between Generated and External Code Using C API” on page 62-2
- “Use C API to Access Model Signals and States” on page 62-19
- “Use C API to Access Model Parameters” on page 62-24

Exchange Data Between Generated and External Code Using C API

Some Simulink Coder applications must interact with signals, states, root-level inputs/outputs, or parameters in the generated code for a model. For example, calibration applications monitor and modify parameters. Signal monitoring or data logging applications interface with signal, state, and root-level input/output data. Using the Simulink Coder C API, you can build target applications that log signals, states, and root-level inputs/outputs, monitor signals, states, and root-level inputs/outputs, and tune parameters, while the generated code executes.

The C API minimizes its memory footprint by sharing information common to signals, states, root-level inputs/outputs, and parameters in smaller structures. Signal, state, root-level input/output, and parameter structures include an index into the structure map, allowing multiple signals, states, root-level inputs/outputs, or parameters to share data.

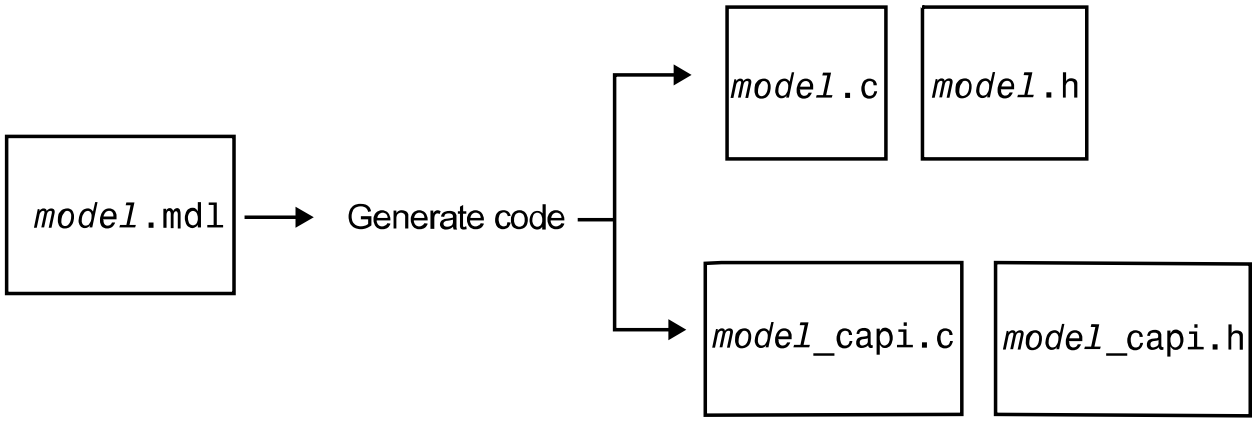
To get started with an example, see “Use C API to Access Model Signals and States” on page 62-19 or “Use C API to Access Model Parameters” on page 62-24.

In this section...

“Generated C API Files” on page 62-2
“Generate C API Files” on page 62-4
“Description of C API Files” on page 62-5
“Generate C API Data Definition File for Exchanging Data with a Target System” on page 62-16
“C API Limitations” on page 62-17

Generated C API Files

When you configure a model to use the C API, the Simulink Coder code generator generates two additional files, *model*`_capi.c` (or `.cpp`) and *model*`_capi.h`, where *model* is the name of the model. The code generator places the two C API files in the build folder, based on settings in the Configuration Parameters dialog box. The C API source code file contains information about global block output signals, states, root-level inputs/outputs, and global parameters defined in the generated code model source code. The C API header file is an interface header file between the model source code and the generated C API. You can use the information in these C API files to create your application. Among the files generated are those shown in the next figure.



Generated Files with C API Selected

Note When you configure the code generator to produce code that includes support for the C API interface and data logging, the code generator can include text for block names in the block paths logged to C API files *model*`_capi.c` (or `.cpp`) and *model*`_capi.h`. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see "Internationalization and Code Generation".

Generate C API Files

To generate C API files for your model:

1. Select the C API interface for your model. There are two ways to select the C API interface for your model, as described in the following sections.
   - "Select C API with Configuration Parameters Dialog" on page 62-4
   - "Select C API from the Command Line" on page 62-4
2. Generate code for your model.

After generating code, you can examine the files *model*`_capi.c` (or `.cpp`) and *model*`_capi.h` in the model build folder.

Select C API with Configuration Parameters Dialog

1. Open your model, and open the Configuration Parameters dialog box.
2. In the **Code Generation** > **Interface** pane, in the **Data exchange interface** subgroup, select one or more `C API` options. Based on the options you select, support for accessing signals, parameters, states, and root-level I/O will appear in the C API generated code.
   - If you want to generate C API code for global block output signals, select **Generate C API for: signals**.
   - If you want to generate C API code for global block parameters, select **Generate C API for: parameters**.
   - If you want to generate C API code for discrete and continuous states, select **Generate C API for: states**.
   - If you want to generate C API code for root-level inputs and outputs, select **Generate C API for: root-level I/O**.

Select C API from the Command Line

From the MATLAB command line, you can use the `set_param` function to select or clear C API model configuration parameters. At the MATLAB command line, enter one or more of the following commands, where `modelname` is the name of your model.

To select **Generate C API for: signals**, enter:

```
set_param('modelname','RTWCAPISignals','on')
```

To clear **Generate C API for: signals**, enter:

```
set_param('modelname','RTWCAPISignals','off')
```

To select **Generate C API for: parameters**, enter:

```
set_param('modelname','RTWCAPIParams','on')
```

To clear **Generate C API for: parameters**, enter:

```
set_param('modelname','RTWCAPIParams','off')
```

To select **Generate C API for: states**, enter:

```
set_param('modelname','RTWCAPIStates','on')
```

To clear **Generate C API for: states**, enter:

```
set_param('modelname','RTWCAPIStates','off')
```

To select **Generate C API for: root-level I/O**, enter:

```
set_param('modelname','RTWCAPIRootIO','on')
```

To clear **Generate C API for: root-level I/O**, enter:

```
set_param('modelname','RTWCAPIRootIO','off')
```

Description of C API Files

- “About C API Files” on page 62-5
- “Structure Arrays Generated in C API Files” on page 62-7
- “Generate Example C API Files” on page 62-9
- “C API Signals” on page 62-9
- “C API States” on page 62-12
- “C API Root-Level Inputs and Outputs” on page 62-12
- “C API Parameters” on page 62-14
- “Map C API Data Structures to rtModel” on page 62-15

About C API Files

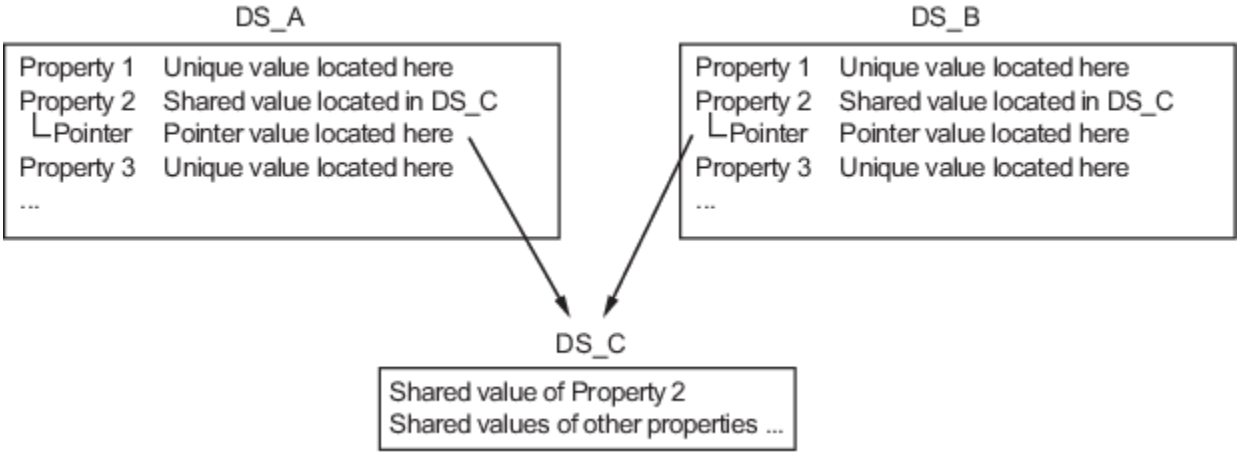
The *model*`_capi.c` (or `.cpp`) file provides external applications with a consistent interface to model data. Depending on your configuration settings, the data could be a signal, state, root-level input or output, or parameter. In this document, the term data item refers to either a signal, a state, a root-level input or output, or a parameter. The C API uses structures that provide an interface to the data item properties. The interface packages the properties of each data item in a data structure. If the model contains multiple data items, the interface generates an array of data structures. The members of a data structure map to data properties.

To interface with data items, an application requires the following properties for each data item:

- Name
- Block path
- Port number (for signals and root-level inputs/outputs only)
- Address
- Data type information: native data type, data size, complexity, and other attributes
- Dimensions information: number of rows, number of columns, and data orientation (scalar, vector, matrix, or *n*-dimensional)

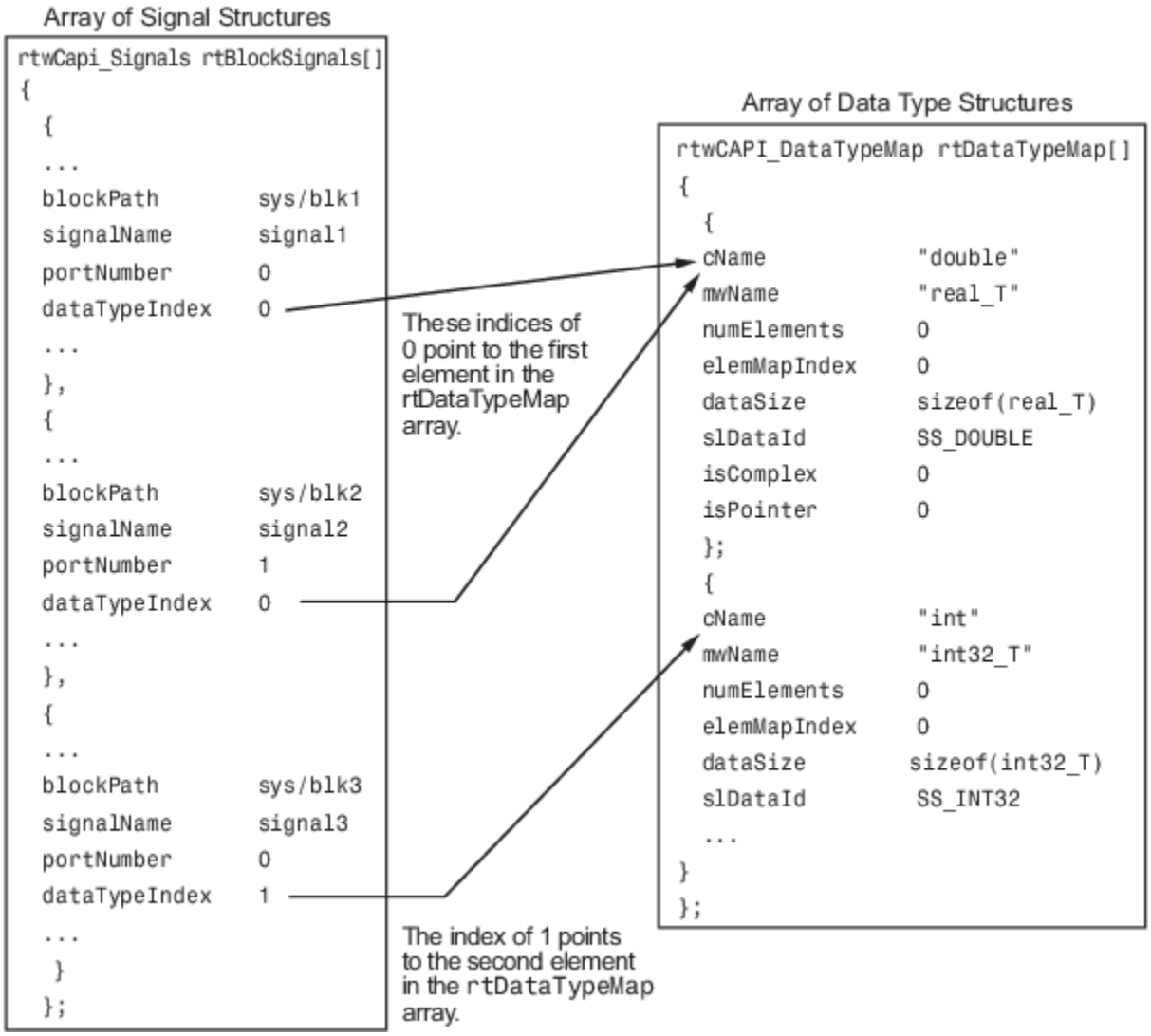
- Fixed-point information: slope, bias, scale type, word length, exponent, and other attributes
- Sample-time information (for signals, states, and root-level inputs/outputs only): sample time, task identifier, frames

As illustrated in the next figure, the properties of data item A, for example, are located in data structure DS_A. The properties of data item B are located in data structure DS_B.



Some property *values* can be unique to each data item, and there are some property values that several data items can share in common. Name, for example, has a unique value for each data item. The interface places the unique property values directly in the structure for the data item. The name value of data item A is in DS_A, and the name value of data item B is in DS_B.

But data type could be a property whose value several data items have in common. The ability of some data items to share a property allows the C API to have a reuse feature. In this case, the interface places only an index value in DS_A and an index value in DS_B. These indices point to a different data structure, DS_C, that contains the actual data type value. The next figure shows this scheme with more detail.



The figure shows three signals. `signal1` and `signal2` share the same data type, `double`. Instead of specifying this data type value in each signal data structure, the interface provides only an index value, 0, in the structure. `"double"` is described by entry 0 in the `rtDataTypeMap` array, which is referenced by both signals. Additionally, property values can be shared between signals, states, root-level inputs/outputs, and parameters, so states, root-level inputs/outputs, and parameters also might reference the `double` entry in the `rtDataTypeMap` array. This reuse of information reduces the memory size of the generated interface.

Structure Arrays Generated in C API Files

As with data type, the interface maps other common properties (such as address, dimension, fixed-point scaling, and sample time) into separate structures and provides an index in the structure for the data item. For a complete list of structure definitions, refer to the file *`matlabroot`*`/rtw/c/src/rtw_capi.h`. This file also describes each member in a structure. The structure arrays generated in the *`model`*`_capi.c` (or `.cpp`) file are of structure types defined in the `rtw_capi.h` file. Here is a brief description of the structure arrays generated in *`model`*`_capi.c` (or `.cpp`):

- **rtBlockSignals** is an array of structures that contains information about global block output signals in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the signal name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **rtBlockParameters** is an array of structures that contains information about the tunable block parameters in the model by block name and parameter name. Each element in the array is of type `struct rtwCAPI_BlockParameters`. The members of this structure provide the parameter name, block path, address, and indices to data type, dimension, and fixed-point structure arrays.
- **rtBlockStates** is an array of structures that contains information about discrete and continuous states in the model. Each element in the array is of type `struct rtwCAPI_States`. The members of this structure provide the state name, block path, type (continuous or discrete), and indices to the address, data type, dimension, fixed-point, and sample-time structure arrays.
- **rtRootInputs** is an array of structures that contains information about root-level inputs in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the root-level input name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **rtRootOutputs** is an array of structures that contains information about root-level outputs in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the root-level output name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **rtModelParameters** is an array of structures that contains information about workplace variables that one or more blocks or Stateflow charts in the model reference as block parameters. Each element in the array is of data type `rtwCAPI_ModelParameters`. The members of this structure provide the variable name, address, and indices to data type, dimension, and fixed-point structure arrays.
- **rtDataAddrMap** is an array of base addresses of signals, states, root-level inputs/outputs, and parameters that appear in the `rtBlockSignals`, `rtBlockParameters`, `rtBlockStates`, and `rtModelParameters` arrays. Each element of the `rtDataAddrMap` array is a pointer to `void (void*)`.
- **rtDataTypeMap** is an array of structures that contains information about the various data types in the model. Each element of this array is of type `struct rtwCAPI_DataTypeMap`. The members of this structure provide the data type name, size of the data type, and information on whether or not the data is complex.
- **rtDimensionMap** is an array of structures that contains information about the various data dimensions in the model. Each element of this array is of type `struct rtwCAPI_DimensionMap`. The members of this structure provide information on the number of dimensions in the data, the orientation of the data (whether it is scalar, vector, or a matrix), and the actual dimensions of the data.
- **rtFixPtMap** is an array of structures that contains fixed-point information about the signals, states, root-level inputs/outputs, and parameters. Each element of this array is of type `struct rtwCAPI_FixPtMap`. The members of this structure provide information about the data scaling, bias, exponent, and whether or not the fixed-point data is signed. If the model does not have fixed-point data (signal, state, root-level input/output, or parameter), the Simulink Coder software assigns `NULL` or zero values to the elements of the `rtFixPtMap` array.
- **rtSampleTimeMap** is an array of structures that contains sampling information about the global signals, states, and root-level inputs/outputs in the model. (This array does not contain information about parameters.) Each element of this array is of type `struct rtwCAPI_SampleTimeMap`. The members of this structure provide information about the sample period, offset, and whether or not the data is frame-based or sample-based.

Generate Example C API Files

Subtopics “C API Signals” on page 62-9, “C API States” on page 62-12, “C API Root-Level Inputs and Outputs” on page 62-12, and “C API Parameters” on page 62-14 discuss generated C API structures using the example model `rtwdemo_capi`. To generate code from the example model, do the following:

1. Open the model by clicking the `rtwdemo_capi` link above or by typing `rtwdemo_capi` on the MATLAB command line.
2. If you want to generate C API structures for root-level inputs/outputs in `rtwdemo_capi`, select model configuration parameter **Generate C API for: root-level I/O**.

   The setting this parameter must match between the top model and the referenced model. If you modify the parameter setting, save the top model and the referenced model to the same writable work folder.
3. Generate code for the model.

The C API code examples in the next subtopics are generated with C as the target language.

This model has three global block output signals that will appear in C API generated code:

- `top_sig1`, which is a test point at the output of the Gain1 block in the top model
- `sig2_eg`, which appears in the top model and is defined in the base workspace as a `Simulink.Signal` object having storage class `ExportedGlobal`
- `bot_sig1`, which appears in the referenced model `rtwdemo_capi_bot` and is defined as a `Simulink.Signal` object having storage class `Model default`

The model also has two discrete states that will appear in the C API generated code:

- `top_state`, which is defined for the Delay1 block in the top model
- `bot_state`, which is defined for the Discrete Filter block in the referenced model

The model has root-level inputs/outputs that will appear in the C API generated code if you select model configuration parameter **Generate C API for: root-level I/O**:

- Four root-level inputs, `In1` through `In4`
- Six root-level outputs, `Out1` through `Out6`

Additionally, the model has five global block parameters that will appear in C API generated code:

- `Kp` (top model `Gain1` block and referenced model `Gain2` block share)
- `Ki` (referenced model `Gain3` block)
- `p1` (lookup table `lu1d`)
- `p2` (lookup table `lu2d`)
- `p3` (lookup table `lu3d`)

C API Signals

The `rtwCAPI_Signals` structure captures signal information including the signal name, address, block path, output port number, data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API signals for the top model in `rtwdemo_capi`:

```
/* Block output signal information */
static const rtwCAPI_Signals rtBlockSignals[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 0, 0, "rtwdemo_capi/Gain1",
    "top_sig1", 0, 0, 0, 0, 0 },

  { 1, 0, "rtwdemo_capi/lu2d",
    "sig2_eg", 0, 0, 1, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

Note To better understand the code, read the comments in the file. For example, notice the comment that begins on the third line in the preceding code. This comment lists the members of the `rtwCAPI_Signals` structure, in order. This tells you the order in which the assigned values for each member appear for a signal. In this example, the comment tells you that `signalName` is the fourth member of the structure. The following lines describe the first signal:

```
  { 0, 0, "rtwdemo_capi/Gain1",
    "top_sig1", 0, 0, 0, 0, 0 },
```

From these lines you infer that the name of the first signal is `top_sig1`.

Each array element, except the last, describes one output port for a block signal. The final array element is a sentinel, with all elements set to null values. For example, examine the second signal, described by the following code:

```
  { 1, 0, "rtwdemo_capi/lu2d",
    "sig2_eg", 0, 0, 1, 0, 0 },
```

This signal, named `sig2_eg`, is the output signal of the first port of the block `rtwdemo_capi/lu2d`. (This port is the first port because the zero-based index for `portNumber` displayed on the second line is assigned the value `0`.)

The address of this signal is given by `addrMapIndex`, which, in this example, is displayed on the first line as `1`. This provides an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`:

```
/* Declare Data Addresses statically */
static void* rtDataAddrMap[] = {
  &rtwdemo_capi_B.top_sig1,            /* 0: Signal */
  &sig2_eg[0],                         /* 1: Signal */
  &rtwdemo_capi_DWork.top_state,       /* 2: Discrete State */
  &rtP_Ki,                             /* 3: Model Parameter */
  &rtP_Kp,                             /* 4: Model Parameter */
  &rtP_p1[0],                          /* 5: Model Parameter */
  &rtP_p2[0],                          /* 6: Model Parameter */
  &rtP_p3[0],                          /* 7: Model Parameter */
};
```

The index of `1` points to the second element in the `rtDataAddrMap` array. From the `rtDataAddrMap` array, you can infer that the address of this signal is `&sig2_eg[0]`.

This level of indirection supports multiple code instances of the same model. For multiple instances, the signal information remains constant, except for the address. In this case, the model is a single instance. Therefore, the `rtDataAddrMap` is declared statically. If you choose to generate reusable code, an initialize function is generated that initializes the addresses dynamically per instance. For details on generating reusable code, see "Configure Generated C Function Interface for Model Entry-Point Functions" and see "Configure Code Reuse Support" on page 50-10.

The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the signal:

```
/* Data Type Map - use dataTypeMapIndex to access this structure */
static const rtwCAPI_DataTypeMap rtDataTypeMap[] = {
  /* cName, mwName, numElements, elemMapIndex, dataSize, slDataId, *
   * isComplex, isPointer */
  { "double", "real_T", 0, 0, sizeof(real_T), SS_DOUBLE, 0, 0 }
};
```

Because the index is `0` for `sig2_eg`, the index points to the first structure element in the array. You can infer that the data type of the signal is `double`. The value of `isComplex` is `0`, indicating that the signal is not complex. Rather than providing the data type information directly in the `rtwCAPI_Signals` structure, a level of indirection is introduced. The indirection allows multiple signals that share the same data type to point to one map structure, saving memory for each signal.

The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`, indicating the dimensions of the signal. Because this index is `1` for `sig2_eg`, the index points to the second element in the `rtDimensionMap` array:

```
/* Dimension Map - use dimensionMapIndex to access elements of ths structure*/
static const rtwCAPI_DimensionMap rtDimensionMap[] = {
  /* dataOrientation, dimArrayIndex, numDims, vardimsIndex */
  { rtwCAPI_SCALAR, 0, 2, 0 },

  { rtwCAPI_VECTOR, 2, 2, 0 },
...
};
```

From this structure, you can infer that this is a nonscalar signal having a dimension of `2`. The `dimArrayIndex` value, 2, provides an index into `rtDimensionArray`, found later in `rtwdemo_capi_capi.c`:

```
/* Dimension Array- use dimArrayIndex to access elements of this array */
static const uint_T rtDimensionArray[] = {
  1,                                    /* 0 */
  1,                                    /* 1 */
  2,                                    /* 2 */
...
};
```

The `fxpIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the signal. Your code can use the scaling information to compute the real-world value of the signal, using the equation `V=SQ+B`, where V is "real-world" (that is, base-10) value, S is user-specified slope, Q is "quantized fixed-point value" or "stored integer," and B is user-specified bias. For details, see "Scaling" (Fixed-Point Designer).

Because this index is `0` for `sig2_eg`, the signal does not have fixed-point information. A fixed-point map index of zero means that the signal does not have fixed-point information.

The `sTimeIndex` (sample-time index) provides the index to the `rtSampleTimeMap` array, found later in `rtwdemo_capi_capi.c`, indicating task information about the signal. If you log multirate signals or conditionally executed signals, the sampling information can be useful.

Note *model*_capi.c (or .cpp) includes rtw_capi.h. A source file that references the rtBlockSignals array also must include rtw_capi.h.

C API States

The rtwCAPI_States structure captures state information including the state name, address, block path, type (continuous or discrete), data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in rtwdemo_capi_capi.c that provides information on C API states for the top model in rtwdemo_capi:

```
/* Block states information */
static const rtwCAPI_States rtBlockStates[] = {
  /* addrMapIndex, contStateStartIndex, blockPath,
   * stateName, pathAlias, dWorkIndex, dataTypeIndex, dimIndex,
   * fixPtIdx, sTimeIndex, isContinuous
   */
  { 2, -1, "rtwdemo_capi/Delay1",
    "top_state", "", 0, 0, 0, 0, 0, 0 },

  {
    0, -1, (NULL), (NULL), (NULL), 0, 0, 0, 0, 0, 0
  }
};
```

Each array element, except the last, describes a state in the model. The final array element is a sentinel, with all elements set to null values. In this example, the C API code for the top model displays one state:

```
  { 2, -1, "rtwdemo_capi/Delay1",
    "top_state", "", 0, 0, 0, 0, 0, 0 },
```

This state, named top_state, is defined for the block rtwdemo_capi/Delay1. The value of isContinuous is zero, indicating that the state is discrete rather than continuous. The other fields correspond to the like-named signal equivalents described in “C API Signals” on page 62-9, as follows:

- The address of the signal is given by addrMapIndex, which, in this example, is 2. This is an index into the rtDataAddrMap array, found later in rtwdemo_capi_capi.c. Because the index is zero based, 2 corresponds to the third element in rtDataAddrMap, which is &rtwdemo_capi_DWork.top_state.
- The dataTypeIndex provides an index into the rtDataTypeMap array, found later in rtwdemo_capi_capi.c, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The dimIndex (dimensions index) provides an index into the rtDimensionMap array, found later in rtwdemo_capi_capi.c. The value 0 corresponds to the first entry, which is { rtwCAPI_SCALAR, 0, 2, 0 }.
- The fixPtIndex (fixed-point index) provides an index into the rtFixPtMap array, found later in rtwdemo_capi_capi.c, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

C API Root-Level Inputs and Outputs

The rtwCAPI_Signals structure captures root-level input/output information including the input/output name, address, block path, port number, data type information, dimensions information, fixed-

point information, and sample-time information. (This structure also is used for block output signals, as previously described in “C API Signals” on page 62-9.)

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API root-level inputs/outputs for the top model in `rtwdemo_capi`:

```
/* Root Inputs information */
static const rtwCAPI_Signals rtRootInputs[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 3, 0, "rtwdemo_capi/In1",
    "", 1, 0, 0, 0, 0 },

  { 4, 0, "rtwdemo_capi/In2",
    "", 2, 0, 0, 0, 0 },

  { 5, 0, "rtwdemo_capi/In3",
    "", 3, 0, 0, 0, 0 },

  { 6, 0, "rtwdemo_capi/In4",
    "", 4, 0, 0, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};

/* Root Outputs information */
static const rtwCAPI_Signals rtRootOutputs[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 7, 0, "rtwdemo_capi/Out1",
    "", 1, 0, 0, 0, 0 },

  { 8, 0, "rtwdemo_capi/Out2",
    "", 2, 0, 0, 0, 0 },

  { 9, 0, "rtwdemo_capi/Out3",
    "", 3, 0, 0, 0, 0 },

  { 10, 0, "rtwdemo_capi/Out4",
    "", 4, 0, 0, 0, 0 },

  { 11, 0, "rtwdemo_capi/Out5",
    "sig2_eg", 5, 0, 1, 0, 0 },

  { 12, 0, "rtwdemo_capi/Out6",
    "", 6, 0, 1, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

Note When you generate C++ code, the code generator does not provide information on C API root-level inputs/outputs.

For information about interpreting the values in the `rtwCAPI_Signals` structure, see the previous section “C API Signals” on page 62-9.

C API Parameters

The `rtwCAPI_BlockParameters` and `rtwCAPI_ModelParameters` structures capture parameter information including the parameter name, block path (for block parameters), address, data type information, dimensions information, and fixed-point information.

The `rtModelParameters` array contains entries for workspace variables that are referenced as tunable Simulink block parameters or Stateflow data of machine scope. For example, tunable parameters include `Simulink.Parameter` objects that use a storage class other than `Auto`. The Simulink Coder software assigns its elements only `NULL` or zero values in the absence of such data.

The setting that you select for the model configuration parameter **Default parameter behavior** determines how information is generated into the `rtBlockParameters` array in *`model`*`_capi.c` (or `.cpp`).

- If you set **Default parameter behavior** to `Tunable`, the `rtBlockParameters` array contains an entry for every modifiable parameter of every block in the model. However, if you use a MATLAB variable or a tunable parameter to specify a block parameter, the block parameter does not appear in `rtBlockParameters`. Instead, the variable or tunable parameter appears in `rtModelParameters`.
- If you set **Default parameter behavior** to `Inlined`, the `rtBlockParameters` array is empty. The Simulink Coder software assigns its elements only `NULL` or zero values.

The last member of each array is a sentinel, with all elements set to null values.

Here is the `rtBlockParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Individual block tuning is not valid when inline parameters is *
 * selected. An empty map is produced to provide a consistent     *
 * interface independent  of inlining parameters.                 *
 */
static const rtwCAPI_BlockParameters rtBlockParameters[] = {
  /* addrMapIndex, blockPath,
   * paramName, dataTypeIndex, dimIndex, fixPtIdx
   */
  {
    0, (NULL), (NULL), 0, 0, 0
  }
};
```

In this example, only the final, sentinel array element is generated, with all members of the structure `rtwCAPI_BlockParameters` set to `NULL` and zero values. This is because **Default parameter behavior** is set to `Inlined` by default for the `rtwdemo_capi` example model. If you set **Default parameter behavior** to `Tunable`, the block parameters are generated in the `rtwCAPI_BlockParameters` structure. However, MATLAB variables and tunable parameters appear in the `rtwCAPI_ModelParameters` structure.

Here is the `rtModelParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Tunable variable parameters */
static const rtwCAPI_ModelParameters rtModelParameters[] = {
  /* addrMapIndex, varName, dataTypeIndex, dimIndex, fixPtIndex */
  { 2, TARGET_STRING("Ki"), 0, 0, 0 },

  { 3, TARGET_STRING("Kp"), 0, 0, 0 },

  { 4, TARGET_STRING("p1"), 0, 2, 0 },

  { 5, TARGET_STRING("p2"), 0, 3, 0 },

  { 6, TARGET_STRING("p3"), 0, 4, 0 },
```

```
  { 0, (NULL), 0, 0, 0 }
};
```

In this example, the `rtModelParameters` array contains entries for each variable that is referenced as a tunable Simulink block parameter.

For example, the `varName` (variable name) of the fourth parameter is `p2`. The other fields correspond to the like-named signal equivalents described in "C API Signals" on page 62-9, as follows:

- The address of the fourth parameter is given by `addrMapIndex`, which, in this example, is `5`. This is an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`. Because the index is zero based, `5` corresponds to the sixth element in `rtDataAddrMap`, which is `rtP_p2`.
- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 3 corresponds to the fourth entry, which is `{ rtwCAPI_MATRIX_COL_MAJOR, 6, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

For more information about tunable parameter storage in the generated code, see "How Generated Code Stores Internal Signal, State, and Parameter Data" on page 36-49.

Map C API Data Structures to rtModel

The real-time model data structure encapsulates model data and associated information that describes the model fully. When you select the C API feature and generate code, the Simulink Coder code generator adds another member to the real-time model data structure generated in *`model`*`.h`:

```
/*
 * DataMapInfo:
 * The following substructure contains information regarding
 * structures generated in the model's C API.
 */
struct {
  rtwCAPI_ModelMappingInfo mmi;
} DataMapInfo;
```

This member defines `mmi` (for model mapping information) of type `struct rtwCAPI_ModelMappingInfo`. The structure is located in *`matlabroot`*`/rtw/c/src/rtw_modelmap.h`. The `mmi` substructure defines the interface between the model and the C API files. More specifically, members of `mmi` map the real-time model data structure to the structures in *`model`*`_capi.c` (or `.cpp`).

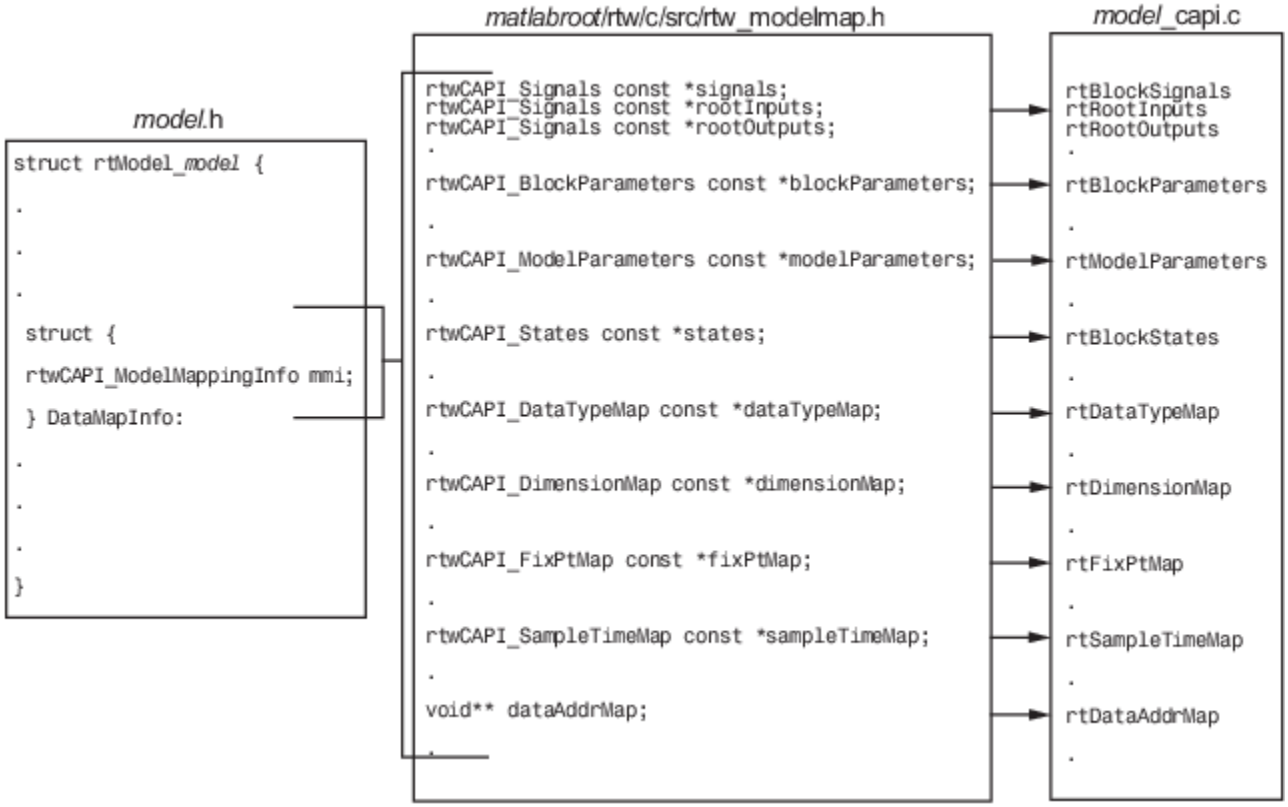
Initializing values of `mmi` members to the arrays accomplishes the mapping, as shown in "Map Model to C API Arrays of Structures" on page 62-16. Each member points to one of the arrays of structures in the generated C API file. For example, the address of the `rtBlockSignals` array of structures is allocated to the first member of the `mmi` substructure in *`model`*`.c` (or `.cpp`), using the following code in the `rtw_modelmap.h` file:

```
/* signals */
struct {
```

```
    rtwCAPI_Signals const *signals;     /* Signals Array */
    uint_T                numSignals;   /* Num Signals   */
    rtwCAPI_Signals const *rootInputs;  /* Root Inputs array */
    uint_T               numRootInputs; /* Num Root Inputs  */
    rtwCAPI_Signals const *rootOutputs; /* Root Outputs array */
    uint_T               numRootOutputs;/* Num Root Outputs  */
} Signals;
```

The model initialize function in *model*.c (or .cpp) performs the initializing by calling the C API initialize function. For example, the following code is generated in the model initialize function for example model rtwdemo_capi:

```
/* Initialize DataMapInfo substructure containing ModelMap for C API */
rtwdemo_capi_InitializeDataMapInfo(rtwdemo_capi_M);
```



Map Model to C API Arrays of Structures

Note This figure lists the arrays in the order that their structures appear in rtw_modelmap.h, which differs slightly from their generated order in *model*_capi.c.

Generate C API Data Definition File for Exchanging Data with a Target System

This example shows how to use the target-based C API to interface with generated code that represents signals, states, parameters, and root-level I/O.

Open Example Model

Open the example model `rtwdemo_capi`.

```
open_system('rtwdemo_capi');
```

The C API is useful for interacting with application data in generated code, without stopping program execution or recompiling the generated code. To use the C API interface, for a top model and its referenced models:

1. Set up a client/server protocol (such as TCP/IP or a dual-port memory connection) between your development computer and the target computer.

2. Select at least one of the C API model configuration parameters: **signals**, **parameters**, **states**, and **root-level I/O**.

3. Configure the data elements that you want to access with the C API with addressable storage classes.

C API configuration settings for the top model and referenced models must match.

The code generator places the C API interface in the file *`model`*`_capi.c`. Depending on your configuration settings, the data can represent signals, states, parameters, and root-level I/O configured with addressable storage classes. The file includes structures that provide an interface to data properties.

C API Limitations

The C API feature has the following limitations.

- The C API does not support the following values for the `CodeFormat` TLC variable:
  - `S-Function`
  - `Accelerator_S-Function` (for accelerated simulation)
- For ERT-based targets, the C API requires that support for floating-point code be enabled.
- Local block output signals are not supported.
- Local Stateflow parameters are not supported.
- The following custom storage class objects are not supported:
  - Objects without the package `csc_registration` file
  - Grouped custom storage classes
  - Objects defined by using macros
  - `BitField` objects
  - `FileScope` objects
- Customized data placement is disabled when you are using the C API. The interface looks for global data declaration in *`model`*`.h` and *`model`*`_private.h`. Declarations placed in any other file by customized data placement result in code that does not compile.

Note Custom Storage Class objects work in code generation, only if you use the ERT system target file and clear the model configuration parameter **Ignore custom storage classes**.

See Also

Related Examples

- “Access Signal, State, and Parameter Data During Execution”
- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Use C API to Access Model Signals and States

This example helps you get started writing application code to interact with model signals and states. To get started writing application code to interact with model parameters, see “Use C API to Access Model Parameters” on page 62-24.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with the Simulink Coder generated code into an executable. The target-based application code accesses the C API structure arrays in *model*`_capi.c` (or `.cpp`). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files `rtw_modelmap.h` and `rtw_capi.h`, located in *matlabroot*`/rtw/c/src` (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that logs global signals and states in a model to a text file. This code is intended as a starting point for accessing signal and state addresses. You can extend the code to perform signal logging and monitoring, state logging and monitoring, or both.

This example uses the following macro and function interfaces:

- `rtmGetDataMapInfo` macro

  Accesses the model mapping information (MMI) substructure of the real-time model structure. In the following macro call, `rtM` is the pointer to the real-time model structure in *model*`.c` (or `.cpp`):

  ```
  rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
  ```

- `rtmGetTPtr` macro

  Accesses the absolute time information for the base rate from the timing substructure of the real-time model structure. In the following macro call, `rtM` is the pointer to the real-time model structure in *model*`.c` (or `.cpp`):

  ```
  rtmGetTPtr(rtM)
  ```

- Custom functions `capi_StartLogging`, `capi_UpdateLogging`, and `capi_TerminateLogging`, provided via the files `rtwdemo_capi_datalog.h` and `rtwdemo_capi_datalog.c`. These files are located in *matlabroot*`/toolbox/rtw/rtwdemos` (open).

  - `capi_StartLogging` initializes signal and state logging.
  - `capi_UpdateLogging` logs a signal and state value at each time step.
  - `capi_TerminateLogging` terminates signal and state logging and writes the logged values to a text file.

  You can integrate these custom functions into generated code for a model by using these methods:

  - **Custom Code** of the Model Configuration Parameters dialog box.
  - Custom Code library blocks
  - TLC custom code functions

  This tutorial uses the **Custom Code** pane of the Model Configuration Parameters dialog box and the System Outputs block from the Custom Code library to insert calls to the custom functions into *model*`.c` (or `.cpp`), as follows:

- `capi_StartLogging` is called in the *`model`*`_initialize` function.
- `capi_UpdateLogging` is called in the *`model`*`_step` function.
- `capi_TerminateLogging` is called in the *`model`*`_terminate` function.

The following excerpts of generated code from *`model`*`.c` (rearranged to reflect their order of execution) show how the function interfaces are used.

```
void rtwdemo_capi_initialize(void)
{
...
  /* user code (Initialize function Body) */

  /* C API Custom Logging Function: Start Signal and State logging via C API.
   * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    printf("** Started state/signal logging via C API **\n");
    capi_StartLogging(MMI, MAX_DATA_POINTS);
  }
...
}
...
/* Model step function */
void rtwdemo_capi_step(void)
{
...
  /* user code (Output function Trailer) */

  /* System '<Root>' */

  /* C API Custom Logging Function: Update Signal and State logging buffers.
   * capi_UpdateLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
  }
...
}
...
/* Model terminate function */
void rtwdemo_capi_terminate(void)
{
  /* user code (Terminate function Body) */

  /* C API Custom Logging Function: Dump Signal and State buffers into a text file.
   * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
    printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
  }
}
```

The following procedure illustrates how you can use the C API macro and function interfaces to log global signals and states in a model to a text file.

1. At the MATLAB command line, enter `rtwdemo_capi` to open the example model.
2. Save the top model `rtwdemo_capi` and the referenced model `rtwdemo_capi_bot` to the same writable work folder.
3. Open the Configuration Parameters dialog box.
4. If you are licensed for Embedded Coder software and you want to use the `ert.tlc` system target file instead of the default `grt.tlc`, change the setting of model configuration parameter **System target file**. Make sure that you also select `ert.tlc` for the referenced model `rtwdemo_capi_bot`.
5. For the top model, confirm these model configuration parameter settings:

a Select parameters **Generate C API for signals**, **Generate C API for states**, and **Generate C API for parameters**.

b If you are using the `ert.tlc` system target file, select **Support complex numbers**

c Select **MAT-file logging**.

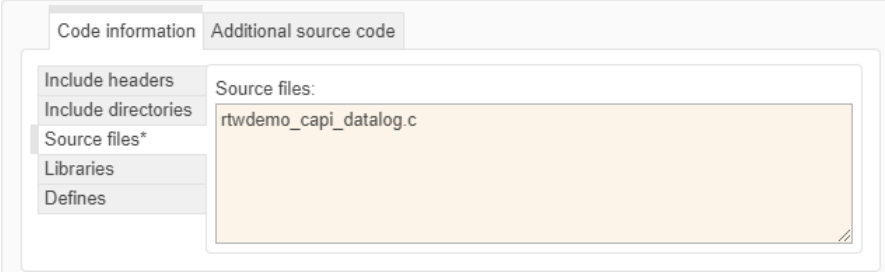
d Click **Apply**.

e Update configuration parameter settings in the referenced model, `rtwdemo_capi_bot`, to match changes you made in the top model.

6 Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane. On the **Code information** tab, click **Include directories**. The **Include directories** input field is displayed.

7 In the **Include directories** field, type *`matlabroot`*`/toolbox/rtw/rtwdemos`, where *`matlabroot`* represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)

8 On the same tab, click **Source files**, and type `rtwdemo_capi_datalog.c`.



9 On the **Additional source code** tab, click **Additional code**, and type or copy and paste the following include statement:

```
#include "rtwdemo_capi_datalog.h"
```

10 On the same tab, in the **Initialize code** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Start Signal and State logging via C API.
 * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
  printf("** Started state/signal logging via C API **\n");
  capi_StartLogging(MMI, MAX_DATA_POINTS);
}
```

Note If you renamed the top model `rtwdemo_capi`, update the name `rtwdemo_capi_M` in the application code to reflect the new model name.

11 On the same tab, in the **Terminate code** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Dump Signal and State buffers into a text file.
 * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
}
```

Click **Apply**.

12 In the MATLAB Command Window, enter `custcode` to open the Simulink Coder Custom Code library. At the top level of the `rtwdemo_capi` model, add a System Outputs block.

13 Double-click the System Outputs block to open the System Outputs Function Custom Code dialog box. In the **System Outputs Function Exit Code** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Update Signal and State logging buffers.
 * capi_UpdateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
  capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
}
```

If you renamed the top model `rtwdemo_capi`, update two instances of the name `rtwdemo_capi_M` in the application code to reflect the new model name.

Click **OK**.

14 Clear model configuration parameter **Generate code only**.

Build the model and generate an executable program. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

15 In the MATLAB Command Window, enter the command `!rtwdemo_capi` to run the executable file. During execution, signals and states are logged using the C API and then written to the text file `rtwdemo_capi_ModelLog.txt` in your current working folder.

```
>> !rtwdemo_capi

** starting the model **
** Started state/signal logging via C API **
** Logging 2 signal(s) and 1 state(s). In this demo, only scalar named
 signals/states are logged **
** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **
```

16 Examine the text file in the MATLAB editor or other text editor. Here is an excerpt of the signal and state logging output.

```
******** Signal Log File ********

Number of Signals Logged: 2
Number of points (time steps) logged: 51

Time           bot_sig1 (Referenced Model)             top_sig1
0              70                                      4
0.2            70                                      4
0.4            70                                      4
0.6            70                                      4
0.8            70                                      4
1              70                                      4
1.2            70                                      4
1.4            70                                      4
1.6            70                                      4
1.8            70                                      4
2              70                                      4
...

******** State Log File ********

Number of States Logged: 1
Number of points (time steps) logged: 51

Time           bot_state (Referenced Model)
0              0
0.2            70
0.4            35
0.6            52.5
```

```
0.8             43.75
1               48.13
1.2             45.94
1.4             47.03
1.6             46.48
1.8             46.76
2               46.62
...
```

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Use C API to Access Model Parameters

This example helps you get started writing application code to interact with model parameters. To get started writing application code to interact with model signals and states, see “Use C API to Access Model Signals and States” on page 62-19.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with generated code into an executable program. The target-based application code accesses the C API structure arrays in *model*`_capi.c` (or `.cpp`). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files `rtw_modelmap.h` and `rtw_capi.h`, located in *matlabroot*`/rtw/c/src` (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that prints the parameter values of tunable parameters in a model to the standard output. This code is intended as a starting point for accessing parameter addresses. You can extend the code to perform parameter tuning. The application:

- Uses the `rtmGetDataMapInfo` macro to access the mapping information in the `mmi` substructure of the real-time model structure

  ```
  rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
  ```

  where `rtM` is the pointer to the real-time model structure in *model*`.c` (or `.cpp`).
- Uses `rtwCAPI_GetNumModelParameters` to get the number of model parameters in mapped C API:

  ```
  uint_T nModelParams = rtwCAPI_GetNumModelParameters(mmi);
  ```
- Uses `rtwCAPI_GetModelParameters` to access the array of model parameter structures mapped in C API:

  ```
  rtwCAPI_ModelParameters* capiModelParams = \
   rtwCAPI_GetModelParameters(mmi);
  ```
- Loops over the `capiModelParams` array to access individual parameter structures. A call to the function `capi_PrintModelParameter` displays the value of the parameter.

The example application code is provided below:

```
{
/* Get CAPI Mapping structure from Real-Time Model structure */
rtwCAPI_ModelMappingInfo* capiMap = \
&(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);

/* Get number of Model Parameters from capiMap */
uint_T nModelParams = rtwCAPI_GetNumModelParameters(capiMap);
printf("Number of Model Parameters: %d\n", nModelParams);

/* If the model has Model Parameters, print them using the
application capi_PrintModelParameter */
if (nModelParams == 0) {
   printf("No Tunable Model Parameters in the model \n");
}
else {
```

```
   unsigned int idx;

   for (idx=0; idx < nModelParams; idx++) {
      /* call print utility function */
      capi_PrintModelParameter(capiMap, idx);
   }
}
}
```

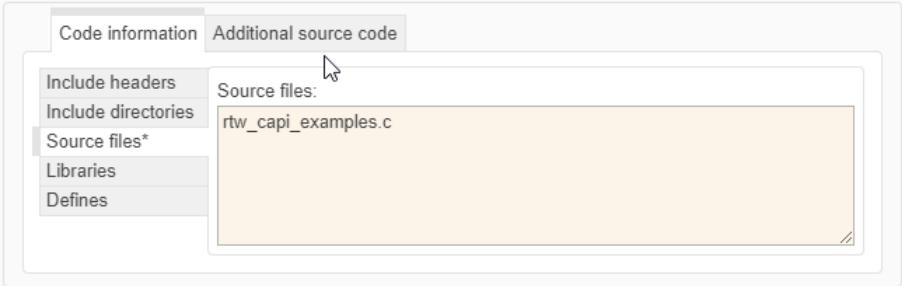
The print utility function is located in *matlabroot*/rtw/c/src/rtw_capi_examples.c. This file contains utility functions for accessing the C API structures.

To become familiar with the example code, try building a model that displays the tunable block parameters and MATLAB variables. You can use rtwdemo_capi, the C API example model. The following steps apply to both grt.tlc and ert.tlc system target files, unless otherwise indicated.

1. At the MATLAB command line, enter rtwdemo_capi to open the example model.
2. Save the top model rtwdemo_capi and the referenced model rtwdemo_capi_bot to the same writable work folder.
3. If you are licensed for Embedded Coder software and you want to use the ert.tlc system target tile instead of the default grt.tlc, use model configuration parameter **System target file** to select an ert.tlc system target file. Make sure that you also select ert.tlc for the referenced model rtwdemo_capi_bot.
4. Confirm these model configuration parameter settings:
   a. Select **Generate C API for parameters**.
   b. If you are using the ert.tlc system target file, select **Support complex numbers**.
   c. Select **MAT-file logging**.
   d. Click **Apply**.
   e. Update configuration parameter settings in the referenced model, rtwdemo_capi_bot, to match changes you made in the top model.
5. Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane. On the **Additional source code** tab, click **Initialize code**. The **Initialize code** input field is displayed.
6. In the **Initialize code** input field, type or copy and paste the example application code shown above step 1. This embeds the application code in the *model*_initialize function.

   Note If you renamed the top model rtwdemo_capi, update the name rtwdemo_capi_M in the application code to reflect the new model name.

7. On the **Code information** tab, click **Include directories**, and type *matlabroot*/rtw/c/src, where *matlabroot* represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)
8. On the same tab, click **Source files**, and type rtw_capi_examples.c.



Click **Apply**.

9 Clear model configuration parameter **Generate code only**.

Build the model and generate an executable program. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

10 In the MATLAB Command Window, enter `!rtwdemo_capi` to run the executable file. Running the program displays parameter information in the Command Window.

```
>> !rtwdemo_capi

** starting the model **
Number of Model Parameters: 5
Ki =
    7
Kp =
    4
p1 =
    0.15
    0.36
    0.81
p2 =
    0.09    0.75    0.57
    0.13    0.96    0.059
p3 =
ans(:,:,1) =
    0.23    0.82    0.04    0.64
    0.35    0.01    0.16    0.73

ans(:,:,2) =
    0.64    0.54    0.74    0.68
    0.45    0.29    0.18    0.18
```

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

ASAP2 Data Measurement and Calibration in Simulink Coder

- “Export ASAP2 File for Data Measurement and Calibration” on page 63-2
- “Calibrate Generated Code and Monitor Signals” on page 63-11

Export ASAP2 File for Data Measurement and Calibration

"Generate ASAP2 and CDF Calibration Files" on page 64-2 is recommended way of generating the ASAP2 file, because it allows you to generate the latest version of ASAP2 file as per ASAM standards and provides the **Generate Calibration Files** tool.

Generation of ASAP2 File Using Generate Calibration Files Tool

- "Open Generate Calibration Files Tool" on page 63-2
- "ASAP2 File Generation" on page 63-3

Open Generate Calibration Files Tool

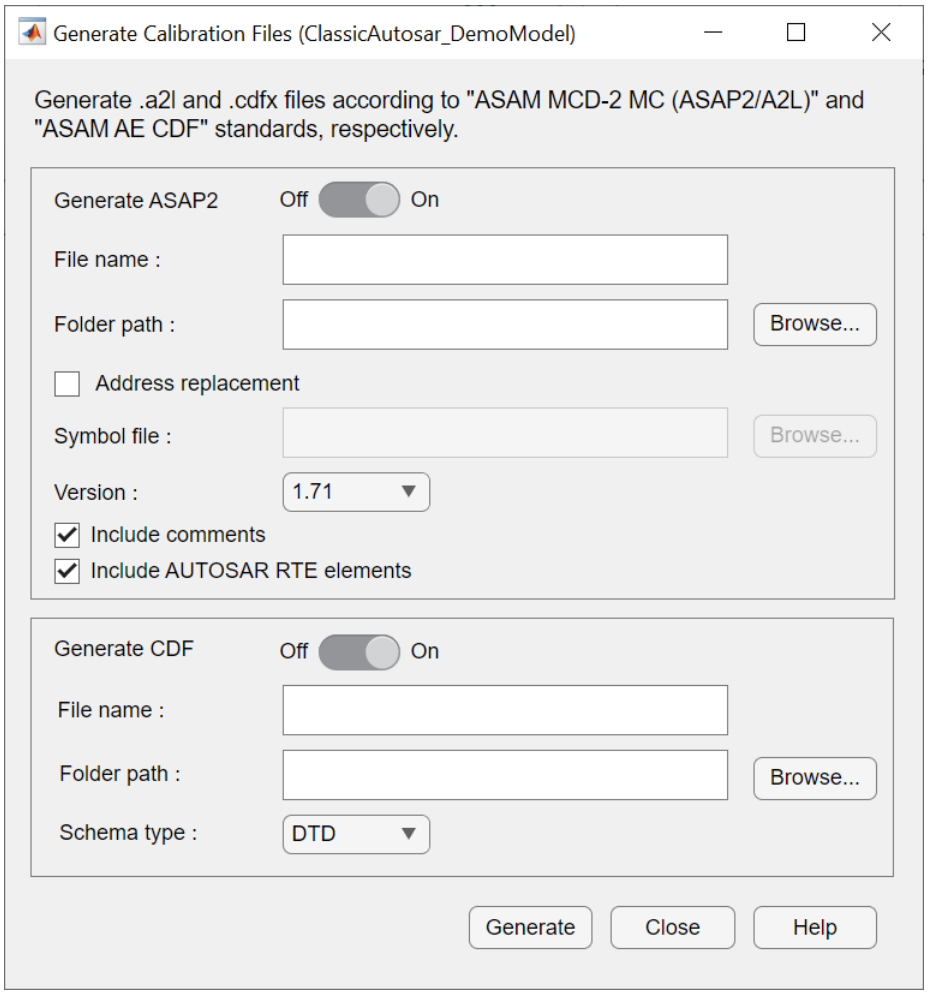
1. Open the model.
2. Open the Embedded Coder app.
3. Generate code for the model.
4. On the **C Code** tab, select **Share** > **Generate Calibration Files**.

Note For AUTOSAR models, select **Autosar** > **Share** > **Generate Calibration Files**.

Using the tool, you can generate A2L and `cdfx` files at the same time for a model. You can use the tool for models configured with an ERT-based, GRT-based, Simulink Real-Time, Classic AUTOSAR, or Adaptive AUTOSAR system target file. The tool remembers the last used settings, such as the version of the ASAP2 file, include or exclude comments, turn off or on the ASAP2 file and CDF file generation. Simulink saves these settings in the MATLAB preferences. The settings will override the default settings.

For Adaptive AUTOSAR and Simulink Real-Time system target files, you can generate the ASAP2 file for the C++ language. For GRT- and ERT- based system target files, you can generate an ASAP2 file only for C.

For classic AUTOSAR models, the tool allows you to include or exclude RTE elements into the ASAP2 file by using the `Include AUTOSAR RTE elements` option.



ASAP2 File Generation

ASAP2 defines the description format of the internal ECU variables used for tuning parameters and measuring signal data as an application executes. To generate an ASAP2 file:

1. By default the **Generate ASAP2** option is enabled. If it is not enabled, slide the **Generate ASAP2** slider to the on position to enable A2L generation.
2. To change the name of the A2L file, in the **File name** field, specify the name.
3. By default, the code generator uses the build folder for the folder path of the selected model. If you want to specify a different folder path for the A2L file, you can do that in the **Folder path** field.
4. To replace ECU addresses in the A2L file, in the **Symbol file** field, provide a symbol file associated with the application executable. Otherwise, clear the **Address replacement** field.

   The tool supports an ELF, PDB, DWARF, or executable file as a symbol file to replace the ECU addresses.
5. Set **Version** to the required version of the A2L file to be generated. The default version is 1.71.
6. To exclude comments in the A2L file, clear **Include comments**.

7. For Classic AUTOSAR models, to exclude RTE elements in the A2L file, clear **Include AUTOSAR RTE elements**.
8. Click **Generate**.

To generate the A2L file programmatically, use the `coder.asap2.export` function.

Generation of ASAP2 File Using ASAP2Interface Parameter

- "What You Should Know" on page 63-4
- "System Target Files Supporting ASAP2" on page 63-4
- "Define ASAP2 Information" on page 63-5
- "Generate an ASAP2 File" on page 63-8
- "Structure of ASAP2 File" on page 63-10

The `ASAP2Interface` parameter is not present in the configuration parameters anymore. In a coming release the parameter will be completely removed. You can generate the ASAP2 file using **Generate Calibration Files** tool, or `coder.asap2.export` function. For more information, see "Generate ASAP2 and CDF Calibration Files" on page 64-2.

The ASAM MCD-2 MC standard, also known as ASAP2, is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostic systems. For more information on ASAM and the ASAM MCD-2 MC (ASAP2) standard, see the ASAM website at `www.asam.net`.

The code generator enables you to export an ASAP2 file containing information about your model during the code generation process.

Simulink Coder support for ASAP2 file generation is version-neutral. By default, the software generates ASAP2 version 1.31 format, but the generated model information is generally compatible with other ASAP2 versions. ASAP2 file generation is also neutral with respect to the specific needs of ASAP2 measurement and calibration tools. The software provides customization APIs that you can use to customize ASAP2 file generation to generate an ASAP2 version and to meet the specific needs of your ASAP2 tools.

What You Should Know

To make use of ASAP2 file generation, you should become familiar with:

- ASAM and the ASAP2 standard and terminology. See the ASAM Web site at `www.asam.net`.
- Configuring model data elements for code generation. See **Code Mappings Editor – C**.
- Storage and representation of signals and parameters in generated code. See "Data Representation in Generated Code".
- If you have Embedded Coder, see "Data Representation in Generated Code".

System Target Files Supporting ASAP2

You can use ASAP2 file generation with varying system target file configurations. For example, GRT- and ERT- based configurations enables you to generate an ASAP2 file as part of the code generation and build process.

For Adaptive AUTOSAR target and Simulink Real-Time targets, you can generate the ASAP2 file for C ++ language. For ERT and GRT targets, you can generate ASAP2 file only for language C and language C++ is not supported.

For procedures for generating ASAP2 files by using the system target file configurations, see ASAP2 interface.

Define ASAP2 Information

- “Define ASAP2 Information for Parameters and Signals” on page 63-5
- “Memory Address Attribute” on page 63-5
- “Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)” on page 63-6
- “Define ASAP2 Information for Lookup Tables” on page 63-7

Define ASAP2 Information for Parameters and Signals

The ASAP2 file generation process requires information about parameters and signals in your model. Some of this information is contained in the model itself. Use Simulink data objects in a workspace or data dictionary to complete the ASAP2 file generation process. In some cases, the use of workspace objects is optional.

Use the Model Data Editor and built-in Simulink data objects to provide information, such as using `Simulink.Signal` objects to provide MEASUREMENT information and `Simulink.Parameter` objects to provide CHARACTERISTIC information. You can use data objects from data classes that are derived from `Simulink.Signal` and `Simulink.Parameter` to provide the information. See “Data Objects” and **Model Data Editor**.

This table contains the minimum set of data attributes required for ASAP2 file generation. Some data attributes are defined in the model. Others attributes are supplied in the properties of objects. For attributes that are defined in `Simulink.Signal` or `Simulink.Parameter` objects, the table lists the associated property name.

Data Attribute	Defined In	Property Name
Name (symbol)	Model and data object	Inherited from the handle of the data object to which parameter or signal name resolves
Description	Data object	`Description`
Data type	Model or data object	`DataType`
Scaling (if fixed-point data type)	Model or data object	`DataType`
Minimum allowable value	Model or data object	`Min`
Maximum allowable value	Model or data object	`Max`
Unit	Model or data object	`Unit`
Memory address (optional)	Model or data object	`MemoryAddress_ASAP2`

Memory Address Attribute

If the memory address attribute is unknown before code generation, the code generator inserts `ECU Address` placeholder text in the generated ASAP2 file. You can substitute an actual address for the placeholder by postprocessing the generated file. For an example, see the file *`matlabroot/`*

`toolbox/rtw/targets/asap2/asap2/asap2post.m`. `asap2post.m` parses through the linker map file that you provide and replaces the `ECU Address` placeholders in the ASAP2 file with the actual memory addresses. Because linker map files vary from compiler to compiler, you might need to modify the regular expression code in `asap2post.m` to match the format of the linker map that you use.

If you have Embedded Coder installed on your system, and if you are generating Executable and Linkable Format (ELF) or Program Database (PDB) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate ECU address replacement. For more information, see "Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)" on page 63-6.

If the memory address attribute is known before code generation, it can be defined in the data item or data object. By default, the `MemoryAddress_ASAP2` property does not exist in the `Simulink.Signal` or `Simulink.Parameter` data object classes. If you want to add the attribute, add a property called `MemoryAddress_ASAP2` to a custom class that is a subclass of the `Simulink` or `ASAP2` class. For information on subclassing Simulink data classes, see "Define Data Classes".

This table indicates the Simulink object properties that have replaced the ASAP2 object properties of the previous release.

Differences Between ASAP2 and Simulink Parameter and Signal Object Properties

ASAP2 Object Properties (Previous)	Simulink Object Properties (Current)
LONGID_ASAP2	Description
PhysicalMin_ASAP2	Min
PhysicalMax_ASAP2	Max
Units_ASAP2	Unit

Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)

If you have Embedded Coder installed on your system, and if you are generating Executable and Linkable Format (ELF) or Program Database (PDB) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate the replacement of `ECU Address` placeholder memory address values with actual addresses in the generated ASAP2 file.

If the memory address attribute is unknown before code generation, the code generator inserts `ECU Address` placeholder text in the generated ASAP2 file, as shown

```
/begin CHARACTERISTIC
  /* Name            */ Ki
  /* Long Identifier */ ""
  /* Type            */ VALUE
  /* ECU Address     */ 0x0000 /* @ECU_Address@Ki@ */
```

To substitute actual addresses for the `ECU Address` placeholders, process the generated ASAP2 file by using the `rtw.asap2SetAddress` function. The syntax is:

```
rtw.asap2SetAddress(ASAP2File,InfoFile)
```

The arguments are character vectors specifying the name of the generated ASAP2 file and the name of the generated executable ELF file, PDB file from Microsoft toolchain, or DWARF debug information files for the model. When called, `rtw.asap2SetAddress` extracts the actual ECU address from the specified ELF, PDB, or DWARF file and replaces the placeholder in the ASAP2 file with the actual address, for example:

```
/begin CHARACTERISTIC
  /* Name            */ Ki
  /* Long Identifier */ ""
  /* Type            */ VALUE
  /* ECU Address     */ 0x40009E60
```

Define ASAP2 Information for Lookup Tables

Simulink Coder software generates ASAP2 descriptions for lookup table data and its breakpoints. The software represents 1-D table data as `CURVE` information, 2-D table data as `MAP` information, and breakpoints as `AXIS_DESCR` and `AXIS_PTS` information. You can model lookup tables by using one of the following Simulink Lookup Table blocks:

- Interpolation Using Prelookup — 1 and 2 dimensions
- 1–D Lookup Table
- 2–D Lookup Table
- n-D Lookup Table — 1 and 2 dimensions

The software supports these types of lookup table breakpoints (axis points).

Breakpoint Type	Information Generated
Tunable and shared among multiple table axes (common axis)	`COM_AXIS`
Fixed and nontunable (fixed axis)	One of these variants of `FIX_AXIS`: • `FIX_AXIS_PAR` if breakpoints are integers with equidistant spacing and the equidistant spacing is a power of two • `FIX_AXIS_PAR_DIST` if breakpoints are integers with equidistant spacing • `FIX_AXIS_PAR_LIST` if breakpoints are integers with non-equidistant spacing
Tunable but not shared among multiple tables (standard axis)	`STD_AXIS`

When you configure the blocks for ASAP2 code generation:

- For table data, use a `Simulink.Parameter` data object with a non-`Auto` storage class.
- For tunable breakpoint data that is shared among multiple table axes (`COM_AXIS`), use a `Simulink.Parameter` data object that has a non-`Auto` storage class.
- For fixed, nontunable breakpoint data (`FIX_AXIS`), use workspace variables or arrays specified in the block parameters dialog box. The breakpoints must be stored as integers in the code. The data type must be a built-in integer type (`int8`, `int16`, `int32`, `uint8`, `uint16`, or `uint32`), a fixed-point data type, or an equivalent alias type.
- For tunable breakpoint data that is not shared among multiple tables (STD_AXIS):

  1. Create a `Simulink.Bus` object to define the `struct` packaging (names and order of the fields). The fields of the parameter structure must correspond to the lookup table data and each axis of the lookup table block. For example, in an n-D Lookup Table block that has 2 dimensions, the structure must contain only three fields. This bus object describes the record layout for the lookup characteristic.

2. Create a `Simulink.Parameter` object to represent a tunable parameter.
3. Create table and axis values.
4. Optionally, specify the **Units**, **Minimum**, and **Maximum** properties for the parameter object.

Here is an example of an n-D Lookup Table record generated into an ASAP2 file in Standard Axis format:

```
/begin CHARACTERISTIC
  /* Name */  STDAxisParam
  ...
  /* Record Layout */   Lookup1D_X_WORD_Y_FLOAT32_IEEE
  ...
  begin AXIS_DESCR
    /* Description of X-Axis Points */
    /* Axis Type */      STD_AXIS
    ...
  /end AXIS_DESCR
/end CHARACTERISTIC

/begin RECORD_LAYOUT Lookup1D_X_WORD_Y_FLOAT32_IEEE
  AXIS_PTS_X 1 WORD INDEX_INCR DIRECT
  FNC_VALUES 2 FLOAT32_IEEE COLUMN_DIR DIRECT
/end   RECORD_LAYOUT
```

Generate an ASAP2 File

- “Generating ASAP2 Files” on page 63-8
- “Use GRT- or ERT- Based System Target File” on page 63-8
- “Generate ASAP2 Files for Referenced Models” on page 63-9
- “Merge ASAP2 Files for Top and Referenced Models” on page 63-9

Generating ASAP2 Files

Use the Generic Real-Time system target file or a Embedded Coder system target file to generate an ASAP2 file as part of the code generation and build process.

Generate ASAP2 file by using the system target files that have built-in ASAP2 support.

Use GRT- or ERT- Based System Target File

To generate the ASAP2 data definition for a model by using a Generic Real-Time(GRT) system target file or Embedded Coder system target file:

1. Use the **Code Mappings Editor – C** to apply storage classes to signals, block states, and model parameters, as described in “C Code Generation Configuration for Model Interface Elements” on page 33-2.

   Parameters are not represented in the ASAP2 file if one or more of the following conditions exist:

   - You apply storage class `Auto`, `FileScope`, `Struct`, or `Default` (through `Model default`).
   - You apply storage class through `Embedded Coder Dictionary`.
   - You apply a storage class that causes the code generator to generate a macro or non-addressable variable.

2. Use the Property Inspector to configure the remaining storage class properties for each data element.
3. Set the model configuration parameter **System target file** to a GRT- or ERT- based system target file.
4. Enable the configuration parameter **ASAP2 interface** with the command

   ```
   set_param('model_Name','GenerateASAP2',true);
   ```

   Note You can no longer enable generation of ASAP2 files from the Model Configuration Parameter dialog box. As "Generate ASAP2 and CDF Calibration Files" on page 64-2 describes, use the `Generate Calibration Files` tool, or use the command line model configuration parameter `GenerateASAP2`.
5. Select parameter **Generate code only**.
6. Click **Apply**.
7. Generate code.

   The code generator writes the ASAP2 file to the build folder. By default, the file is named *`model`*`.a2l`, where *`model`* is the name of the model. The ASAP2 setup file controls the ASAP2 file name. For details, see "Customize Generated ASAP2 File" on page 92-2.

Generate ASAP2 Files for Referenced Models

The build process can generate an ASAP2 file for each referenced model in a model reference hierarchy. In the generated ASAP2 file, MEASUREMENT records represent signals and states inside the referenced model.

To generate ASAP2 files for referenced models, select ASAP2 file generation for the top model and for each referenced model in the reference hierarchy.

Merge ASAP2 Files for Top and Referenced Models

Use function `rtw.asap2MergeMdlRefs` to merge the ASAP2 files generated for top and referenced models. The function has this syntax:

```
[status,info] = rtw.asap2MergeMdlRefs(topModelName,asap2FileName)
```

- `topModelName` is the name of the model containing one or more referenced models.
- `asap2FileName` is the name that you specify for the merged ASAP2 file.
- *Optional*: `status` returns false (logical 0) if the merge completed and returns true (logical 1) otherwise.
- *Optional*: `info` returns additional information about merge failure if `status` is true. Otherwise, it returns an empty character vector.

Consider the following example.

```
[status,info] = rtw.asap2MergeMdlRefs('myTopMdl','merged.a2l')
```

This command merges the ASAP2 files generated for the top model `myTopMdl` and its referenced models in the file `merged.a2l`.

Structure of ASAP2 File

This table outlines the basic structure of the ASAP2 file and describes the Target Language Compiler (TLC) functions and files used to create each part of the file:

- Static parts of the ASAP2 file are shown in **bold**.
- Function calls are indicated by `%<FunctionName()>`.

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
File header	`%<ASAP2UserFcnWriteFileHead()>`	`asap2userlib.tlc`
`/begin PROJECT ""`	**`/begin PROJECT`** `"%<ASAP2ProjectName>"`	`asap2setup.tlc`
`/begin HEADER ""` HEADER contents	**`/begin HEADER`**`"%<ASAP2HeaderName>"` `%<ASAP2UserFcnWriteHeader()>`	`asap2setup.tlc` `asap2userlib.tlc`
`/end HEADER`	**`/end HEADER`**	`asap2userlib.tlc`
`/begin MODULE ""` MODULE contents:	**`/begin MODULE`** `"%<ASAP2ModuleName>"}`	`asap2setup.tlc` `asap2userlib.tlc`
- A2ML - MOD_PAR - MOD_COMMON ...	`%<ASAP2UserFcnWriteHardwareInterface()>`	`asap2userlib.tlc`
Model-dependent MODULE contents:	`%<SLibASAP2WriteDynamicContents()>` Calls user-defined functions:	`asap2lib.tlc`

- RECORD_LAYOUT
- CHARACTERISTIC
- ParameterGroups
- ModelParameters
	`...WriteRecordLayout_TemplateName()`
	`...WriteCharacteristic_TemplateName()`
	`...WriteCharacteristic_Scalar()`
		`user/templates/...`

- MEASUREMENT - ExternalInputs - BlockOutputs	`...WriteMeasurement()`	`asap2userlib.tlc`
- COMPU_METHOD	`...WriteCompuMethod()`	`asap2userlib.tlc`
`/end MODULE`	**`/end MODULE`**	
File footer/tail	`%<ASAP2UserFcnWriteFileTail()>`	`asap2userlib.tlc`

See Also

Related Examples

- “Generate ASAP2 and CDF Calibration Files”
- “Create Tunable Calibration Parameter in the Generated Code” on page 36-112
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Calibrate Generated Code and Monitor Signals

When you create an external mode target application that uses an ASAM MCD-1 XCP communication channel, you can also generate an ASAP2 file that provides a description of the Simulink Coder Universal Measurement and Calibration Protocol (XCP) server configuration. The description enables third-party calibration tools, such as ETAS® INCA and Vector CANape®, to connect to the external mode target application. You can use the third-party tools to calibrate generated code and monitor signals in environments supported by the third-party tools, for example, a vehicle, a test bench, or a virtual environment on a PC.

Configure your model to generate an ASAP2 file that provides a description of the XCP server configuration:

1. On the **Apps** tab, click **Run on Custom Hardware**, which displays the **Hardware** tab.
2. In the **Hardware** section, in the **System Target File Description** field, select `Embedded Coder (ert.tlc)`.
3. In the **Prepare** section, click **Hardware Settings**, which opens the Configuration Parameters dialog box.
4. Clear **Generate code only**.
5. Specify **Toolchain settings** for your development computer.
6. Select **External mode**.
7. Set **Transport layer** to `XCP on TCP/IP` or `XCP on Serial`.
8. For **Mex-file arguments**, specify the required arguments. For example:
   - For `XCP on TCP/IP`, specify the IP address of the target processor and the port number of the TCP/IP server.
   - For `XCP on Serial`, specify the serial port ID and the baud value.
9. Click **OK**.

To generate the target application and the ASAP2 file:

1. On the **Hardware** tab, in the **Run on Hardware** section, click **Monitor & Tune**.
2. Under **Step By Step Commands**, click **Build for Monitoring**.
3. Use **C Code** > **Share** > **Generate Calibration Files** tool to generate the ASAP2 file. Or generate the file by using this command:

   ```
   coder.asap2.export('modelName','MapFile','symbolFileName');
   ```
4. In the code generation folder, the build process generates the XCP external mode target application *`modelName`*`.exe` and the **Generate Calibration Files** tool generates ASAP2 file *`modelName`*`.a2l`. The ASAP2 file contains an `IF_DATA XCP` block that describes the Simulink Coder XCP server configuration.

   ```
   …
   /begin IF_DATA XCP
     /begin PROTOCOL_LAYER
     …
     /end PROTOCOL_LAYER
     …
     /begin XCP_ON_TCP_IP
     …
   ```

```
    /end XCP_ON_TCP_IP
  /end IF_DATA
  …
```

Import the ASAP2 file into your third-party tool and manually run the external mode application on your target hardware. Through XCP, connect the third-party tool to the target application. Then, using that tool, perform data acquisition and calibration of the generated code.

See Also

`coder.asap2.export`

More About

- “Generate ASAP2 and CDF Calibration Files” on page 64-2
- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Customize XCP Server Software” on page 60-45

Generate ASAP2 and CDF Calibration Files

Generate ASAP2 and CDF Calibration Files

Generate and customize an ASAP2 file and a CDF (Calibration Data Format) file for a model by using the **Generate Calibration Files** tool. By default, the tool generates the latest supported version of the A2L file according to ASAM MCD-2 MC standards. You can choose to generate a different version. You can customize the options for calibration files, such as the version for the A2L file and the schema type, for the CDF file. For more information on the ASAM MCD-2 MC (ASAP2) standard and the ASAM CDF Standards, see the ASAM website at `www.asam.net`.

Open Generate Calibration Files Tool

1. Open the model.
2. Open the Embedded Coder app.

   Note For AUTOSAR models, open the AUTOSAR Component Designer app. And for Simulink Real-Time models, open the Simulink Coder app.

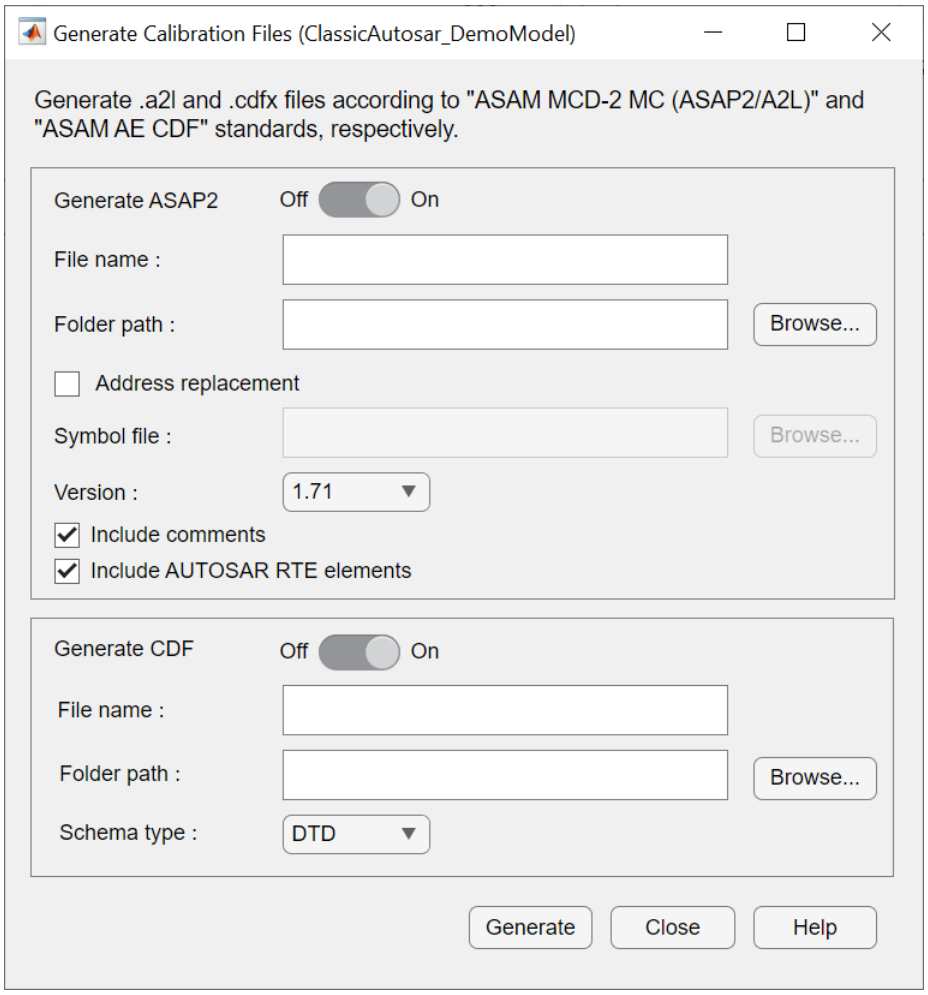
3. Generate code for the model.
4. On the **C Code** tab, select **Share** > **Generate Calibration Files**.

Note For AUTOSAR models, select **Autosar** > **Share** > **Generate Calibration Files**.

Using the tool, you can generate A2L and `cdfx` files at the same time for a model. You can use the tool for models configured with an ERT-based, GRT-based, Simulink Real-Time, Classic AUTOSAR, or Adaptive AUTOSAR system target file. The tool remembers the last used settings, such as the version of the ASAP2 file, include or exclude comments, turn off or on the ASAP2 file and CDF file generation. Simulink saves these settings in the MATLAB preferences. The settings will override the default settings.

For Adaptive AUTOSAR and Simulink Real-Time system target files, you can generate the ASAP2 file for the C++ language. For GRT- and ERT- based system target files, you can generate an ASAP2 file only for C.

For classic AUTOSAR models, the tool allows you to include or exclude RTE elements into the ASAP2 file by using the `Include AUTOSAR RTE elements` option.



ASAP2 File Generation

ASAP2 defines the description format of the internal ECU variables used for tuning parameters and measuring signal data as an application executes. To generate an ASAP2 file:

1. By default the **Generate ASAP2** option is enabled. If it is not enabled, slide the **Generate ASAP2** slider to the on position to enable A2L generation.
2. To change the name of the A2L file, in the **File name** field, specify the name.
3. By default, the code generator uses the build folder for the folder path of the selected model. If you want to specify a different folder path for the A2L file, you can do that in the **Folder path** field.
4. To replace ECU addresses in the A2L file, in the **Symbol file** field, provide a symbol file associated with the application executable. Otherwise, clear the **Address replacement** field.

   The tool supports an ELF, PDB, DWARF, or executable file as a symbol file to replace the ECU addresses.
5. Set **Version** to the required version of the A2L file to be generated. The default version is 1.71.
6. To exclude comments in the A2L file, clear **Include comments**.

7. For Classic AUTOSAR models, to exclude RTE elements in the A2L file, clear **Include AUTOSAR RTE elements**.
8. Click **Generate**.

To generate the A2L file programmatically, use the `coder.asap2.export` function.

CDF File Generation

ASAM CDF defines the data description format for storing the ECU parameter values and the associated meta data. The ASAM CDF files use XML, which can be easily validated, edited, imported, and exported by calibration tools and XML editors. To generate a CDF file by using the tool:

1. Slide the **Generate CDF** slider to the on position.
2. To change the name of the CDF file, specify it in the **File name** field.
3. By default, the code generator selects the build folder for the folder path of the selected model. If you want to specify a different folder path for the CDF file, in the **Folder path** field, specify the path.
4. Use the **Schema type** field to set to **XSD** (XML Schema Definition) or **DTD** (Document Type Definition). The **DTD** schema type is the default.

To generate the CDF file programmatically, use `coder.cdf.export` function.

Note CDF file generation is disabled for Simulink Real-Time system target file.

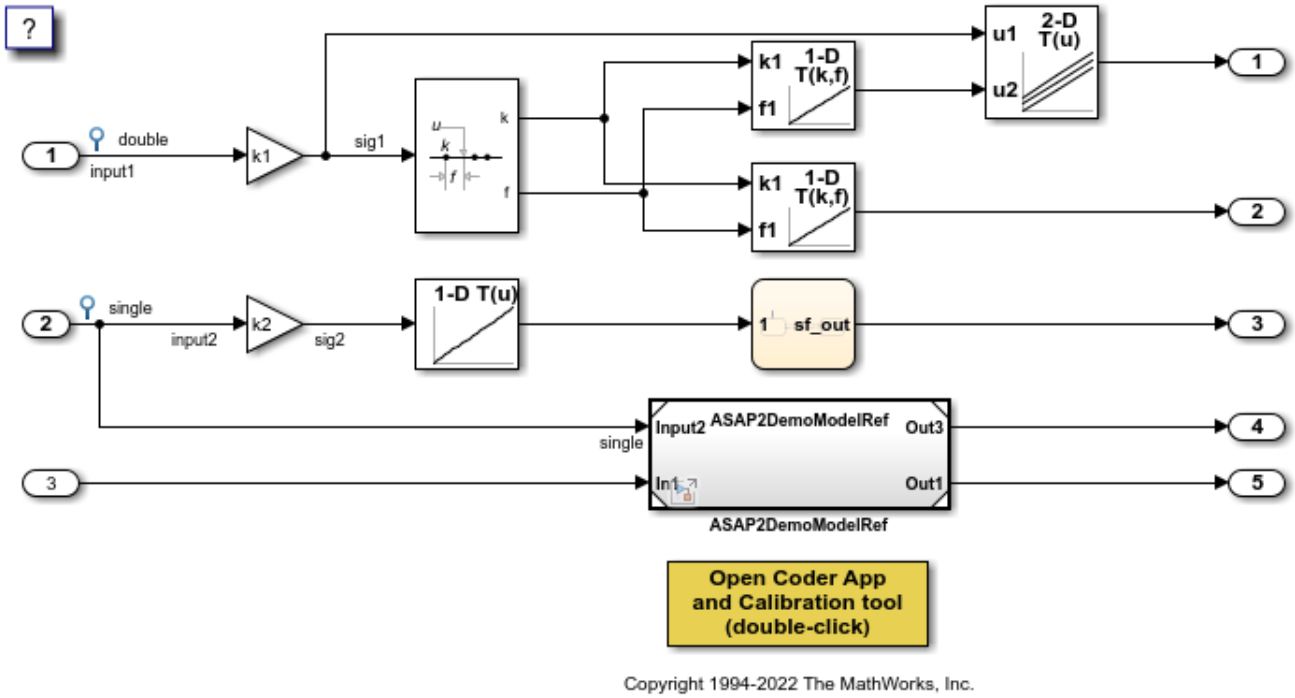
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration

This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM).

Open Example Model

Open the example model ASAP2Demo.

```
open_system('ASAP2Demo');
```



See the “ASAP2 File Generation” to customize and generate the ASAP2 file.

See Also

`coder.cdf.export` | `coder.asap2.export` | **Generate Calibration Files**

More About

- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Customize XCP Server Software” on page 60-45
- “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (AUTOSAR Blockset)

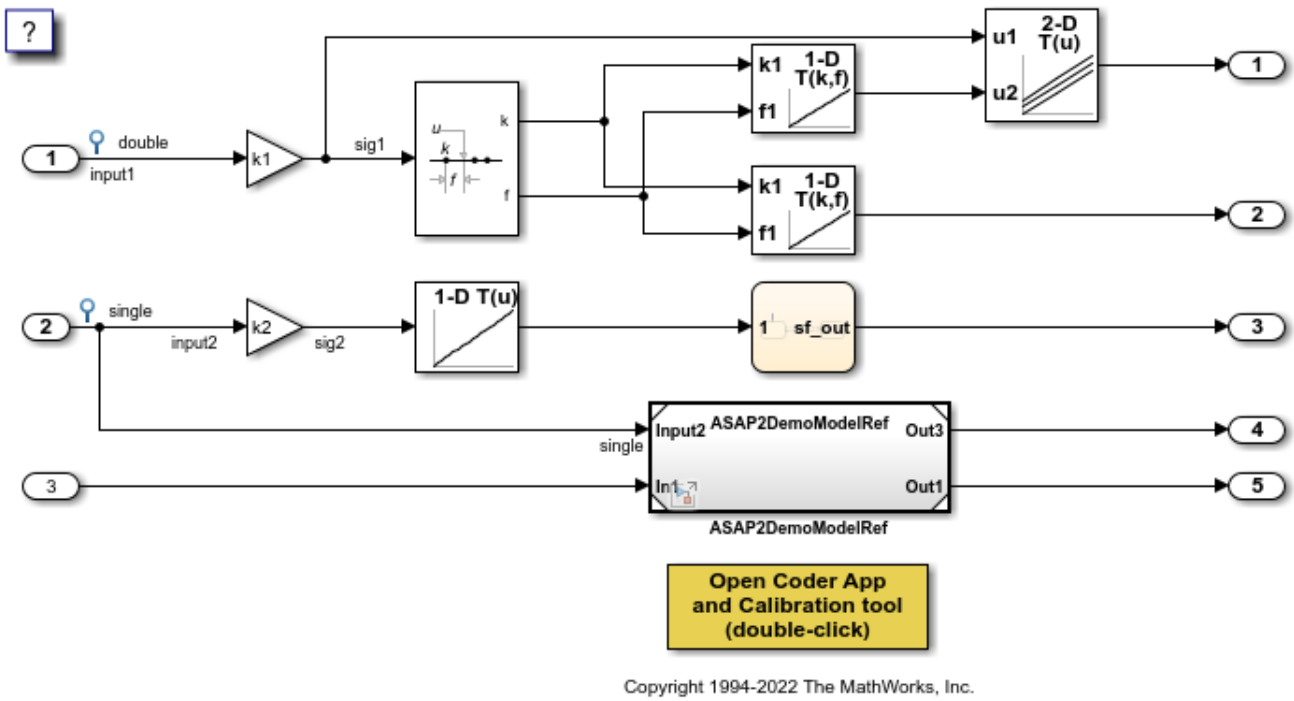
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration

This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM).

Open Example Model

Open the example model ASAP2Demo.

```
open_system('ASAP2Demo');
```



See the “ASAP2 File Generation” to customize and generate the ASAP2 file.

Customize Generated ASAP2 File

Customize Generated ASAP2 File

In this section...

"Customize Computation Method Names" on page 65-2
"Export Characteristics and Measurement Objects in Groups" on page 65-3
"Generate Record Layouts" on page 65-4
"Exclude Structures and 64-Bit Integers" on page 65-4
"Generate ASAP2 Containing Default Event List" on page 65-5
"Specify ECU Address Extension" on page 65-6
"Merge ASAP2 Files" on page 65-6

You can customize the fields of an ASAP2 file by using the code generator in any of these ways:

Customize Computation Method Names

In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model-oriented representation. Using Simulink Coder software, you can customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file readability, or names that meet organizational requirements.

To customize computation method names, use the MATLAB function `getCompuMethodName`, which is defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.

The `getCompuMethodName` function constructs a computation method name. The function prototype is:

```
cmName = getCompuMethodName(dataTypeName, cmUnits)
```

where

- *`dataTypeName`* is the name of the data type associated with the computation method.
- *`cmUnits`* are the units as specified in the `Unit` property of a `Simulink.Parameter` or `Simulink.Signal` object (for example, `rpm` or `m/s`).
- *`cmName`* returns the constructed computation method name.

The default constructed name returned by the function has the format:

```
<localPrefix><datatype>_<cmUnits>
```

where

- `<local_Prefix>` is a local prefix, `CM_`, defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.
- `<datatype>` and `<cmUnits>` are the arguments that you specified to the `getCompuMethodName` function.

Special characters in *`cmUnits`* are not supported in the computation method name in ASAP2 file. The special characters are converted as shown in this table.

Special Character	Conversion
%	_percent_
-	_dash_
/	_per_
*	_into_
°	_deg_

For example, if you call the `getCompuMethodName` function by using the *`dataTypeName`* argument `'int16'` and the *`cmUnits`* argument `'m/s'`, and then generate an ASAP2 file for a model named `myModel`, the computation method name appears in the generated file:

```
/begin COMPU_METHOD
   /* Name of CompuMethod */ myModel_CM_int16_m_per_s
   /* Units */ "m/s"
   ...
/end COMPU_METHOD
```

Export Characteristics and Measurement Objects in Groups

Starting in R2021b, you can group the characteristic and measurement objects in the ASAP2 file based on their properties. By default, the generated ASAP2 file contains `GROUPS` and `SUB_GROUPS` based on the subsystems present in the model.

1. Open the `sldemo_fuelsys` model.

   ```
   open_system("sldemo_fuelsys")
   ```

2. Use **Embedded Coder Quick Start** app by entering into code perspective of the model and clicking **C Code** > **Quick Start**.
3. Build the model.
4. Generate the ASAP2 file by using the `Generate Calibration Files` tool or by using this command:

   ```
   coder.asap2.export("sldemo_fuelsys")
   ```

   The data elements are grouped based on the model architecture. The top model is referred to as `GROUP` and subsystems in the model are referred to as `SUB_GROUPS`. The generated file contains these lines:

   ```
   /begin GROUP
      /* Name                    */      sldemo_fuelsys
      /* Long identifier         */      "sldemo_fuelsys"
      /* Root                    */      ROOT
      /begin SUB_GROUP
        rtB.es_o
        sldemo_fuelsys_Dashboard
        sldemo_fuelsys_Engine_Gas_Dynamics
        sldemo_fuelsys_Throttle_Command
        sldemo_fuelsys_To_Controller
        sldemo_fuelsys_To_Plant
        sldemo_fuelsys_fuel_rate_control
      /end SUB_GROUP
    /end GROUP
   ```

5 To group the `scalar` objects into a group named `GROUP_TYPE_SCALAR` in the ASAP2 file, use this command:

```
coder.asap2.export("sldemo_fuelsys",CustomizeGroupsBy={'SCALAR'})
```

You can also group the objects of type `array`, and lookup tables into `MAP`, `CURVE`, `CUBOID`, `CUBE_4`, and `CUBE_5` based on dimensions of the table.

Generate Record Layouts

1 Open the `sldemo_fuelsys` model.

```
open_system("sldemo_fuelsys")
```

2 Build the model.

3 Generate the ASAP2 file by using this command:

```
coder.asap2.export("sldemo_fuelsys",IncludeAllRecordLayouts=true)
```

4 The `RecordLayouts.a2l` file is generated according to the version of the ASAP2 file. In the ASAP2 file, verify this line:

```
/include "RecordLayouts.a2l"
```

5 The generated `RecordLayouts.a2l` file contains the record layouts of the data types present in the model.

Note The `RecordLayouts.a2l` file is required to be in the same folder as the ASAP2 file.

6 `coder.asap2.export("sldemo_fuelsys",IncludeAllRecordLayouts=false)` does not generate separate record layouts file, but includes the record layouts of the data types of exported data objects in the ASAP2 file.

```
/begin  RECORD_LAYOUT Scalar_FLOAT32_IEEE
    FNC_VALUES 1 FLOAT32_IEEE COLUMN_DIR DIRECT
  /end  RECORD_LAYOUT

  /begin  RECORD_LAYOUT Scalar_FLOAT64_IEEE
    FNC_VALUES 1 FLOAT64_IEEE COLUMN_DIR DIRECT
  /end  RECORD_LAYOUT
...
```

Tip In the ASA2P file, to preserve the shape of a lookup table that is used as a model argument, make sure that the LookUp Table block is in the referenced model. Pass the `Lookup table object` from the top model as an instance parameter value.

Exclude Structures and 64-Bit Integers

Starting in R2022a, you can exclude the structure elements and 64-bit integers in the generated A2L file. By default, structures and 64-bit integers are included in the A2L file.

1 Open the `sldemo_fuelsys` model.

```
open_system("sldemo_fuelsys")
```

2. Build the model.
3. Generate the ASAP2 file by using this command:

```
coder.asap2.export("sldemo_fuelsys", SupportStructureElements=false, ...
Support64bitIntegers=false)
```

   The generated ASAP2 file does not have 64-bit integers and structure elements in it.

Note Starting in R2022a, the pointer variables will not be generated in the ASAP2 file.

Generate ASAP2 Containing Default Event List

Starting in R2022a, for a multirate model, the ASAP2 file contains the `IF_DATA` section that has the `DEFAULT_EVENT_LIST`, enabling you to view the respective raster values for the data elements in the calibration tools.

1. Open the `sldemo_fuelsys` model.

```
open_system("sldemo_fuelsys")
```

2. Build the model.
3. Generate the ASAP2 file by using `Generate Calibration Files` tool or by using this command:

```
coder.asap2.export("sldemo_fuelsys")
```

4. The model has a single rate of `0.01`. The generated file defines measurements as:

```
/begin MEASUREMENT
 /* Name                   */      rtB.es_o.throttle
 /* Long identifier        */      ""
 /* Data type              */      FLOAT32_IEEE
 /* Conversion method      */      sldemo_fuelsys_CM_single
 /* Resolution (Not used)  */      0
 /* Accuracy (Not used)    */      0
 /* Lower limit            */      -3.4E+38
 /* Upper limit            */      3.4E+38
 ECU_ADDRESS                       0x0000 /* @ECU_Address@rtB.es_o.throttle@ */
/end MEASUREMENT
```

5. Convert the model into multirate model by using this command:

```
blockName = ['sldemo_fuelsys' '/EGO Fault Switch'];
set_param(blockName, 'SampleTime', '0.02');
```

6. Build the model.
7. Generate the ASAP2 file by using this command:

```
coder.asap2.export("sldemo_fuelsys")
```

8. The generated file now defines the measurements as:

```
/begin MEASUREMENT
 /* Name                   */      rtB.es_o.throttle
 /* Long identifier        */      ""
 /* Data type              */      FLOAT32_IEEE
 /* Conversion method      */      sldemo_fuelsys_CM_single
 /* Resolution (Not used)  */      0
 /* Accuracy (Not used)    */      0
 /* Lower limit            */      -3.4E+38
 /* Upper limit            */      3.4E+38
 ECU_ADDRESS                       0x0000 /* @ECU_Address@rtB.es_o.throttle@ */
 /begin IF_DATA XCP
```

```
      /begin DAQ_EVENT VARIABLE
          /begin DEFAULT_EVENT_LIST
            EVENT  0x0001
          /end DEFAULT_EVENT_LIST
      /end DAQ_EVENT
     /end IF_DATA
  /end MEASUREMENT
```

9 To generate the ASAP2 file without `DEFAULT_EVENT_LIST`, use the command:

```
coder.asap2.export("sldemo_fuelsys",IncludeDefaultEventList=false);
```

Specify ECU Address Extension

Starting in R2022a, you can specify additional address information in the ASAP2 file by specifying an integer value for the `EcuAddressExtension` property. By default, no additional address information is provided.

1 Open the `sldemo_fuelsys` model.

```
open_system("sldemo_fuelsys")
```

2 Build the model.

3 Generate the ASAP2 file by using this command:

```
coder.asap2.export("sldemo_fuelsys",EcuAddressExtension=4)
```

The `ECU_ADDRESS_EXTENSION` key word for the characteristics, measurements, and axis points are in the ASAP2 file, enabling you to specify additional address information.

```
/begin MEASUREMENT
  /* Name                   */      rtB.MinMax
  /* Long identifier        */      ""
  /* Data type              */      FLOAT64_IEEE
  /* Conversion method      */      sldemo_fuelsys_CM_double
  /* Resolution (Not used)  */      0
  /* Accuracy (Not used)    */      0
  /* Lower limit            */      -1.7E+308
  /* Upper limit            */      1.7E+308
  ECU_ADDRESS                       0x0000 /* @ECU_Address@rtB.MinMax@ */
  ECU_ADDRESS_EXTENSION 4
/end MEASUREMENT
```

Merge ASAP2 Files

Starting in R2023a, you can merge the data elements of multiple A2L files to a Simulink model. Use the `coder.asap2.merge` function to create the merged descriptor object and generate an A2L file containing the data elements of the model and merged A2L files.

Vehicle Network Toolbox is required to merge A2L files using the `coder.asap2.merge` function.

1 Open the `sldemo_fuelsys` model.

```
open_system("sldemo_fuelsys")
```

2 Build the model.

3 Merge the data elements of an A2L file to the model using the command:

```
mergeObj = coder.asap2.merge("sldemo_fuelsys",{"ASAP2File1.a2l"})
```

To merge more than one A2L file, specify the files in comma separated format. For example:

```
mergeObj = coder.asap2.merge("sldemo_fuelsys",{"ASAP2File1.a2l","ASAP2File2.a2l"})
```

4 Generate the A2L file using the `mergeObj`.

```
coder.asap2.export("sldemo_fuelsys",CustomEcuDescriptions=mergeObj)
```

The generated A2L file contains the data elements present in `sldemo_fuelsys` model and the `ASAP2File1` A2L file.

Note Lookup table objects will not be merged.

See Also

`coder.asap2.export`

More About

- "Generate ASAP2 and CDF Calibration Files" on page 64-2
- "External Mode Simulation by Using XCP Communication" on page 60-8

Create and Add Custom Data Elements

Generate an ASAP2 file by adding custom characteristics, measurements, axis-points, functions, record layouts, and compu methods. By default, an ASAP2 file contains the data elements present in model that were configured to export.

Using the Simulink Coder you can:

- Create and add custom data elements such as characteristics, measurements, axis-points, functions, record layouts, and compu methods.
- Remove data elements from the ASAP2 file.
- Filter and filter data elements based on their properties.
- Get or update the properties of data elements.

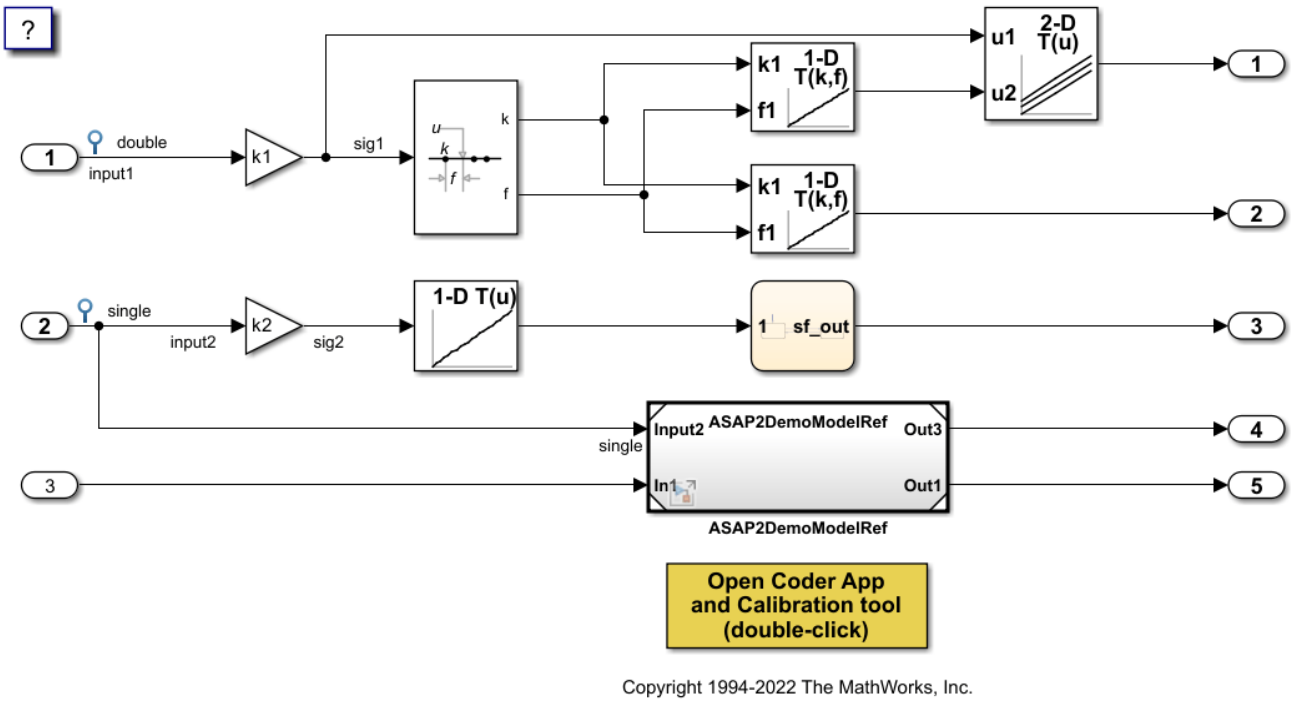
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Build the model.

```
rtwbuild("ASAP2Demo");
```

```
### Starting serial model reference code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo

Build Summary

Code generation targets built:

Model               Action                        Rebuild Reason
========================================================================================
ASAP2DemoModelRef   Code generated and compiled.  ASAP2DemoModelRef.c does not exist.

Top model targets built:

Model       Action                          Rebuild Reason
```

```
==========================================================================================
ASAP2Demo  Code generated and compiled.  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 30.188s
```

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")
```

```
ans = 1x10 string
    "ASAP2DemoModelRef_CM_double"    "ASAP2DemoModelRef_CM_int16_rpm"    "ASAP2Demo_CM_double"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')
```

```
ans = 1x3 string
    "ASAP2DemoModelRef_CM_int16_rpm"    "ASAP2Demo_CM_double_rpm"    "ASAP2Demo_CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_CM1 = coder.asap2.CompuMethod;
CompuMethod_CM1.Name = 'CompuMethod_1';
CompuMethod_CM1.ConversionType = 'LINEAR';
CompuMethod_CM1.Coefficients = [2 3];
CompuMethod_CM1.LongIdentifier = 'longIdentifierTest';
CompuMethod_CM1.Format = '%2.3';
CompuMethod_CM1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_CM1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans = 
  CompuMethod with properties:

               Name: 'CompuMethod_1'
     LongIdentifier: 'longIdentifierTest'
             Format: '%2.3'
              Units: 's'
       Coefficients: [2 3]
     ConversionType: 'LINEAR'
    CompuVTabValues: [1x1 struct]
         CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *LongIdentifier* field of the computation method.

```
set(descObj,"CompuMethod","CompuMethod_1",LongIdentifier="longIdentifierCM1")
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans = 
  Characteristic with properties:

                   Name: 'Custom_parameter1'
         LongIdentifier: 'longIdentifierParam'
                   Type: 'VALUE'
             EcuAddress: '0x0000'
        CompuMethodName: 'NO_COMPU_METHOD'
             LowerLimit: 0
             UpperLimit: 255
      EcuAddressComment: ""
    EcuAddressExtension: []
      CalibrationAccess: 'Calibration'
      DisplayIdentifier: ""
                 Format: ""
                BitMask: []
               AxisInfo: []
           RecordLayout: ""
             Dimensions: []
                 Export: 1
             MaxRefresh: [1x1 struct]
             SymbolLink: [1x1 struct]
```

```
            CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj,"Characteristic","Custom_parameter1",UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Remove the newly added characteristic from the description object.

```
delete(descObj,"Characteristic","Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj,Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj,"Measurement","Custom_signal1")
```

```
ans = 
  Measurement with properties:

                   Name: 'Custom_signal1'
         LongIdentifier: 'longIdentifierSignal'
               DataType: 'UBYTE'
             EcuAddress: '0x0000'
        CompuMethodName: ""
             LowerLimit: 0
             UpperLimit: 255
                 Raster: [1x1 struct]
      EcuAddressComment: ""
    EcuAddressExtension: []
      CalibrationAccess: 'NoCalibration'
      DisplayIdentifier: ""
                 Format: ""
                BitMask: []
             Dimensions: []
                 Export: 1
               MaskData: [1x1 struct]
             MaxRefresh: [1x1 struct]
             SymbolLink: [1x1 struct]
```

```
            CustomData: ""
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj,"Measurement","Custom_signal1",CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Remove the newly added measurement from the description object.

```
delete(descObj,"Measurement","Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj,LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

See Also

`coder.asap2.export` | `coder.asap2.getEcuDescriptions` | `coder.asap2.Characteristic` | `coder.asap2.Measurement` | `coder.asap2.CompuMethod` | `coder.asap2.Function` | `coder.asap2.RecordLayout`

Related Examples

- “Generate ASAP2 and CDF Calibration Files” on page 64-2

Customize Using Base Files

You can add custom A2L file fragments such as PROJECT, MODULE, MOD, MOD_PAR, and MOD_COMMON settings to the ASAP2 file. You can do it in the following ways:

Customize ASAP2 Fields by Using Base Class

1. Create an object from the base class using the `coder.asap2.UserCustomizeBase` function.

   ```
   obj = coder.asap2.UserCustomizeBase;
   ```

2. Define the object fields.

   ```
   obj.HeaderComment = 'Header comment';
   obj.ModParComment = 'Mod Par comment';
   obj.ModCommonComment = 'Mod Common comment';
   obj.ASAP2FileName = 'File_name';
   obj.ByteOrder = 'BYTEORDER MSB_LAST';
   ```

3. Pass the object to the function and generate the ASAP2 file.

   ```
   coder.asap2.export(modelName,CustomizationObject=obj);
   ```

Customize ASAP2 Sections by Deriving from Base Class

1. Derive from the base class `coder.asap2.UserCustomizeBase`. You can place the derived file anywhere on the MATLAB path.
2. To edit the Header section, use the `writeHeader` function.
3. Use the `writeHardwareInterface` function to edit the hardware interface section consisting of the MOD_PAR and MOD_COMMON fields of the ASAP2 file.
4. To add text at the beginning of the ASAP2 file, use the `writeFileHead` function.
5. To add text at the end of the ASAP2 file, use the `writeFileTail` function.

   For example, the code for generating a derived class `TestASAP2Customization` might look like this code:

   ```
   classdef TestASAP2Customization < coder.asap2.UserCustomizeBase
       %   Customization class for asap2 file
       methods
           function header = writeHeader(obj)
               header = sprintf(['  /begin HEADER ' 'test Header text'  '\n',...
                   'header test comments from user\n',...
                   '  /end HEADER']);
           end

           function hardwareInterface = writeHardwareInterface(obj)
               hardwareInterface = sprintf(['    /begin MOD_PAR "' 'UserDefined' '"\n',...
                   '      test comments from user\n',...
                   '    /end MOD_PAR\n\n',...
                   '    /begin MOD_COMMON "'  'UserDefined MOD_COMMON' '"\n',...
                   '      user defined values for MOD_COMMON\n',...
                   '     ' obj.ByteOrder '\n',...
                   '     /end MOD_COMMON']);
           end

           function fileTail = writeFileTail(obj)
               fileTail = sprintf(['/* This File can be used for ' ,...
                   'calibration .*/\n'...
                   '/* EOF ',obj.ASAP2FileName, '*/']);
           end
   ```

```
        function fileHeader = writeFileHead(obj)
            fileHeader = sprintf(['/*****************************************************************************\n
                ' This is a a2l file which can be used for calibration with INCA or CANAPE\n',...
                ' ****************************************************************************/'
                ]);
        end
    end
end
```

6 Create an object from the derived class. For example, this command creates an object from `TestASAP2Customization`.

```
obj = TestASAP2Customization;
```

7 Pass the object to the function that generates the ASAP2 file.

```
coder.asap2.export(modelName,CustomizationObject=obj);
```

See Also

`coder.asap2.export` | `coder.asap2.UserCustomizeBase`

More About

- "Generate ASAP2 and CDF Calibration Files" on page 64-2
- "External Mode Simulation by Using XCP Communication" on page 60-8

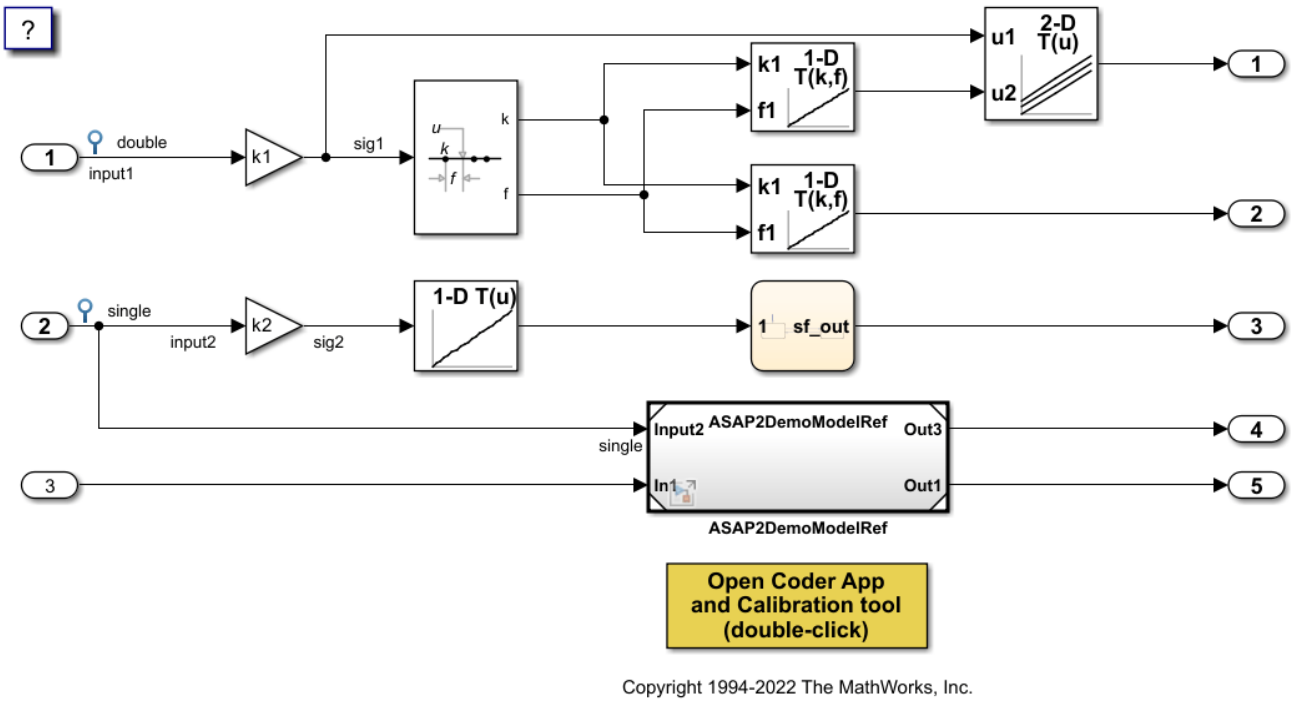
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Build the model.

```
rtwbuild("ASAP2Demo");
```

```
### Starting serial model reference code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo

Build Summary

Code generation targets built:

Model               Action                        Rebuild Reason
========================================================================================
ASAP2DemoModelRef   Code generated and compiled.  ASAP2DemoModelRef.c does not exist.

Top model targets built:

Model       Action                          Rebuild Reason
```

```
==========================================================================================
ASAP2Demo  Code generated and compiled.  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 30.188s
```

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")
```

```
ans = 1x10 string
    "ASAP2DemoModelRef_CM_double"    "ASAP2DemoModelRef_CM_int16_rpm"    "ASAP2Demo_CM_double"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')
```

```
ans = 1x3 string
    "ASAP2DemoModelRef_CM_int16_rpm"    "ASAP2Demo_CM_double_rpm"    "ASAP2Demo_CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_CM1 = coder.asap2.CompuMethod;
CompuMethod_CM1.Name = 'CompuMethod_1';
CompuMethod_CM1.ConversionType = 'LINEAR';
CompuMethod_CM1.Coefficients = [2 3];
CompuMethod_CM1.LongIdentifier = 'longIdentifierTest';
CompuMethod_CM1.Format = '%2.3';
CompuMethod_CM1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_CM1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans = 
  CompuMethod with properties:

               Name: 'CompuMethod_1'
     LongIdentifier: 'longIdentifierTest'
             Format: '%2.3'
              Units: 's'
       Coefficients: [2 3]
     ConversionType: 'LINEAR'
    CompuVTabValues: [1x1 struct]
         CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *LongIdentifier* field of the computation method.

```
set(descObj,"CompuMethod","CompuMethod_1",LongIdentifier="longIdentifierCM1")
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans = 
  Characteristic with properties:

                   Name: 'Custom_parameter1'
         LongIdentifier: 'longIdentifierParam'
                   Type: 'VALUE'
             EcuAddress: '0x0000'
        CompuMethodName: 'NO_COMPU_METHOD'
             LowerLimit: 0
             UpperLimit: 255
      EcuAddressComment: ""
    EcuAddressExtension: []
      CalibrationAccess: 'Calibration'
      DisplayIdentifier: ""
                 Format: ""
                BitMask: []
               AxisInfo: []
           RecordLayout: ""
             Dimensions: []
                 Export: 1
             MaxRefresh: [1x1 struct]
             SymbolLink: [1x1 struct]
```

```
            CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj,"Characteristic","Custom_parameter1",UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Remove the newly added characteristic from the description object.

```
delete(descObj,"Characteristic","Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj,Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj,"Measurement","Custom_signal1")
```

```
ans = 
  Measurement with properties:

                   Name: 'Custom_signal1'
         LongIdentifier: 'longIdentifierSignal'
               DataType: 'UBYTE'
             EcuAddress: '0x0000'
        CompuMethodName: ""
             LowerLimit: 0
             UpperLimit: 255
                 Raster: [1x1 struct]
      EcuAddressComment: ""
    EcuAddressExtension: []
      CalibrationAccess: 'NoCalibration'
      DisplayIdentifier: ""
                 Format: ""
                BitMask: []
             Dimensions: []
                 Export: 1
               MaskData: [1x1 struct]
             MaxRefresh: [1x1 struct]
             SymbolLink: [1x1 struct]
```

```
            CustomData: ""
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj,"Measurement","Custom_signal1",CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Remove the newly added measurement from the description object.

```
delete(descObj,"Measurement","Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj,LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

```
Following Characteristics or Measurements with unsupported data types are not exported in ASAP2 f
```

Configure Model Data Elements for ASAP2 File Generation

Configure Model Data Elements for ASAP2 File Generation

When you generate an ASAP2 file from a model, the ASAP2 file includes information about the data elements in your model such as characteristics and measurement variables. Each data element is described by information such as the memory address, storage structure, data types and conversion rules. In Simulink you can now customize how the data elements are described in an ASAP2 file. You can choose only those characteristics and measurements that you are interested in exporting into ASAP2 file.

To configure data elements for ASAP2 file generation in the **Code Mappings Editor – C** display, use the tabs:

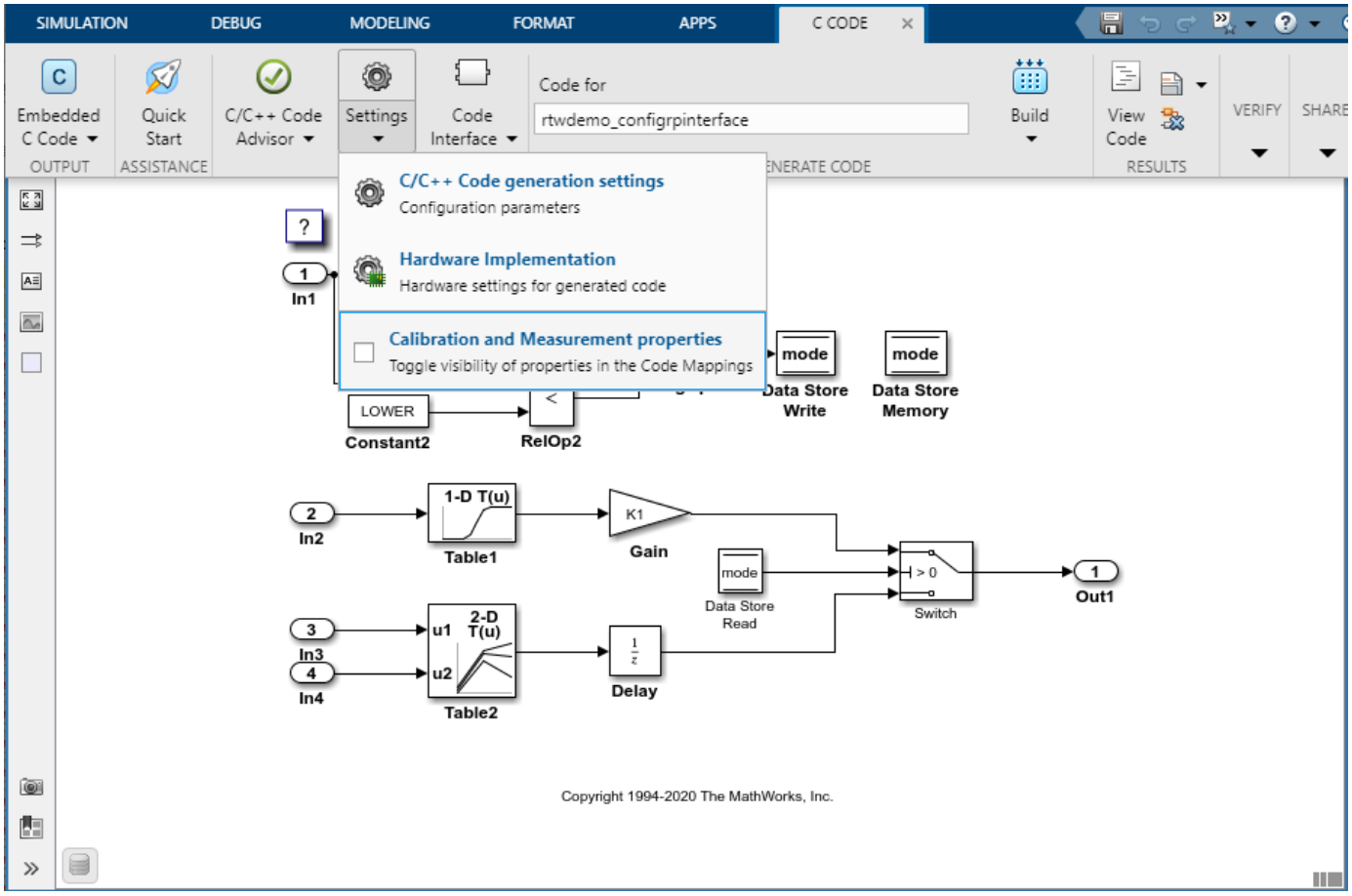
- Inports
- Outports
- Parameters
- Data Stores
- Signals/States

View Calibration Attributes

1. Open a model with multiple data elements.

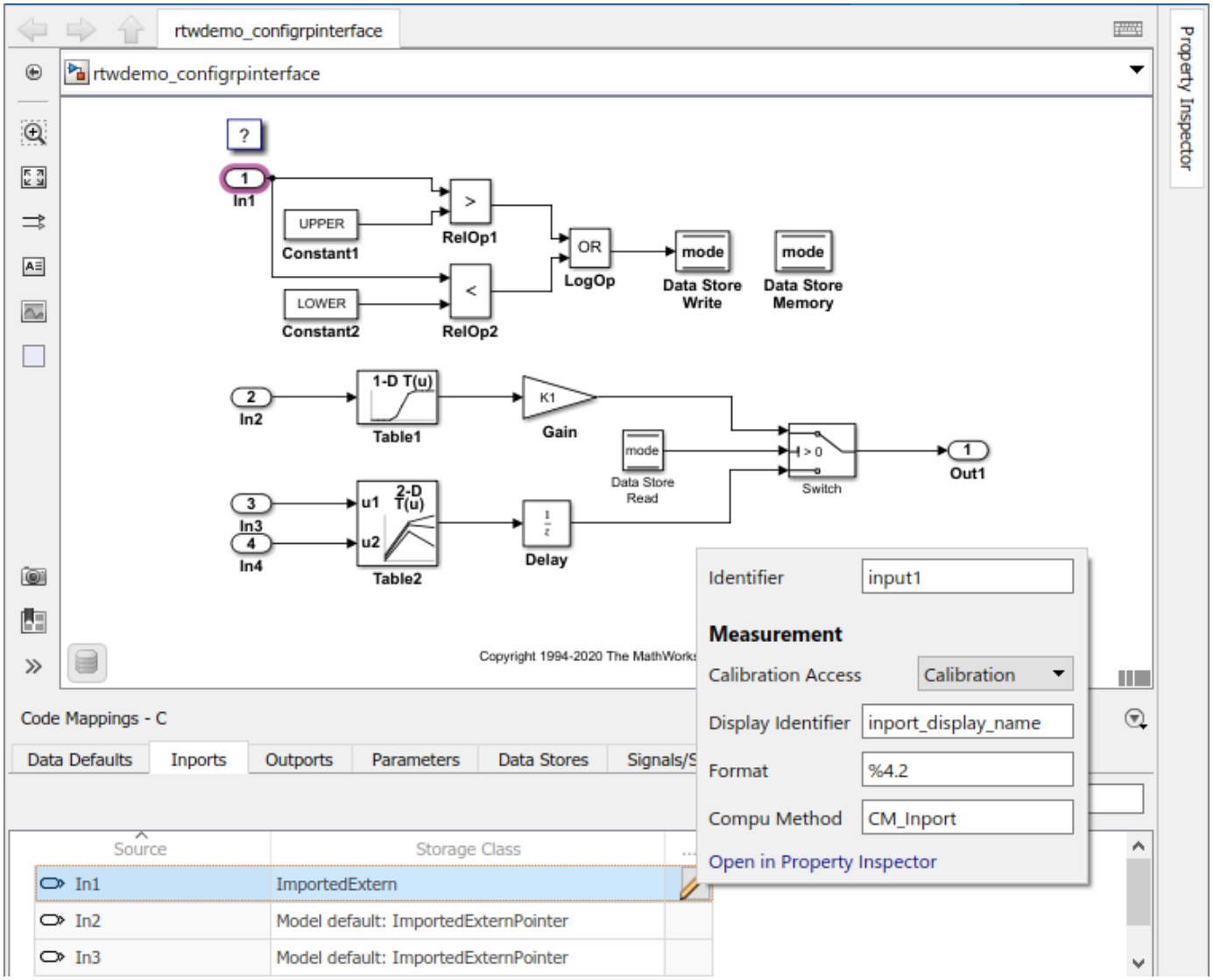
   ```
   open_system('rtwdemo_configrpinterface')
   ```

2. Open the Embedded Coder app. On the **C Code** tab, select **Code Interface** + **Individual Element Code Mappings**.
3. On the **C Code** tab, select **Settings** + **Calibration and Measurement properties**.



Configure Inport for ASAP2 File Generation

1. On the **Inports** tab, select an inport and click the icon.
2. Specify the measurement property fields for the inport.



3 Alternatively, at the command line, you can enter these commands:

```
setInport(cm,'In1','CalibrationAccess','Calibration');
setInport(cm,'In1','CompuMethod','CM_Inport');
setInport(cm,'In1','DisplayIdentifier','inport_display_name');
setInport(cm,'In1','Format','%4.2');
```

4 Build the model.

5 Generate the ASAP2 file by using this command:

```
coder.asap2.export('rtwdemo_configrpinterface');
```

6 Verify the exported inport in the ASAP2 file and verify the fields that you specified are present.

```
/begin MEASUREMENT
  /* Name                    */      input1
  /* Long identifier         */      ""
  /* Data type               */      FLOAT64_IEEE
  /* Conversion method       */      CM_Inport
  /* Resolution (Not used)   */      0
  /* Accuracy (Not used)     */      0
  /* Lower limit             */      -20
  /* Upper limit             */      20
  ECU_ADDRESS                        0x0000 /* @ECU_Address@input1@ */
  DISPLAY_IDENTIFIER inport_display_name
  FORMAT "%4.2"
  READ_WRITE
/end MEASUREMENT
```

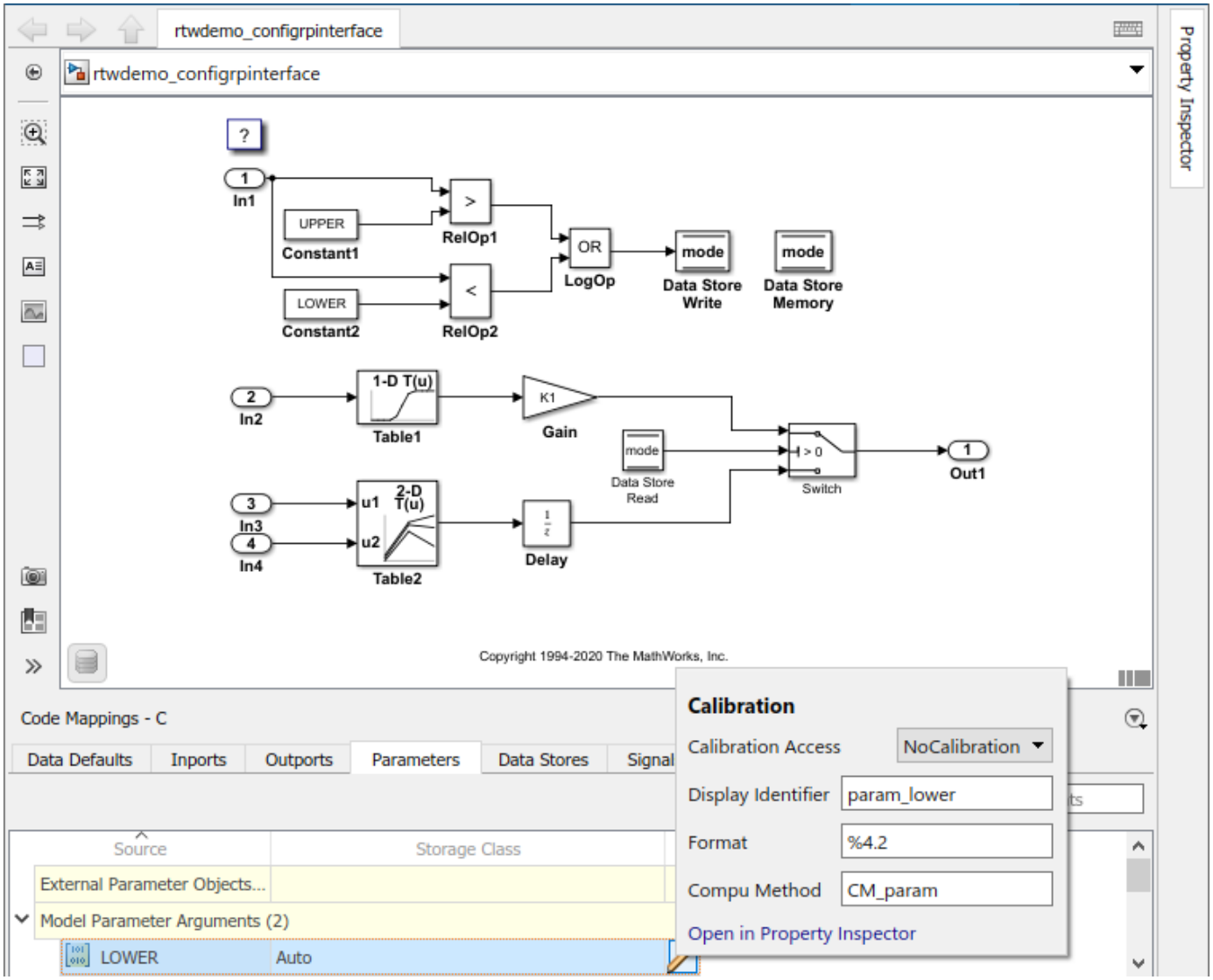
You can configure an outport in the same way. By default, all the inports and outports are exported to A2L file. To avoid exporting an inport or an outport to A2L file, set the `Export` property to false using the command:

```
setInport(cm,'In1','Export',false);
```

Configure Parameter for ASAP2 File Generation

1 On the **Parameters** tab, select a parameter and click the icon.

2 Specify the calibration property fields for the parameter.



3 Alternatively, at the command line, you can enter these commands:

```
setModelParameter(cm,'LOWER','CalibrationAccess','NoCalibration');
setModelParameter(cm,'LOWER','CompuMethod','CM_param');
setModelParameter(cm,'LOWER','DisplayIdentifier','param_lower');
setModelParameter(cm,'LOWER','Format','%4.2');
```

4 Build the model.

5 Generate the ASAP2 file by using this command:

```
coder.asap2.export('rtwdemo_configrpinterface');
```

6 Verify the exported parameter in the ASAP2 file and verify the fields that you specified are present.

```
/begin CHARACTERISTIC
  /* Name                    */      rtwdemo_configrpinterface_InstP.LOWER
  /* Long Identifier         */      ""
  /* Type                    */      VALUE
  /* ECU Address             */      0x0000 /* @ECU_Address@rtwdemo_configrpinterface_InstP.LOWER@ */
  /* Record Layout           */      Scalar_FLOAT64_IEEE
  /* Maximum Difference      */      0
  /* Conversion Method       */      CM_param
  /* Lower Limit             */      -20.000000
  /* Upper Limit             */      0.000000
  CALIBRATION_ACCESS NO_CALIBRATION
  DISPLAY_IDENTIFIER param_lower
  FORMAT "%4.2"
/end CHARACTERISTIC
```

You can configure data stores, states in the same way. By default, signals, states, and data stores are exported to A2L file. To avoid exporting a data element to A2L file, set the `Export` property to false using the command:

```
setModelParameter(cm,'LOWER','Export',false);
```

Note To export signals, states, data stores, and parameters to the A2L file, configure the data element to a non-auto storage class.

See Also

`setSignal` | `setInport` | `setOutport` | `setModelParameter` | `setDataStore` | `setState` | `coder.asap2.export` | `coder.mapping.api.CodeMapping` | **Code Mappings Editor – C**

More About

- “Generate ASAP2 and CDF Calibration Files” on page 64-2
- “Customize Generated ASAP2 File” on page 65-2
- “C Code Generation Configuration for Model Interface Elements” on page 33-2
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Control Data and Function Interface in Generated Code” on page 36-44

Direct Memory Access to Generated Code for Simulink Coder

Access Memory in Generated Code Using Global Data Map

Simulink Coder provides a Target Language Compiler (TLC) function library that lets you create a global data map record. The global data map record, when generated, is added to the `CompiledModel` structure in the `model.rtw` file. The global data map record is a database containing information required for accessing memory in the generated code, including

- Signals (Block I/O)
- Parameters
- Data type work vectors (DWork)
- External inputs
- External outputs

Use of the global data map requires knowledge of the Target Language Compiler and of the structure of the `model.rtw` file. See "Target Language Compiler Overview" for information on these topics.

The TLC functions that are required to generate and access the global data map record are contained in *`matlabroot`*`/rtw/c/tlc/mw/globalmaplib.tlc`. The comments in the source code fully document the global data map structures and the library functions.

The global data map structures and functions might be modified or enhanced in future releases.

See Also

Related Examples

- "How Generated Code Exchanges Data with an Environment"
- "How Generated Code Stores Internal Signal, State, and Parameter Data"

Desktops in Simulink Coder

- “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 68-2
- “Run Rapid Simulations over a Range of Parameter Values” on page 68-24
- “Run Batch Simulations Without Recompiling Generated Code” on page 68-30
- “Use MAT-Files to Feed Data to Inport Blocks for Rapid Simulations” on page 68-35
- “Use S-Function Target for Model or Subsystem” on page 68-40

Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File

After you create a model, you can use the rapid simulation (RSim) system target file to characterize model behavior. The executable program that results from the build process is for non-real-time execution on your development computer. The executable program is highly optimized for simulating models of hybrid dynamic systems, including models that use variable-step solvers and zero-crossing detection. The speed of the generated code makes the RSim system target file ideal for building programs for batch or Monte Carlo simulations.

About Rapid Simulation

Use the RSim target to generate an executable that runs fast, standalone simulations. You can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model. This can accelerate the characterization and tuning of model behavior and code generation testing.

Using command-line options:

- Define parameter values and input signals in one or more MAT-files that you can load and reload at the start of simulations without rebuilding your model.
- Redirect logging data to one or more MAT-files that you can then analyze and compare.
- Control simulation time.
- Specify external mode options.

Note To run an RSim executable, configure your computer to run MATLAB and have the MATLAB and Simulink installation folders accessible. To deploy a standalone host executable (that is, without MATLAB and Simulink installed), consider using the Host-Based Shared Library target (ert_shrlib).

Rapid Simulation Advantage

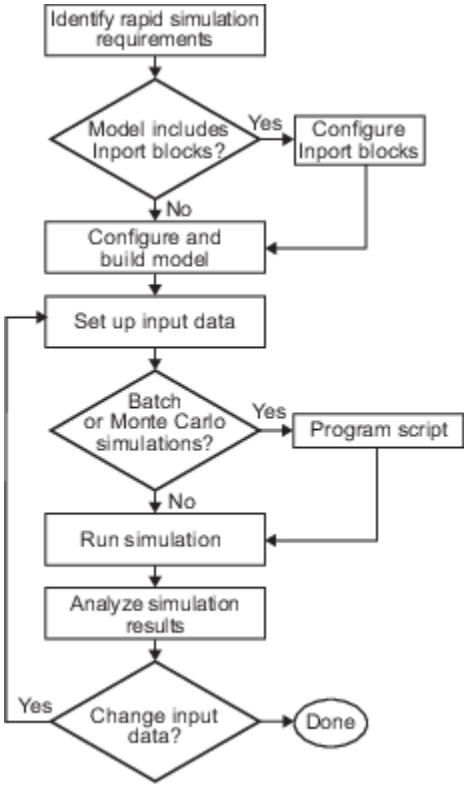
The advantage that you gain from rapid simulation varies. Larger simulations achieve speed improvements of up to 10 times faster than standard Simulink simulations. Some models might not show noticeable improvement in simulation speed. To determine the speed difference for your model, time your standard Simulink simulation and compare the results with a rapid simulation. In addition, test the model simulation in Rapid Accelerator simulation mode.

General Rapid Simulation Workflow

Like other stages of Model-Based Design, characterization and tuning of model behavior is an iterative process, as shown in the general workflow diagram in the figure. Tasks in the workflow are:

1. Identify your rapid simulation requirements. on page 68-3
2. Configure Inport blocks on page 68-4 that provide input source data for rapid simulations.
3. Configure the model on page 68-4 for rapid simulation.
4. Set up simulation input data. on page 68-6

5 Run the rapid simulations. on page 68-13



Identify Rapid Simulation Requirements

The first step to setting up a rapid simulation is to identify your simulation requirements.

Question...	For More Information, See...
How long do you want simulations to run?	“Configure and Build Model for Rapid Simulation” on page 68-4
Are there solver requirements? Do you expect to use the same solver for which the model is configured for your rapid simulations?	“Configure and Build Model for Rapid Simulation” on page 68-4
Do your rapid simulations need to accommodate flexible custom code interfacing? Or, do your simulations need to retain storage class settings?	“Configure and Build Model for Rapid Simulation” on page 68-4
Will you be running simulations with multiple data sets?	“Set Up Rapid Simulation Input Data” on page 68-6
Will the input data consist of global parameters, signals, or both?	“Set Up Rapid Simulation Input Data” on page 68-6
What type of source blocks provide input data to the model — From File, Inport, From Workspace?	“Set Up Rapid Simulation Input Data” on page 68-6

Question...	For More Information, See...
Will the model's parameter vector (*model*_P) be used as input data?	“Create a MAT-File That Includes a Model Parameter Structure” on page 68-6
What is the data type of the input parameters and signals?	“Set Up Rapid Simulation Input Data” on page 68-6
Will the source data consist of one variable or multiple variables?	“Set Up Rapid Simulation Input Data” on page 68-6
Does the input data include tunable parameters?	“Create a MAT-File That Includes a Model Parameter Structure” on page 68-6
Do you need to gain access to tunable parameter information — model checksum and parameter data types, identifiers, and complexity?	“Create a MAT-File That Includes a Model Parameter Structure” on page 68-6
Will you have a need to vary the simulation stop time for simulation runs?	“Configure and Build Model for Rapid Simulation” on page 68-4 and“Override a Model Simulation Stop Time” on page 68-15
Do you want to set a time limit for the simulation? Consider setting a time limit if your model experiences frequent zero crossings and has a small minor step size.	“Set a Clock Time Limit for a Rapid Simulation” on page 68-15
Do you need to preserve the output of each simulation run?	“Specify a New Output File Name for a Simulation” on page 68-21 and “Specify New Output File Names for To File Blocks” on page 68-21
Do you expect to run the simulations interactively or in batch mode?	“Scripts for Batch and Monte Carlo Simulations” on page 68-13

Configure Inports to Provide Simulation Source Data

You can use Inport blocks as a source of input data for rapid simulations. To do so, configure the blocks so that they can import data from external MAT-files. By default, the Inport block inherits parameter settings from downstream blocks. In most cases, to import data from an external MAT-file, you must explicitly set these block parameters to match the source data in the MAT-file.

- **Main > Interpolate data**
- **Signal Attributes > Port dimensions**
- **Signal Attributes > Data type**
- **Signal Attributes > Signal type**

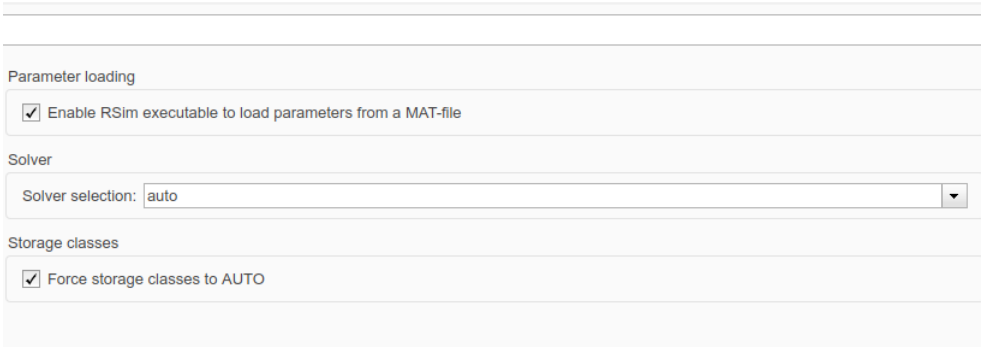
If you do not have control over the model content, you might need to modify the data in the MAT-file to conform to what the model expects for input. Input data characteristics and specifications of the Inport block that receives the data must match.

For details on adjusting these parameters and on creating a MAT-file for use with an Inport block, see “Create a MAT-File for an Inport Block” on page 68-10. For descriptions of the preceding block parameters, see the block description of Inport.

Configure and Build Model for Rapid Simulation

After you identify your rapid simulation requirements, configure the model for rapid simulation.

1. Set model configuration parameter **System target file** to `rsim.tlc` (Rapid Simulation Target).
2. Click **RSim Target** to view the **RSim Target** pane.



3. Set the RSim target configuration parameters for your rapid simulation requirements.

If You Want to...	Then...
Generate code that allows the RSim executable to load parameters from a MAT-file	Select **Enable RSim executable to load parameters from a MAT-file** (default).
Let the target choose a solver based on the solver already configured for the model	Set **Solver selection** to `auto` (default). The code generator uses a built-in solver if a fixed-step solver is specified on the **Solver** pane or calls the Simulink solver module (a shared library) if a variable-step solver is specified.
Explicitly instruct the target to use a fixed-step solver	Set **Solver selection** to `Use fixed-step solvers`. In the Configuration Parameters dialog box, on the **Solver** pane, specify a fixed-step solver.
Explicitly instruct the target to use a variable-step solver	Set **Solver selection** to `Use Simulink solver module`. In the Configuration Parameters dialog box, on the **Solver** pane, specify a variable-step solver.
Force storage classes to `Auto` for flexible custom code interfacing	Select **Force storage classes to AUTO** (default).
Retain storage class settings, such as `ExportedGlobal` or `ImportedExtern`, due to application requirements	Clear **Force storage classes to AUTO**.

4. Set up data import and export options. On the **Data Import/Export** pane, in the **Save to Workspace** section, select the **Time**, **States**, **Outputs**, and **Final States** parameters, as they apply. By default, the code generator saves simulation logging results to a file named *`model`*`.mat`. For more information, see “Save Simulation Data”.
5. If you are using external mode communication, set up the interface, using the **Code Generation** > **Interface** pane. See “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” for details.
6. Press **Ctrl+B**. The code generator builds a highly optimized executable program that you can run on your development computer with varying data, without rebuilding.

For more information on compilers that are compatible with the Simulink Coder product, see “Select and Configure C or C++ Compiler” on page 59-3 and “Template Makefiles and Make Options” on page 59-36 .

Set Up Rapid Simulation Input Data

- “About Rapid Simulation Data Setup” on page 68-6
- “Create a MAT-File That Includes a Model Parameter Structure” on page 68-6
- “Create a MAT-File for a From File Block” on page 68-9
- “Create a MAT-File for an Inport Block” on page 68-10

About Rapid Simulation Data Setup

The format and setup of input data for a rapid simulation depends on your requirements.

If the Input Data Source Is...	Then...
The model's global parameter vector (*model*_P)	Use the `rsimgetrtp` function to get the vector content and then save it to a MAT-file.
The model's global parameter vector and you want a mapping between the vector and tunable parameters	Call the `rsimgetrtp` function to get the global parameter structure and then save it to a MAT-file.
Provided by a From File block	Create a MAT-file that a From File block can read.
Provided by an Inport block	Create a MAT-file that adheres to one of the three data file formats that the Inport block can read.
Provided by a From Workspace block	Create structure variables in the MATLAB workspace.

The RSim target requires that MAT-files used as input for From File and Inport blocks contain data. The `grt` target inserts MAT-file data directly into the generated code, which is then compiled and linked as an executable. In contrast, RSim allows you to replace data sets for each successive simulation. A MAT-file containing From File or Inport block data must be present if a From File block or Inport block exists in your model.

Create a MAT-File That Includes a Model Parameter Structure

To create a MAT-file that includes a model global parameter structure (*model*_P),

1. Get the structure by calling the function `rsimgetrtp`.
2. Save the parameter structure to a MAT-file.

If you want to run simulations over varying data sets, consider converting the parameter structure to a cell array and saving the parameter variations to a single MAT-file.

Get the Parameter Structure for a Model

Get the global parameter structure (*model*_P) for a model by calling the function `rsimgetrtp`.

```
param_struct = rsimgetrtp('model')
```

Argument	Description
model	The model for which you are running the rapid simulations.

The `rsimgetrtp` function forces an update diagram action for the specified model and returns a structure that contains the following fields.

Field	Description
`modelChecksum`	A four-element vector that encodes the structure of the model. The code generator uses the checksum to check whether the structure of the model has changed since the RSim executable was generated. If you delete or add a block, and then generate a new *model*`_P` vector, the new checksum does not match the original checksum anymore. The RSim executable detects this incompatibility in parameter vectors and exits to avoid returning incorrect simulation results. If the model structure changes, you must regenerate the code for the model.
`parameters`	A structure that contains the model's global parameters.

The parameter structure contains the following information.

Field	Description
`dataTypeName`	The name of the parameter data type, for example, `double`
`dataTypeID`	Internal data type identifier used by the code generator
`complex`	The value 0 if real; 1 if complex
dtTransIdx	Internal data index used by the code generator
values	A vector of the parameter values associated with this structure
map	This field contains the mapping information that correlates the 'values' to the tunable parameters of the model. This mapping information, in conjunction with `rsimsetrtpparam`, is useful for creating subsequent `rtP` structures without compiling the block diagram.

The code generator reports a tunable fixed-point parameter according to its stored value. For example, an `sfix(16)` parameter value of `1.4` with a scaling of `2^-8` has a value of `358` as an `int16`.

In the following example, `rsimgetrtp` returns the parameter structure for the example model `rtwdemo_rsimtf` to `param_struct`.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf')
```

```
param_struct =

    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009 2.3064e+009]
       parameters: [1x1 struct]
```

Save the Parameter Structure to a MAT-File

After you issue a call to `rsimgetrtp`, save the return value of the function call to a MAT-file. Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

The following example saves the parameter structure returned for `rtwdemo_rsimtf` to the MAT-file `myrsimdemo.mat`.

```
save myrsimdemo.mat param_struct;
```

For information on using command-line options to specify required files, see “Run Rapid Simulations” on page 68-13.

Convert the Parameter Structure for Running Simulations on Varying Data Sets

To use rapid simulations to test changes to specific parameters, you can convert the model parameter structure to a cell array. You can then access a specific parameter set by using the @ operator to specify the index for a specific parameter set in the file.

To convert the structure to a cell array:

1. Use the function `rsimgetrtp` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   ```

   The `parameters` field of the structure is a substructure that contains parameter information. The `values` field of the `parameters` substructure contains the numeric values of the parameters that you can tune during execution of the simulation code.
2. Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

   ```
   param_struct = rsimsetrtpparam(param_struct,3);
   ```

   The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.
3. Specify new values for the parameters in the second and third parameter sets.

   ```
   param_struct.parameters{2}.values = [-150 -5000 0 4950];
   param_struct.parameters{3}.values = [-170 -5500 0 5100];
   ```
4. Save the structure containing the parameter set information to a MAT-file.

   ```
   save rtwdemo_rsimtf.mat param_struct;
   ```

Alternatively, you can modify the block parameters in the model, and use `rsimgetrtp` to create multiple parameter sets:

1. Use the function `rsimgetrtp` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   ```
2. Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

   ```
   param_struct = rsimsetrtpparam(param_struct,3);
   ```

   The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.

3. Change the values of block parameters or workspace variables. For example, change the value of the variable `w` from `70` to `72`.

   ```
   w = 72;
   ```

4. Use `rimsgetrtp` to get another structure containing parameter information. Store the structure in a temporary variable `rtp_temp`.

   ```
   rtp_temp = rsimgetrtp('rtwdemo_rsimtf');
   ```

5. Assign the value of the `parameters` field of `rtp_temp` to the structure `param_struct` as a second parameter set.

   ```
   param_struct.parameters{2} = rtp_temp.parameters;
   ```

6. Change the value of the variable `w` from `72` to `75`.

   ```
   w = 75;
   ```

7. Use `rimsgetrtp` to get another structure containing parameter information. Then, assign the value of the `parameters` field to `param_struct` as a third parameter set.

   ```
   rtp_temp = rsimgetrtp('rtwdemo_rsimtf');
   param_struct.parameters{3} = rtp_temp.parameters;
   ```

8. Save the structure containing the parameter set information to a MAT-file.

   ```
   save rtwdemo_rsimtf.mat param_struct;
   ```

For more information on how to specify each parameter set when you run the simulations, see “Change Block Parameters for an RSim Simulation” on page 68-20.

Create a MAT-File for a From File Block

You can use a MAT-file as the input data source for a From File block. The format of the data in the MAT-file must match the data format expected by that block. For example, if you are using a matrix as an input for the MAT file, this cannot be different from the matrix size for the executable.

To create a MAT-file for a From File block:

1. For array format data, in the workspace create a matrix that consists of two or more rows. The first row must contain monotonically increasing time points. Other rows contain data points that correspond to the time point in that column. The time and data points must be data of type `double`.

   For example:

   ```
   t=[0:0.1:2*pi]';
   Ina1=[2*sin(t) 2*cos(t)];
   Ina2=sin(2*t);
   Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];
   var_matrix=[t Ina1 Ina2 Ina3]';
   ```

   For other supported data types, such as `int16` or fixed-point, the time data points must be of type `double`, just as for array format data. However, the sample data can be of any dimension.

   For more information on setting up the input data, see the block description of From File.

2. Save the matrix to a MAT-file.

The following example saves the matrix `var_matrix` to the MAT-file `myrsimdemo.mat` in Version 7.3 format.

```
save '-v7.3' myrsimdemo.mat var_matrix;
```

Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

Create a MAT-File for an Inport Block

You can use a MAT-file as the input data source for an Inport block.

The format of the data in the MAT-file must adhere to one of the three column-based formats listed in the following table. The table lists the formats in order from least flexible to most flexible.

Format	Description
Single time/data matrix	• Least flexible. • One variable. • Two or more *columns*. Number of columns must equal the sum of the dimensions of all root Inport blocks plus 1. First column must contain monotonically increasing time points. Other columns contain data points that correspond to the time point in a given row. • Data of type `double`. For an example, see **Single time/data matrix** in the following procedure, step 4. For more information, see “Loading Data Arrays to Root-Level Inputs”.
Signal-and-time structure	• More flexible than the single time/data matrix format. • One variable. • Must contain two top-level fields: `time` and `signals`. The `time` field contains a *column* vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to an Inport block. The substructure index corresponds to the Inport block number. Each `signals` substructure must contain a field named `values`. The `values` field must contain an array of inputs for the corresponding Inport block, where each input corresponds to a time point specified by the `time` field. • If the `time` field is set to an empty value, clear the check box for the Inport block **Interpolate data** parameter. • Data type must match Inport block settings. For an example, see **Signal-and-time structure** in the following procedure, step 4. For more information on this format, see “Loading Data Structures to Root-Level Inputs”.

Format	Description
Per-port structure	• Most flexible • Multiple variables. Number of variables must equal the number of Inport blocks. • Consists of a separate structure-with-time or structure-without-time for each Inport block. Each Inport block data structure has only one `signals` field. To use this format, in the **Input** text field, enter the names of the structures as a comma-separated list, `in1`, `in2`,..., `inN`, where `in1` is the data for your model's first port, `in2` for the second port, and so on. • Each variable can have a different time vector. • If the `time` field is set to an empty value, clear the check box for the Inport block **Interpolate data** parameter. • Data type must match Inport block settings. • To save multiple variables to the same data file, you must save them in the order expected by the model, using the `-append` option. For an example, see **Per-port structure** in the following procedure, step 4. For more information, see “Loading Data Structures to Root-Level Inputs”.

The supported formats and the following procedure are illustrated in `rtwdemo_rsim_i`.

To create a MAT-file for an Inport block:

1. Choose one of the preceding data file formats.
2. Update Inport block parameter settings and specifications to match specifications of the data to be supplied by the MAT-file.

   By default, the Inport block inherits parameter settings from downstream blocks. To import data from an external MAT-file, explicitly set the following parameters to match the source data in the MAT-file.

   - **Main** > **Interpolate data**
   - **Signal Attributes** > **Port dimensions**
   - **Signal Attributes** > **Data type**
   - **Signal Attributes** > **Signal type**

   If you choose to use a structure format for workspace variables and the `time` field is empty,clear block parameter **Interpolate data** or modify the parameter so that it is set to a nonempty value. Interpolation requires time data.

   For descriptions of the preceding block parameters, see the block description of Inport.
3. Build an RSim executable program for the model. The build process creates and calculates a structural checksum for the model and embeds it in the generated executable. The RSim target uses the checksum to verify that data being passed into the model is consistent with what the model executable expects.
4. Create the MAT-file that provides the source data for the rapid simulations. You can create the MAT-file from a workspace variable. Using the specifications in the preceding format comparison table, create the workspace variables for your simulations.

   An example of each format follows:

Single time/data matrix

```
t=[0:0.1:2*pi]';
Ina1=[2*sin(t) 2*cos(t)];
Ina2=sin(2*t);
Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];
var_matrix=[t Ina1 Ina2 Ina3];
```

Signal-and-time structure

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

Per-port structure

```
t=[0:0.1:2*pi]';
Inb1.time=t;
Inb1.signals.values(:,1)=2*sin(t);
Inb1.signals.values(:,2)=2*cos(t);
t=[0:0.2:2*pi]';
Inb2.time=t;
Inb2.signals.values(:,1)=sin(2*t);
t=[0:0.1:2*pi]';
Inb3.time=t;
Inb3.signals.values(:,1)=0.5*sin(3*t);
Inb3.signals.values(:,2)=0.5*cos(3*t);
```

5 Save the workspace variables to a MAT-file.

Single time/data matrix

The following example saves the workspace variable `var_matrix` to the MAT-file `rsim_i_matrix.mat`.

```
save rsim_i_matrix.mat var_matrix;
```

Signal-and-time structure

The following example saves the workspace structure variable `var_single_struct` to the MAT-file `rsim_i_single_struct.mat`.

```
save rsim_i_single_struct.mat var_single_struct;
```

Per-port structure

To order data when saving per-port structure variables to a single MAT-file, use the `save` command's `-append` option. Be sure to append the data in the order that the model expects it.

The following example saves the workspace variables `Inb1`, `Inb2`, and `Inb3` to MAT-file `rsim_i_multi_struct.mat`.

```
save rsim_i_multi_struct.mat Inb1;
save rsim_i_multi_struct.mat Inb2 -append;
save rsim_i_multi_struct.mat Inb3 -append;
```

The `save` command does not preserve the order in which you specify your workspace variables in the command line when saving data to a MAT-file. For example, if you specify the variables `v1`, `v2`, and `v3`, in that order, the order of the variables in the MAT-file could be `v2 v1 v3`.

Using a command-line option, you can then specify the MAT-files as input for rapid simulations.

Scripts for Batch and Monte Carlo Simulations

The RSim target is for batch simulations in which parameters and input signals vary for multiple simulations. New output file names allow you to run new simulations without overwriting prior simulation results. You can set up a series of simulations to run by creating a `.bat` file for use on a Microsoft Windows platform.

Create a file for the Windows platform with a text editor and execute it by typing the file name, for example, `mybatch`, where the name of the text file is `mybatch.bat`.

```
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run1.mat -o results1.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run2.mat -o results2.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run3.mat -o results3.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run4.mat -o results4.mat -tf 10.0
```

In this case, batch simulations run using four sets of input data in files `run1.mat`, `run2.mat`, and so on. The RSim executable saves the data to the files specified with the `-o` option.

The variable names containing simulation results in each of the files are identical. Therefore, loading consecutive sets of data without renaming the data once it is in the MATLAB workspace results in overwriting the prior workspace variable with new data. To avoid overwriting, you can copy the result to a new MATLAB variable before loading the next set of data.

You can also write MATLAB scripts to create new signals and new parameter structures, as well as to save data and perform batch runs using the bang command (`!`).

For details on running simulations and available command-line options, see “Run Rapid Simulations” on page 68-13. For an example of a rapid simulation batch script, see the example “Run Batch Simulations Without Recompiling Generated Code”.

Run Rapid Simulations

- “Rapid Simulations” on page 68-14
- “Requirements for Running Rapid Simulations” on page 68-15
- “Set a Clock Time Limit for a Rapid Simulation” on page 68-15
- “Override a Model Simulation Stop Time” on page 68-15
- “Read the Parameter Vector into a Rapid Simulation” on page 68-16
- “Specify New Signal Data File for a From File Block” on page 68-16
- “Specify Signal Data File for an Inport Block” on page 68-18
- “Change Block Parameters for an RSim Simulation” on page 68-20

- "Specify a New Output File Name for a Simulation" on page 68-21
- "Specify New Output File Names for To File Blocks" on page 68-21

Rapid Simulations

Using the RSim target, you can build a model once and run multiple simulations to study effects of varying parameter settings and input signals. You can run a simulation directly from your operating system command line, redirect the command from the MATLAB command line by using the bang (!) character, or execute commands from a script.

From the operating system command line, use

```
rtwdemo_rsimtf
```

From the MATLAB command line, use

```
!rtwdemo_rsimtf
```

The following table lists ways you can use RSim target command-line options to control a simulation.

To...	Use...
Read input data for a From File block from a MAT-file other than the MAT-file used for the previous simulation	*model* `-f` *oldfilename*`.mat=`*newfilename*`.mat`
Print a summary of the options for RSim executable targets	*executable filename* `-h`
Read input data for an Inport block from a MAT-file	*model* `-i` *filename*`.mat`
Time out after *n* clock time seconds, where *n* is a positive integer value	*model* `-L` *n*
Write MAT-file logging data to file *filename*`.mat`	*model* `-o` *filename*`.mat`
Read a parameter vector from file *filename*`.mat`	*model* `-p` *filename*`.mat`
Override the default TCP port (17725) for external mode	*model* `-port` *TCPport*
Write MAT-file logging data to a MAT-file other than the MAT-file used for the previous simulation	*model* `-t` *oldfilename*`.mat=`*newfilename*`.mat`
Run the simulation until the time value *stoptime* is reached	*model* `-tf` *stoptime*
Run in verbose mode	*model* `-v`
Wait for the Simulink engine to start the model in external mode	*model* `-w`

The following sections use the `rtwdemo_rsimtf` example model in examples to illustrate some of these command-line options. In each case, the example assumes you have already done the following:

- Created or changed to a working folder.
- Opened the example model.
- Copied the data file *matlabroot*`/toolbox/rtw/rtwdemos/rsimdemos/rsim_tfdata.mat` to your working folder. You can perform this operation using the command:

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos',...
'rsimdemos','rsim_tfdata.mat'),pwd);
```

Requirements for Running Rapid Simulations

The following requirements apply to both fixed and variable step executables.

- You must run the RSim executable on a computer configured to run MATLAB. Also, the `RSim.exe` file must be able to access the MATLAB and Simulink installation folders on this machine. To obtain that access, your PATH environment variable must include /bin and /bin/($ARCH), where ($ARCH) represents your operating system architecture. For example, for a personal computer running on a Windows platform, ($ARCH) is “win64”, whereas for a Linux machine, ($ARCH) is “glnxa64”.
- On GNU Linux platforms, to run an RSim executable, define the LD_LIBRARY_PATH environment variable to provide the path to the MATLAB installation folder, as follows:

  ```
  % setenv LD_LIBRARY_PATH /matlab/sys/os/glnx64:$LD_LIBRARY_PATH
  ```
- On the Apple Macintosh OS X platform, to run RSim target executables, you must define the environment variable `DYLD_LIBRARY_PATH` to include the folders `bin/mac` and `sys/os/mac` under the MATLAB installation folder. For example, if your MATLAB installation is under `/MATLAB`, add `/MATLAB/bin/mac` and `/MATLAB/sys/os/mac` to the definition for `DYLD_LIBRARY_PATH`.

Set a Clock Time Limit for a Rapid Simulation

If a model experiences frequent zero crossings and the model's minor step size is small, consider setting a time limit for a rapid simulation. To set a time limit, specify the `-L` option with a positive integer value. The simulation aborts after running for the specified amount of clock time (not simulation time). For example,

```
!rtwdemo_rsimtf -L 20
```

Based on your clock, after the executable runs for 20 seconds, the program terminates. You see a message similar to one of the following:

- On a Microsoft Windows platform,

  ```
  Exiting program, time limit exceeded
  Logging available data ...
  ```
- On The Open Group UNIX platform,

  ```
  ** Received SIGALRM (Alarm) signal @ Fri Jul 25 15:43:23 2003
  ** Exiting model  'rtwdemo_rsimtf' @ Fri Jul 25 15:43:23 2003
  ```

You do not need to do anything to your model or to its configuration to use this option.

Override a Model Simulation Stop Time

By default, a rapid simulation runs until the simulation time reaches the time specified the Configuration Parameters dialog box, on the **Solver** pane. You can override the model simulation stop time by using the `-tf` option. For example, the following simulation runs until the time reaches 6.0 seconds.

```
!rtwdemo_rsimtf -tf 6.0
```

The RSim target stops and logs output data using MAT-file data logging rules.

If the model includes a From File block, the end of the simulation is regulated by the stop time setting specified in the Configuration Parameters dialog box, on the **Solver** pane, or with the RSim target option `-tf`. The values in the block time vector are ignored. However, if the simulation time exceeds the endpoints of the time and signal matrix (if the final time is greater than the final time value of the data matrix), the signal data is extrapolated to the final time value.

Read the Parameter Vector into a Rapid Simulation

To read the model parameter vector into a rapid simulation, you must first create a MAT-file that includes the parameter structure as described in "Create a MAT-File That Includes a Model Parameter Structure" on page 68-6. You can then specify the MAT-file in the command line with the `-p` option.

For example:

1. Build an RSim executable for the example model `rtwdemo_rsimtf`.
2. Modify parameters in your model and save the parameter structure.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   save myrsimdata.mat param_struct
   ```

3. Run the executable with the new parameter set.

   ```
   !rtwdemo_rsimtf -p myrsimdata.mat
   ```

   ```
   ** Starting model 'rtwdemo_rsimtf' @ Tue Dec 27 12:30:16 2005
   ** created rtwdemo_rsimtf.mat **
   ```

4. Load workspace variables and plot the simulation results by entering the following commands:

   ```
   load rtwdemo_rsimf.mat
   plot(rt_yout)
   ```

Specify New Signal Data File for a From File Block

If your model's input data source is a From File block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. Each MAT-file must adhere to the format described in "Create a MAT-File for a From File Block" on page 68-9.

To change the MAT-file after an initial simulation, you specify the executable with the `-f` option and an *`oldfile`*`.mat=`*`newfile`*`.mat` parameter, as shown in the following example.

1. Set some parameters in the MATLAB workspace. For example:

   ```
   w = 100;
   theta = 0.5;
   ```

2. Build an RSim executable for the example model `rtwdemo_rsimtf`.
3. Run the executable.

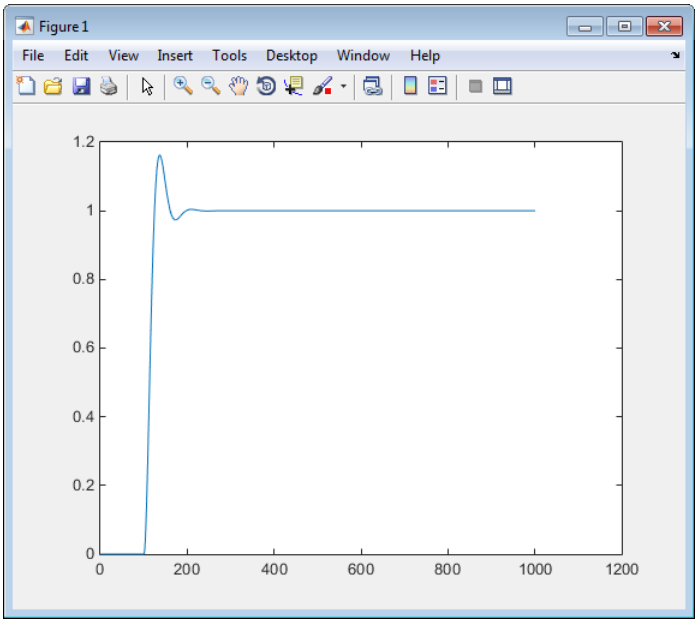
   ```
   !rtwdemo_rsimtf
   ```

   The RSim executable runs a set of simulations and creates output MAT-files containing the specific simulation result.

4. Load the workspace variables and plot the simulation results by entering the following commands:

```
load rtwdemo_rsimtf.mat
plot(rt_yout)
```

The resulting plot shows simulation results based on default input data.



5 Create a new data file, `newfrom.mat`, that includes the following data:

```
t=[0:.001:1];
u=sin(100*t.*t);
tu=[t;u];
save newfrom.mat tu;
```

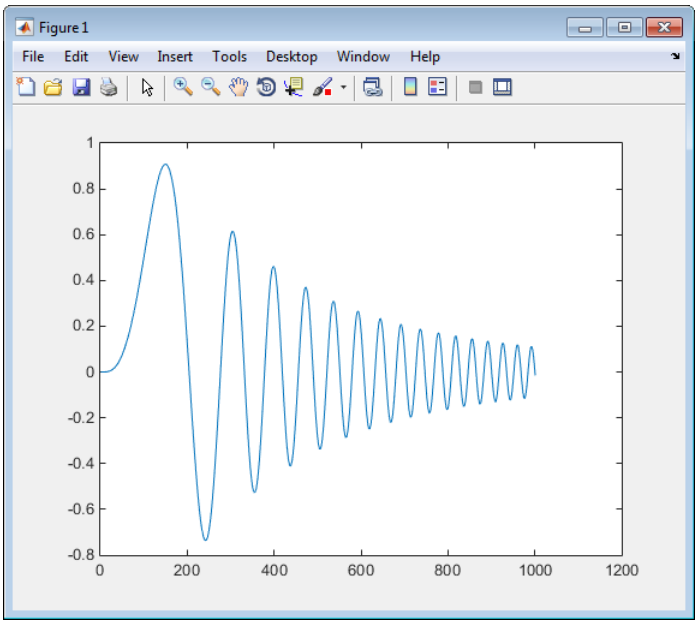
6 Run a rapid simulation with the new data by using the `-f` option to replace the original file, `rsim_tfdata.mat`, with `newfrom.mat`.

```
!rtwdemo_rsimtf -f rsim_tfdata.mat=newfrom.mat
```

7 Load the data and plot the new results by entering the following commands:

```
load rtwdemo_rsimtf.mat
plot(rt_yout)
```

The next figure shows the resulting plot.



From File blocks require input data of type `double`. If you need to import signal data of a data type other than `double`, use an Inport block (see “Create a MAT-File for an Inport Block” on page 68-10) or a From Workspace block with the data specified as a structure.

Workspace data must be in the format:

```
variable.time
variable.signals.values
```

If you have more than one signal, use the following format:

```
variable.time
variable.signals(1).values
variable.signals(2).values
```

Specify Signal Data File for an Inport Block

If your model's input data source is an Inport block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. Each MAT-file must adhere to one of the three formats described in “Create a MAT-File for an Inport Block” on page 68-10.

To specify the MAT-file after a simulation, you specify the executable with the `-i` option and the name of the MAT-file that contains the input data. For example:

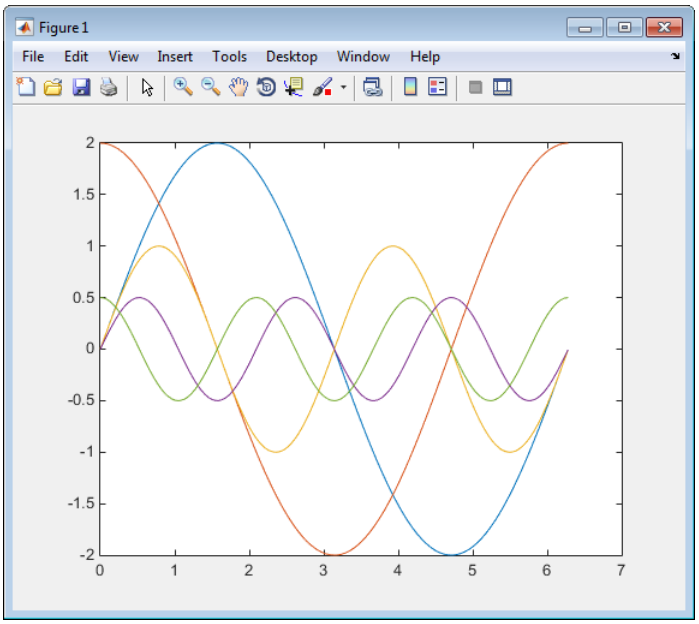
1. Open the model `rtwdemo_rsim_i`.
2. Check the Inport block parameter settings. The following Inport block data parameter settings and specifications that you specify for the workspace variables must match settings in the MAT-file, as indicated in “Configure Inports to Provide Simulation Source Data” on page 68-4:
   - **Main > Interpolate data**
   - **Signal Attributes > Port dimensions**
   - **Signal Attributes > Data type**

- **Signal Attributes > Signal type**

3 Build the model.

4 Set up the input signals. For example:

```
t=[0:0.01:2*pi]';
s1=[2*sin(t) 2*cos(t)];
s2=sin(2*t);
s3=[0.5*sin(3*t) 0.5*cos(3*t)];
plot(t, [s1 s2 s3])
```



5 Prepare the MAT-file by using one of the three available file formats described in "Create a MAT-File for an Inport Block" on page 68-10. The following example defines a signal-and-time structure in the workspace and names it `var_single_struct`.

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

6 Save the workspace variable `var_single_struct` to MAT-file `rsim_i_single_struct`.

```
save rsim_i_single_struct.mat var_single_struct;
```

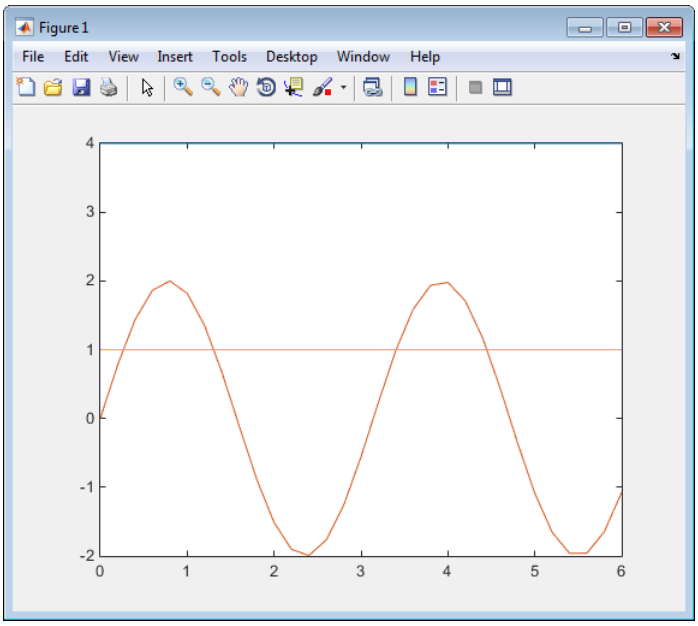
7 Run a rapid simulation with the input data by using the `-i` option. Load and plot the results.

```
!rtwdemo_rsim_i -i rsim_i_single_struct.mat
```

```
** Starting model 'rtwdemo_rsim_i' @ Tue Aug 19 10:26:53 2014
 *** rsim_i_single_struct.mat is successfully loaded! ***
** created rtwdemo_rsim_i.mat **

** Execution time = 0.02024185130718954s
```

8 Load and plot the results.

```
load rtwdemo_rsim_i.mat
plot(rt_tout, rt_yout);
```



Change Block Parameters for an RSim Simulation

As described in “Create a MAT-File That Includes a Model Parameter Structure” on page 68-6, after you alter one or more parameters in a Simulink block diagram, you can extract the parameter vector, *model*_P, for the entire model. You can then save the parameter vector, along with a model checksum, to a MAT-file. This MAT-file can be read directly by the standalone RSim executable, allowing you to replace the entire parameter vector or individual parameter values, for running studies of variations of parameter values representing coefficients, new data for input signals, and so on.

RSim can read the MAT-file and replace the entire *model*_P structure whenever you change one or more parameters, without recompiling the entire model.

For example, assume that you changed one or more parameters in your model, generated the new *model*_P vector, and saved *model*_P to a new MAT-file called `mymatfile.mat`. To run the same `rtwdemo_rsimtf` model and use these new parameter values, use the `-p` option, as shown in the following example:

```
!rtwdemo_rsimtf -p mymatfile.mat
load rtwdemo_rsimtf
plot(rt_yout)
```

If you have converted the parameter structure to a cell array for running simulations on varying data sets, as described in “Convert the Parameter Structure for Running Simulations on Varying Data Sets” on page 68-8, you must add an @*n* suffix to the MAT-file specification. *n* is the element of the cell array that contains the specific input that you want to use for the simulation.

The following example converts `param_struct` to a cell array, changes parameter values, saves the changes to MAT-file `mymatfile.mat`, and then runs the executable using the parameter values in the second element of the cell array as input.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf');
param_struct = rsimsetrtpparam(param_struct,2);
param_struct.parameters{1}
```

```
ans =

       dataTypeName: 'double'
         dataTypeId: 0
            complex: 0
         dtTransIdx: 0
             values: [-140 -4900 0 4900]
                map: []
    structParamInfo: []
```

```
param_struct.parameters{2}.values=[-150 -5000 0 4950];
save mymatfile.mat param_struct;
!rtwdemo_rsimtf -p mymatfile.mat@2 -o rsim2.mat
```

Specify a New Output File Name for a Simulation

If you have specified one or more of the **Save to Workspace** parameters — **Time**, **States**, **Outputs**, or **Final States** — in the Configuration Parameters dialog box, on the **Data Import/Export** pane, the default is to save simulation logging results to the file *`model`*`.mat`. For example, the example model `rtwdemo_rsimtf` normally saves data to `rtwdemo_rsimtf.mat`, as follows:

```
!rtwdemo_rsimtf
created rtwdemo_rsimtf.mat
```

You can specify a new output file name for data logging by using the `-o` option when you run an executable.

```
!rtwdemo_rsimtf -o rsim1.mat
```

In this case, the set of parameters provided at the time of code generation, including From File block data parameters, is run.

Specify New Output File Names for To File Blocks

In much the same way as you can specify a new system output file name, you can also provide new output file names for data saved from one or more To File blocks. To do this, specify the original file name at the time of code generation with a new name, as shown in the following example:

```
!rtwdemo_rsimtf -t rtwdemo_rsimtf_data.mat=mynewrsimdata.mat
```

In this case, assume that the original model wrote data to the output file `rtwdemo_rsimtf_data.mat`. Specifying a new file name forces RSim to write to the file `mynewrsimdata.mat`. With this technique, you can avoid overwriting an existing simulation run.

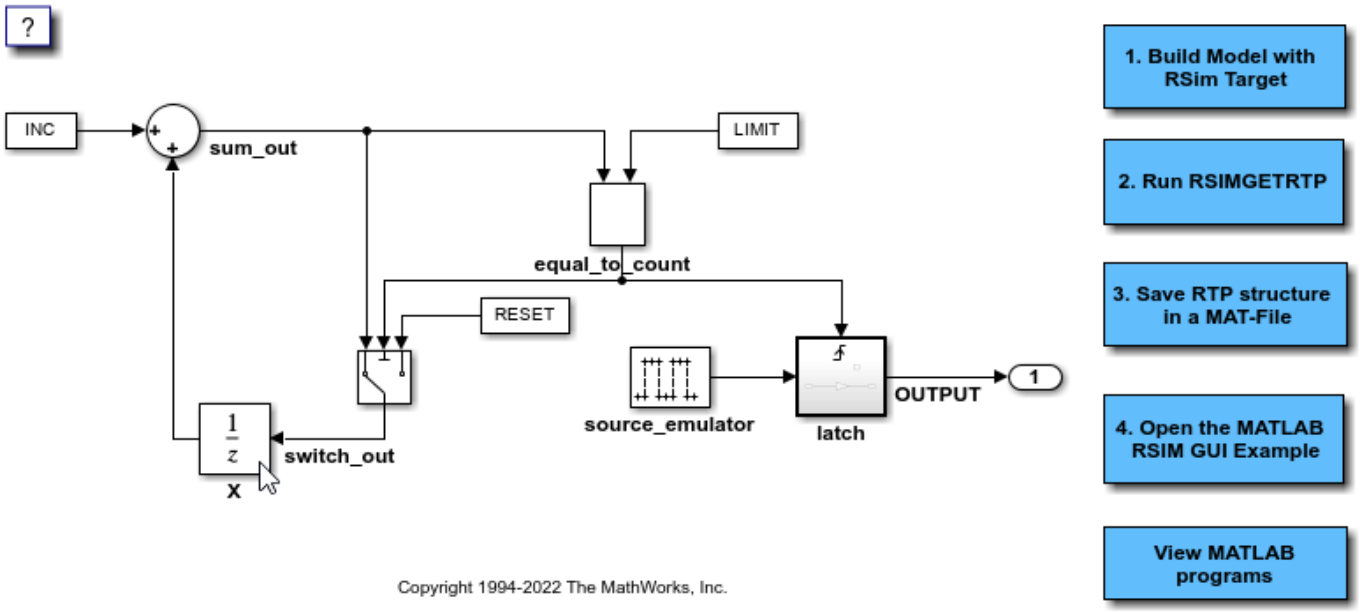
Tune Parameters Interactively During Rapid Simulation

The RSim system target file was designed to let you run batch simulations at the fastest possible speed. Using variable-step or fixed-step solvers with that system target file and a tunable parameter

data structure, whether you set **Default parameter behavior** to `Tunable` or to `Inlined`, you can create multiple parameter sets. You can run the data sets with the generated executable program (.exe on Windows). Each invocation of the executable program allows specification of the file name to use for results.

Open the example model `RsimParamTuning`.

```
open_system('RsimParamTuning.slx');
```



For this model, **Default parameter behavior** is set to `Inlined`. The model declares workspace variables as tunable parameters. To use the RSim system target file with **Default parameter behavior** set to `Tunable`, and without explicitly declaring tunable parameters, see “Run Batch Simulations Without Recompiling Generated Code”.

This model uses the RSim system target file and the `rsimgetrtp` function to allow a non real time executable to be passed new data without the need to recompile the Simulink model. This feature allows you to get a map of the tunable parameters declared in a model and save it in a MAT-file. You can then create your own MATLAB graphical user interface (GUI) or a standalone GUI (independent of MATLAB) to read and write the MAT-file and rerun the executable program to produce new output files.

In the model, double-click the buttons sequentially to run the example.

To review the code used to create the MATLAB and standalone GUIs, double-click the **View MATLAB programs** button.

For more information, see “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File”.

Rapid Simulation Target Limitations

The RSim target has the following limitations:

- Does not support algebraic loops.
- Does not support Interpreted MATLAB Function blocks.
- Does not support noninlined MATLAB language or Fortran S-functions.
- If an RSim build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In such cases, you must regenerate the code for the model.

See Also

More About

- “Acceleration”
- “Use S-Function Target for Model or Subsystem”

Run Rapid Simulations over a Range of Parameter Values

This example shows how to use the RSim system target file to run simulations over a range of parameter values. The example uses the Van der Pol oscillator and performs a parameter sweep over a range of initial state values to obtain the phase diagram of a nonlinear system.

You can customize this example for your own application by modifying the MATLAB® script used to build this example. Click the link in the top left corner of this page to edit the MATLAB script. Click the link in the top right corner to run this example from MATLAB. Run the example from writable location. The example creates files that you may want to investigate later.

For this example, set model configuration parameter **Default parameter behavior** to `Inlined`. In the example, you create `Simulink.Parameter` objects as tunable parameters. To use RSim with **Default parameter behavior** set to `Tunable`, and without explicitly declaring tunable parameters, see “Run Batch Simulations Without Recompiling Generated Code”.

To run multiple simulations in the Simulink® environment, consider using rapid accelerator instead of RSim. For information about rapid accelerator, see “What Is Acceleration?”. To sweep parameter values, see “Optimize, Estimate, and Sweep Block Parameter Values”.

Preparation Model

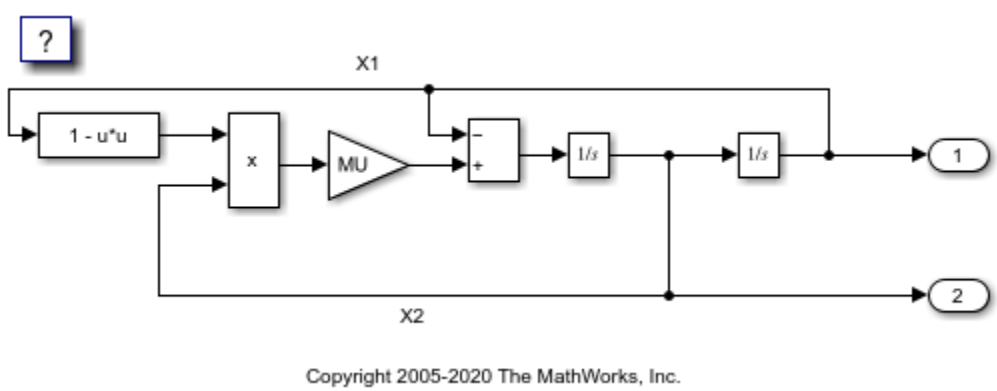
Make sure the current folder is writable. The example creates files.

```
[stat, fa] = fileattrib(pwd);
if ~fa.UserWrite
    disp('This script must be run in a writable directory');
    return
end
```

Open the model. Then, open the **Simulink Coder** app. Set model configuration parameter **System target file** to `rsim.tlc`.

For more information on doing this graphically and setting up other RSim target related options, see “Configure a System Target File”.

```
mdlName = 'rtwdemo_rsim_vdp';
open_system(mdlName);
cs = getActiveConfigSet(mdlName);
cs.switchTarget('rsim.tlc',[]);
```



Specify variables `INIT_X1` (the initial condition for state x1), `INIT_X2` (the initial condition for state `x2`), and `MU` (the gain value) as tunable. To create tunable parameters, convert the variables to `Simulink.Parameter` objects, and configure the objects to use a storage class other than `Auto`. In this example you investigate how the state trajectories evolve from different initial values for states `x1` and `x2` in the model.

```
INIT_X1 = Simulink.Parameter(INIT_X1);
INIT_X1.StorageClass = 'Model default';

INIT_X2 = Simulink.Parameter(INIT_X2);
INIT_X2.StorageClass = 'Model default';

MU = Simulink.Parameter(MU);
MU.StorageClass = 'Model default';
```

Define the names of files that will be created.

```
prmFileName = [mdlName, '_prm_sets.mat'];
logFileName = [mdlName, '_run_scr.log'];
batFileName = [mdlName, '_run_scr'];
exeFileName = mdlName;
if ispc
    exeFileName = [exeFileName, '.exe'];
    batFileName = [batFileName, '.bat'];
end
aggDataFile = [mdlName, '_results'];
startTime = cputime;
```

Build Model

Build the RSim executable program for the model. During the build process, a structural checksum is calculated for the model and embedded into the generated executable program. This checksum is used to check that a parameter set passed to the executable program is compatible with the program.

```
slbuild(mdlName);
```

```
### Starting build procedure for: rtwdemo_rsim_vdp
### Successful completion of build procedure for: rtwdemo_rsim_vdp

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
========================================================================================================
rtwdemo_rsim_vdp  Code generated and compiled  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 37.492s
```

Get Default Parameter Set for Model

Get the default `rtP` structure (parameter set) for the model. The `modelChecksum` field in the `rtP` structure is the structural checksum of the model. This must match the checksum embedded in the RSim executable program. If the two checksums do not match, the executable program produces an error. `rsimgetrtp` generates an `rtP` structure with entries for the named tunable variables `INIT_X1`, `INIT_X2`, and `MU`.

```
rtp = rsimgetrtp(mdlName)
```

```
rtp = 

  struct with fields:

                      modelChecksum: [410569968 1.5914e+09 3.4411e+09 … ]
                         parameters: [1×1 struct]
                 globalParameterInfo: [1×1 struct]
    collapsedBaseWorkspaceVariables: []
                 nonTunableVariables: [1×0 struct]
```

Create Parameter Sets

Using the `rtp` structure, build a structure array with different values for the tunable variables in the model. As mentioned earlier, in this example you want see how the state trajectories evolve for different initial values for states `x1` and `x2` in the model. To do this, generate different parameter sets with different values for `INIT_X1` and `INIT_X2` and leave tunable variable `MU` set to the default value.

```
INIT_X1_vals = -5:1:5;
INIT_X2_vals = -5:1:5;
MU_vals = 1;
nPrmSets = length(INIT_X1_vals)*length(INIT_X2_vals)*length(MU_vals)
```

```
nPrmSets = 

   121
```

This example has `nPrmSets` parameter sets. You need to run that many simulations. Initialize `aggData`, which is a structure array that holds the parameter set and corresponding results.

```
aggData = struct('tout', [], 'yout', [], ...
                'prms', struct('INIT_X1',[],'INIT_X2',[], 'MU', []))
aggData = repmat(aggData, nPrmSets, 1);
```

```
aggData = 

  struct with fields:

    tout: []
    yout: []
    prms: [1×1 struct]
```

The utility function `rsimsetrtpparam` builds the `rtP` structure by adding parameter sets one at a time with different parameters values.

```
idx = 1;
for iX1 = INIT_X1_vals
    for iX2 = INIT_X2_vals
        for iMU = MU_vals
            rtp = rsimsetrtpparam(rtp,idx,'INIT_X1',iX1,'INIT_X2',iX2,'MU',iMU);
```

```
            aggData(idx).prms.INIT_X1 = iX1;
            aggData(idx).prms.INIT_X2 = iX2;
            aggData(idx).prms.MU      = iMU;
            idx = idx + 1;
        end
    end
end
```

Save the `rtP` structure array with the parameter sets to a MAT-file.

```
save(prmFileName,'rtp');
```

Create Batch File

Create a batch script file to run the RSim executable program over the parameter sets. Each run reads the specified parameter set from the parameter MAT-file and writes the results to the specified output MAT-file. Use the time out option so that if a run hangs (for example, because the model has a singularity for that parameter set), abort the run after the specified time limit and proceed to the next run.

For example, this command (on Windows®) specifies using the third parameter set from the `rtP` structure in `prm.mat`, writes the results to `run3.mat`, and aborts execution if a run takes longer than 3600 seconds of CPU time. While running, the script pipes messages from `model.exe` to `run.log`, which you can use to debug issues.

```
  model.exe -p prm.mat@3 -o run3.mat -L 3600 2>&1>> run.log
```

```
fid = fopen(batFileName, 'w');
idx = 1;
for iX1 = INIT_X1_vals
    for iX2 = INIT_X2_vals
        for iMU = MU_vals
            outMatFile = [mdlName, '_run',num2str(idx),'.mat'];
            cmd = [exeFileName, ...
                   ' -p ', prmFileName, '@', int2str(idx), ...
                   ' -o ', outMatFile, ...
                   ' -L 3600'];
            if ispc
                cmd  = [cmd, ' 2>&1>> ', logFileName];
            else % (unix)
                cmd  = ['.' filesep cmd, ' 1> ', logFileName, ' 2>&1'];
            end
            fprintf(fid, ['echo "', cmd, '"\n']);
            fprintf(fid, [cmd, '\n']);
            idx = idx + 1;
        end
    end
end
if isunix,
    system(['touch ', logFileName]);
    system(['chmod +x ', batFileName]);
end
fclose(fid);
```

Creating a batch/script file to run simulations enables you to call the system command once to run the simulations (or even run the batch script outside MATLAB) instead of calling the system command in a loop for each simulation. This improves performance because the system command has significant overhead.

Execute Batch File to Run Simulations

Run the batch/script file, which runs the RSim executable program once for each parameter set and saves the results to a different MAT-file each time. You can run the batch file from outside MATLAB.

```
[stat, res] = system(['.' filesep batFileName]);
if stat ~= 0
    error(['Error running batch file ''', batFileName, ''' :', res]);
end
```

This example logs simulation run results in one batch file as it runs the batch file to sequentially run `n` simulations over `n` parameter sets. You can modify the script to generate multiple batch files and run them in parallel by distributing them across multiple computers. Also, you can run the batch files without launching MATLAB.

Load Output MAT-Files and Collate Results

Collect the simulation results from the output MAT-files into the structure `aggData`. If the output MAT-file corresponding to a particular run is not found, set the results corresponding to that run to `NaN` (not a number). This situation occurs if a simulation run with a particular set of parameters encounters singularities in the model.

```
idx = 1;
for iX1 = INIT_X1_vals
    for iX2 = INIT_X2_vals
        for iMU = MU_vals
            outMatFile = [mdlName, '_run',num2str(idx),'.mat'];
            if exist(outMatFile,'file')
                load(outMatFile);
                aggData(idx).tout = rt_tout;
                aggData(idx).yout = rt_yout;
            else
                aggData(idx).tout = nan;
                aggData(idx).yout = nan;
            end
            idx = idx + 1;
        end
    end
end
```

Save the `aggData` structure to the results MAT-file. At this point, you can delete the other MAT-files, as the `aggData` data structure contains the aggregation of input (parameters sets) and output data (simulation results).

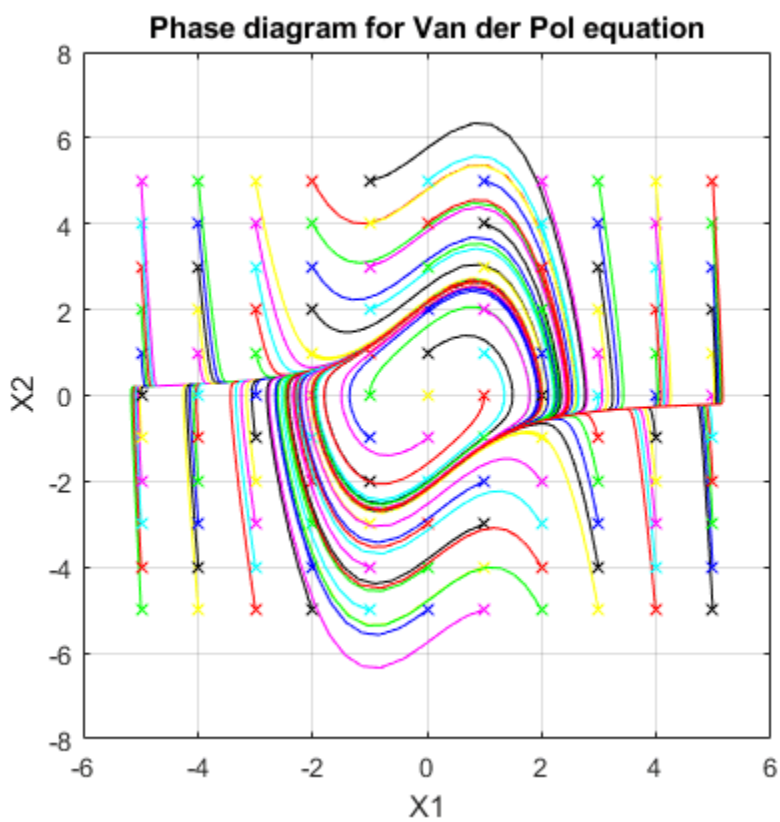
```
save(aggDataFile,'aggData');
disp(['Took ', num2str(cputime-startTime), ...
      ' seconds to generate results from ', ...
      num2str(nPrmSets), ' simulation runs (Steps 2 to 7).']);
```

```
Took 29.0781 seconds to generate results from 121 simulation runs (Steps 2 to 7).
```

Analyze Simulation Results

Plot the phase diagram (`X2` versus `X1`) with different initial values for `X1` and `X2`. The diagram shows that, irrespective of the initial condition, the Van der Pol oscillator converges to its natural oscillator mode.

```
colors = {'b','g','r','c','m','y','k'}; nColors = length(colors);
for idx = 1:nPrmSets
    col = colors{idx - nColors*floor(idx/nColors) + 1};
    plot(aggData(idx).prms.INIT_X1, aggData(idx).prms.INIT_X2, [col,'x'], ...
         aggData(idx).yout(:,1), aggData(idx).yout(:,2),col);
    hold on
end
grid on
xlabel('X1');
ylabel('X2');
axis('square');
title('Phase diagram for Van der Pol equation');
```



Run Batch Simulations Without Recompiling Generated Code

This example shows how to run batch simulations without recompiling the generated code. The example modifies input signal data and model parameters by reading data from a MAT-file. In the first part (steps 1-5), ten parameter sets are created from the Simulink® model by changing the transfer function damping factor. The ten parameter sets are saved to a MAT-file, and the RSim executable reads the specified parameter set from the file. In the second part (step 6-7) of this example, five sets of signal data chirps are created with increasingly high frequencies. In both parts, the RSim executable runs the set of simulations and creates output MAT-files containing the specific simulation result. Finally, a composite of runs appears in a MATLAB® figure.

To quickly run multiple simulations in the Simulink environment, consider using rapid accelerator instead of RSim. See “What Is Acceleration?”.

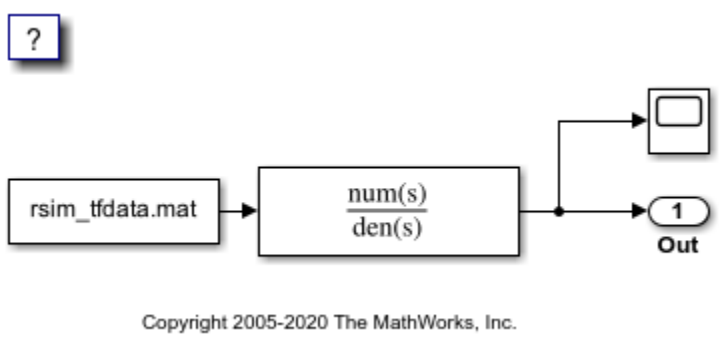
Step 1. Preparation

Make sure the current directory is writable because this example will be creating files.

```
[stat, fa] = fileattrib(pwd);
if ~fa.UserWrite
    disp('This script must be run in a writable directory');
    return;
end
```

Open the model and configure it to use the RSim target. For more information on doing this graphically and setting up other RSim target related options, see “Configure a System Target File” on page 50-2.

```
mdlName = 'rtwdemo_rsimtf';
open_system(mdlName);
cs = getActiveConfigSet(mdlName);
cs.switchTarget('rsim.tlc',[]);
```



The MAT-file rsim_tfdata.mat is required in the local directory.

```
if ~isempty(dir('rsim_tfdata.mat')),
    delete('rsim_tfdata.mat');
end
str1 = fullfile(matlabroot,'toolbox','rtw','rtwdemos','rsimdemos','rsim_tfdata.mat');
str2 = ['copyfile(''', str1, ''','''rsim_tfdata.mat''','''writable''')'];
eval(str2);
```

Step 2. Build the Model

Build the RSim executable for the model. During the build process, a structural checksum is calculated for the model and embedded into the generated executable. This checksum is used to check that a parameter set passed to the executable is compatible with it.

```
evalin('base','w = 70;')
evalin('base','theta = 1.0;')
disp('Building compiled RSim simulation.')
slbuild(mdlName);
```

```
Building compiled RSim simulation.
### Starting build procedure for: rtwdemo_rsimtf
### Successful completion of build procedure for: rtwdemo_rsimtf

Build Summary

Top model targets built:

Model           Action                        Rebuild Reason
========================================================================================
rtwdemo_rsimtf  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.7221s
```

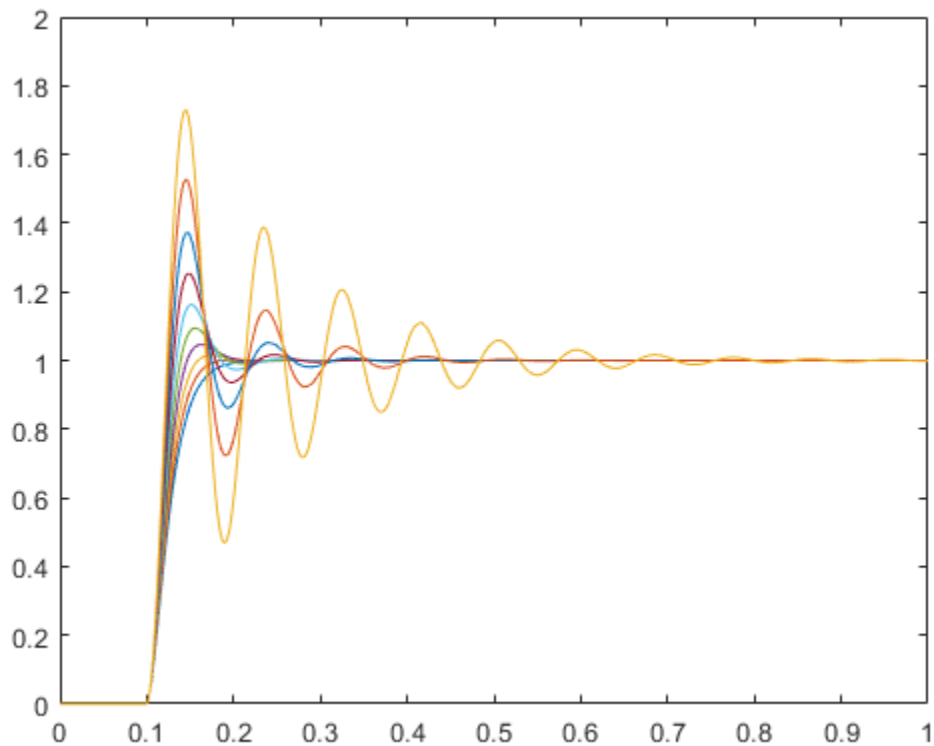
Step 3. Get the Default Parameter Set and Create 10 Parameters Sets

```
disp('Creating rtP data files')
for i=1:10
  % Extract current rtP structure using new damping factor.
  [rtpstruct]=evalin('base','rsimgetrtp(''rtwdemo_rsimtf'');');
  savestr = strcat('save params',num2str(i),'.mat rtpstruct');
  eval(savestr);
  evalin('base','theta = theta - .1;');
end
disp('Finished creating parameter data files.')
```

```
Creating rtP data files
Finished creating parameter data files.
```

Step 4. Run 10 RSim Simulations Using New Parameter Sets and Plot the Results

```
figure
for i=1:10
  % Bang out and run a simulation using new parameter data
  runstr = ['.', filesep, 'rtwdemo_rsimtf -p params',num2str(i),'.mat', ' -v'];
  [status, result] = system(runstr);
  if status ~= 0, error(result); end
  % Load simulation data into MATLAB for plotting.
  load rtwdemo_rsimtf.mat;
  axis([0 1 0 2]);
  plot(rt_tout, rt_yout)
  hold on
end
```



The plot shows 10 simulations, each using a different damping factor.

Step 5. Set Up a Time Vector and an Initial Frequency Vector

The time vector has 4096 points in the event we want to do windowing and spectral analysis on simulation results.

```
dt = .001;
nn = [0:1:4095];
t = dt*nn; [m,n] = size(t);
wlo = 1; whi = 4;
omega = [wlo:((whi-wlo)/n):whi - (whi-wlo)/n];
```

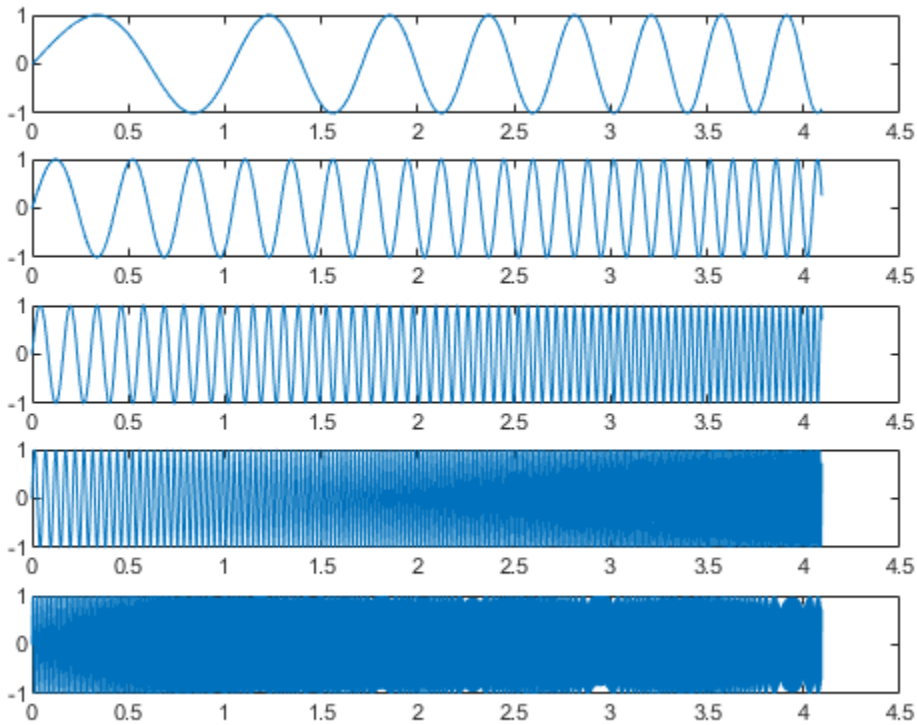
Step 6. Create 5 Sets of Signal Data in MAT Files

Creating .mat files with chirp data.

```
disp('This part of the example illustrates a sequence of 5 plots. Each')
disp('plot shows an input chirp signal of certain frequency range.')
for i = 1:5
  wlo = whi; whi = 3*whi; % keep increasing frequencies
  omega = [wlo:((whi-wlo)/n):whi - (whi-wlo)/n];
  % In a real application we recommend shaping the chirp using
  % a windowing function (hamming or hanning window, etc.)
  % This example does not use a windowing function.
  u      = sin(omega.*t);
  tudata = [t;u];
  % At each pass, save one more set of tudata to the next
```

```
  % .mat file.
  savestr = strcat('save sweep',num2str(i),'.mat tudata');
  eval(savestr);
  % Display each chirp. Note that this is only input data.
  % Simulations have not been run yet.
  plotstr = strcat('subplot(5,1,',num2str(i),');');
  eval(plotstr);
  plot(t,u)
  pause(0.3)
end
```

```
This part of the example illustrates a sequence of 5 plots. Each
plot shows an input chirp signal of certain frequency range.
```



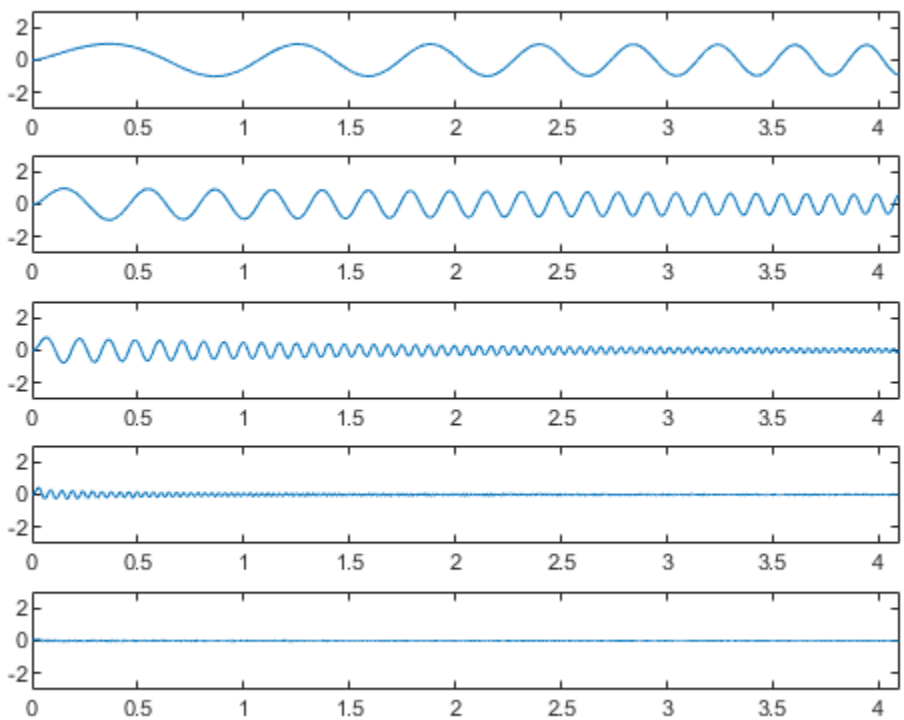
Step 7. Run the RSim Compiled Simulation Using New Signal Data

Replace the original signal data (rsim_tfdata.mat) with the files sweep1.mat, sweep2.mat, and so on.

```
disp('Starting batch simulations.')
for i = 1:5
  % Bang out and run the next set of data with RSim
  runstr = ['.', filesep, 'rtwdemo_rsimtf -f rsim_tfdata.mat=sweep', ...
            num2str(i),'.mat -v -tf 4.096'];
  [status, result] = system(runstr);
  if status ~= 0, error(result); end
  % Load the data to MATLAB and plot the results.
  load rtwdemo_rsimtf.mat
  plotstr = strcat('subplot(5,1,',num2str(i),');');
```

```
  eval(plotstr);
  plot(rt_tout, rt_yout); axis([0 4.1 -3 3]);
end
zoom on
% cleanup
evalin('base','clear w theta')
disp('This part of the example illustrates a sequence of 5 plots. Each plot')
disp('shows the simulation results for the next frequency range. Using the')
disp('mouse, zoom in on each signal to observe signal amplitudes.')
close_system(mdlName, 0);
```

```
Starting batch simulations.
This part of the example illustrates a sequence of 5 plots. Each plot
shows the simulation results for the next frequency range. Using the
mouse, zoom in on each signal to observe signal amplitudes.
```



Use MAT-Files to Feed Data to Inport Blocks for Rapid Simulations

This example shows how the code generator RSim -i option lets you use a MAT-file as the input data source for Inport blocks for rapid simulations. The data in such a MAT-file can be presented in the following formats:

- One variable that defines a time/input data matrix of double values.
- One variable that defines a structure that uses a combination of Simulink® data types.
- Multiple variables, each defining a structure that uses a combination of Simulink® data types.

This flexibility lends itself well to applications for which you must run simulations over a range of input data stored in different data files. This example explains how to use this feature.

To quickly run multiple simulations in the Simulink environment, consider using rapid accelerator instead of RSim. See “What Is Acceleration?”.

Prepare Model

Open the model and save a copy to a writable location. Then, open the **Simulink Coder** app. Set model configuration parameter *System target file to `rsim.tlc`.

For more information on doing this graphically and setting up other model configuration parameters relate to the RSim system target file, see “Configure a System Target File” on page 50-2.

```
mdlName = 'rtwdemo_rsim_i';
open_system(mdlName);
cs = getActiveConfigSet(mdlName);
cs.switchTarget('rsim.tlc',[]);
```

Configure Inport Blocks

To use the RSim `-i` option, you must configure each Inport block properly. You can double-click an Inport block to view its properties. By default, Inport blocks inherit their properties from downstream blocks. Before you can import data from external MAT-files, you must set the parameters of each Inport block to match the data in the MAT file. In most cases, these parameters of an Inport block must be set:

- **Interpolate Data**
- **Port Dimensions**
- **Data Type**
- **Signal Type**

For more information on these parameters, click the Help button. This example model includes three Inport blocks. A requirement for the example is for Inport 1 and Inport 2 to interpolate between data, and for Inport 3 to not interpolate. The dimension of the Inport blocks are 2, 1, and 2, respectively. Signals are real. The settings are as follows:

```
for i =1:3
    portName      =['/In', num2str(i)];
    Interp        = get_param(strcat(mdlName,portName),'Interpolate');
    PortDimension = get_param(strcat(mdlName,portName),'PortDimensions');
```

```
    DataType      = get_param(strcat(mdlName,portName),'OutDataTypeStr');
    SignalType    = get_param(strcat(mdlName,portName),'SignalType');
end
```

- For inports `In1` and `In2`, interpolation is selected. For inport `In3`, interpolation is cleared.
- For inports `In1` and `In3`, port dimension is set to 2. For inport `In2`, port dimension is set to 1.
- For inports `In1`, `In2`, and `In3`, data type is `double` and signal type is `real`.

Build Model

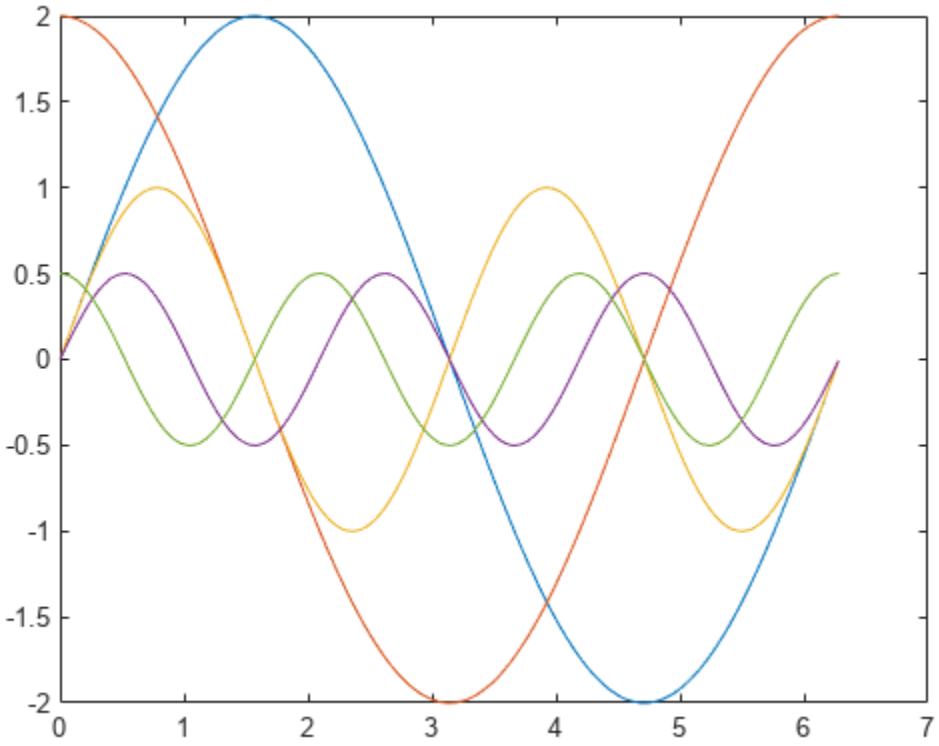
Build an executable for the model. During the build process, a structural checksum is calculated and embedded into the generated executable program. This checksum is used to check that a parameter set passed to the executable program is compatible with the program.

```
evalc('slbuild(mdlName)');
```

Identify Signals for Data File

After you configure the Inport block, prepare the data file based on the Inport blocks. This figure shows the input signals to be used.

```
t=[0:0.01:2*pi]';
s1 = [2*sin(t) 2*cos(t)];
s2 = sin(2*t);
s3 = [0.5*sin(3*t) 0.5*cos(3*t)];
figure;
plot(t, [s1 s2 s3]);
```



This figure displays the input signals.

Prepare MAT-File

Generally, you can create the MAT-file from a workspace variable. The RSim -i option supports three data file formats:

One variable in TU matrix format of doubles

For this format, the first column is the time vector and the remaining columns are input vectors. The number of columns in the TU matrix equals the sum of the dimensions of the root Inport blocks plus 1. This MATLAB® code generates a MAT-file containing one variable var_matrix in TU matrix format. You can use this format only if the input ports in the model have the same data type.

```
t=[0:0.1:2*pi]';
Ina1 = [2*sin(t) 2*cos(t)];
Ina2 = sin(2*t);
Ina3 = [0.5*sin(3*t) 0.5*cos(3*t)];
var_matrix = [t Ina1 Ina2 Ina3];
save rsim_i_matrix.mat var_matrix;
```

MAT-file `rsim_i_matrix.mat` contains one variable `var_matrix` in TU matrix format.

One variable in structure format

For this format, the variable must contain two fields: time and signals. If you set the block parameter **Interpolate data** for one of the Inport blocks, the time field of the variable must not be an empty vector and the width of the signals must equal the total width of the Inport blocks. This code generates a MAT-file that contains one variable `var_matrix` in signal variable structure format. This format is more flexible than the TU matrix format because it can support input ports with different data types.

```
t= [0:0.1:2*pi]';
var_single_struct.time = t;
var_single_struct.signals(1).values(:,1) = 2*sin(t);
var_single_struct.signals(1).values(:,2) = 2*cos(t);
var_single_struct.signals(2).values = sin(2*t);
var_single_struct.signals(3).values(:,1) = 0.5*sin(3*t) ;
var_single_struct.signals(3).values(:,2) = 0.5*cos(3*t) ;
v=[var_single_struct.signals(1).values var_single_struct.signals(2).values ...
    var_single_struct.signals(3).values ];
save rsim_i_single_struct.mat var_single_struct;
```

MAT-file `rsim_i_single_struct.mat` contains one variable `var_single_struct` in `struct` format.

Multiple variables in structure format

For this format, the number of variables equals the number of Inport blocks. Different variables can have different time vectors. This code generates a MAT-file that contains multiple variables, each in structure format. This is the most flexible format because it allows each Inport block to have its own time vector.

```
t= [0:0.1:2*pi]';
Inb1.time = t;
Inb1.signals.values(:,1) = 2*sin(t);
Inb1.signals.values(:,2) = 2*cos(t);
```

```
t= [0:0.2:2*pi]';
Inb2.time = t;
Inb2.signals.values(:,1) = sin(2*t);
t= [0:0.1:2*pi]';
Inb3.time =  t;
Inb3.signals.values(:,1) = 0.5*sin(3*t);
Inb3.signals.values(:,2) = 0.5*cos(3*t);
save rsim_i_multi_struct.mat Inb1;
save rsim_i_multi_struct.mat Inb2 -append;
save rsim_i_multi_struct.mat Inb3 -append;
```

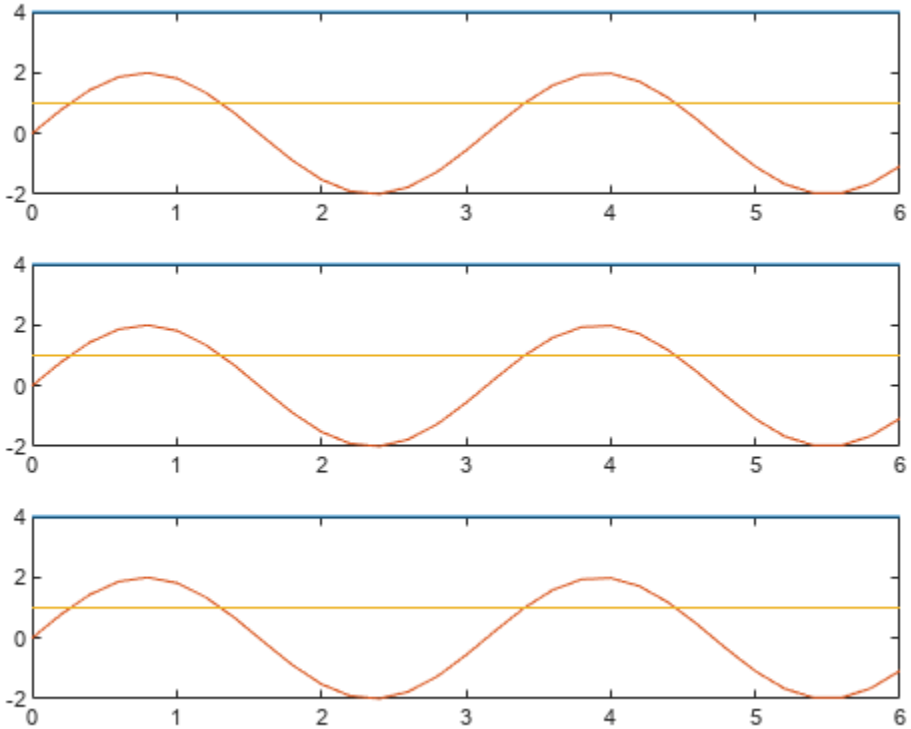
MAT-file `rsim_i_multi_struct.mat` contains three variables `Inb1, Inb2`, and `Inb3` in `struct` format. Use the `save` command with the `-append` option to preserve the order of variables in the generated MAT-file.

Run RSim Simulations and Plot Results

You can use RSim -i options for batch mode simulation. Prepare different MAT-files and run the RSim program executable with them.

This part of the example shows three plots. Each plot shows simulation results for a MAT-file with a different variable format. The model is compiled once.

```
figure
fileName = ({'rsim_i_matrix', 'rsim_i_single_struct', 'rsim_i_multi_struct'});
for i=1:3
    % bang out and run a simulation using new parameter data
    name = fileName(i);
    runstr = ['.', filesep, 'rtwdemo_rsim_i -i ',char(name),'.mat', ' -v'];
    evalc('system(runstr)');
    pause(0.5);
    % load simulation data into MATLAB(R) for plotting.
    load rtwdemo_rsim_i.mat;
    subplot(3,1,i);
    axis([0,6, -5, 5]);
    plot(rt_tout, rt_yout);
    hold on
end
```




```
close_system(mdlName, 0);
```

Use S-Function Target for Model or Subsystem

S-functions are a class of system target file for which the code generator can produce code. An S-function target can encapsulate a subsystem to increase its execution efficiency, facilitate code reuse, and protect its intellectual property.

Note While you can use the S-function target to deploy an application component for reuse while shielding its internal logic from inspection and modification, the preferred solutions for protecting intellectual property in distributed components are:

- The protected model, a referenced model that hides block and line information. For more information, see “Protect Models to Conceal Contents”.
- The shared library system target file, used to generate a shared library for a model or subsystem for use in a system simulation external to Simulink. For more information, see “Package Generated Code as Shared Libraries” on page 69-2.

These solutions also increase execution efficiency and facilitate code reuse.

You can use a generated S-function target with a Generated S-Function block.

The `'S-Function'` value for the `CodeFormat` TLC variable used by the S-function target generates code that conforms to the Simulink C MEX S-function application programming interface (API).

Required Files for S-Function Deployment

There are different files required to deploy a generated S-Function block for simulation versus code generation.

To deploy your generated S-Function block for inclusion in other models *for simulation*, you need only provide the binary MEX-file object that was generated in the current working folder when the S-Function block was created. The required file is:

- *subsys*`_sf.`*mexext*

where *subsys* is the subsystem name and *mexext* is a platform-dependent MEX-file extension (see `mexext`). For example, `SourceSubsys_sf.mexw64`.

To deploy your generated S-Function block for inclusion in other models *for code generation*, provide the files that were generated in the current working folder when the S-Function block was created. The required files are:

- *subsys*`_sf.c` or `.cpp`, where *subsys* is the subsystem name (for example, `SourceSubsys_sf.c`)
- *subsys*`_sf.h`
- *subsys*`_sf.`*mexext*, where *mexext* is a platform-dependent MEX-file extension (see `mexext`)
- Subfolder *subsys*`_sfcn_rtw` and its contents

The `rtwsfcn.tlc` system target file is provided for use with the S-function target.

The generated S-function code uses **Configuration Parameters > Hardware Implementation** parameter values that match the host system on which the function was built. When you use the S-function in a model for code generation, make sure that these parameter values for the model match the parameter values of the S-function.

Sample Time Propagation in Generated S-Functions

A generated S-Function block can inherit its sample time from the model in which it is placed if certain criteria are met. Conditions that govern sample time propagation for both Model blocks and generated S-Function blocks are described in “Referenced Model Sample Times” and “S-Functions That Specify Sample Time Inheritance Rules” on page 13-77.

To generate an S-Function block that meets the criteria for inheriting sample time, you must constrain the solver for the model from which the S-Function block is generated. Set model configuration parameter **Type** to `Fixed-step` and **Periodic sample time constraint** to `Ensure sample time independent`. If the model is unable to inherit sample times, this setting causes the Simulink software to display an error message when building the model. For more information about this option, see Periodic sample time constraint.

Solver Type for Top Models with Generated S-Functions

The table shows the possible combinations of top model solver types as these types relate to whether the model has discrete or continuous sample times and solver types for generated S-functions.

Top Model Solver Options and Sample Times

	Model Configuration Parameters: Top-level model configuration	
Sample Times	**Solver Options, Type: Variable-step**	**Solver Options, Type: Fixed-step**
Discrete	Generated S-function requires a variable-step solver	Generated S-function can have a variable-step solver or a fixed-step solver
Continuous	Generated S-function requires a variable-step solver	Generated S-function requires a fixed-step solver

S-functions generated from a subsystem have parameters that are hardcoded into the block. Simulink calculates parameters such as sample time when it generates the block, not during simulation run time. It is important to verify whether the generated S-Function block works as expected in the destination model.

Tunable Parameters in Generated S-Functions

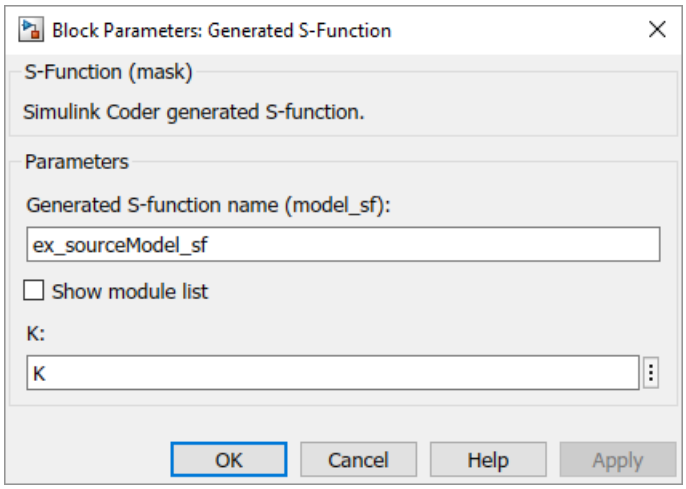
To use tunable parameters in generated S-functions, use model configuration parameters to declare desired block parameters tunable. See “Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box”.

Block parameters that you declare as tunable with the `auto` storage class in the source model become tunable parameters of the generated S-function. These parameters do not become part of a generated *`model`*`_P` (formerly `rtP`) parameter data structure, as they would in code generated from

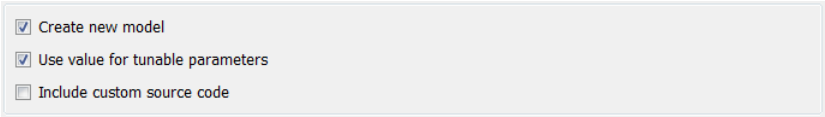
other system target files. Instead, the generated code accesses these parameters by using MEX API calls such as `mxGetPr` or `mxGetData`. Your code should access these parameters in the same way.

For more information on MEX API calls, see “About C MEX S-Functions” and “Integrate MATLAB with External Programming Languages and Systems”.

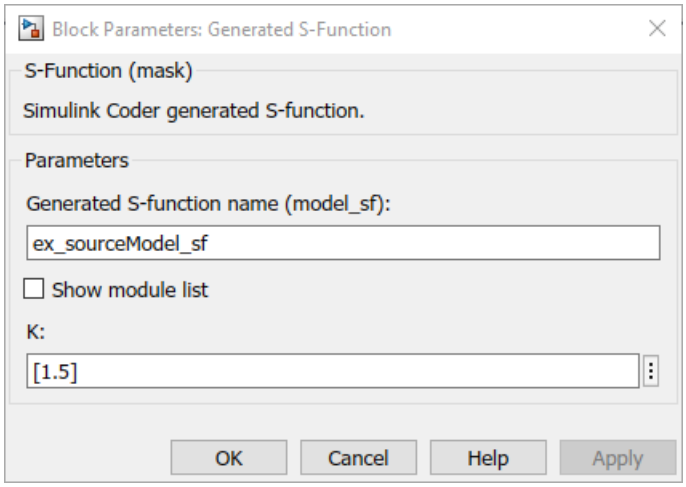
S-Function blocks created by using the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



You can choose to display the value of the parameter rather than its variable name by selecting model configuration parameter **Use value for tunable parameters**.



When select this parameter, the value of the variable (at code generation time) is displayed in the edit field.



Macro Parameters

Suppose that you apply a storage class such as `Define` to a `Simulink.Parameter` object so that the parameter appears as a macro in the generated code. If you use the parameter object inside a subsystem from which you generate an ERT S-function, you cannot select the parameter object as a tunable parameter. Instead, the S-function code generator applies the custom storage class to the parameter object. This generation of macros in the S-function code enables you to generate S-functions from subsystems that contain variant elements, such as Variant Subsystem blocks, that you configure to produce preprocessor conditionals in the generated code. You cannot change the value of the parameter during simulation of the S-function.

To select the parameter object as a tunable parameter, apply a different storage class or create your own storage class. Storage classes that treat parameters as macros include `Define`, `ImportedDefine`, `CompilerFlag`, and storage classes that you create by setting **Data initialization** to `Macro` in the Custom Storage Class Designer. If you use a non-macro storage class, you cannot use the parameter object as a variant control variable and generate preprocessor conditionals.

If you apply a storage class that treats the parameter object as an imported macro, before you generate the ERT S-function, provide the macro definition. For example, suppose that you apply the storage class `ImportedDefine` to a `Simulink.Parameter` object and use the parameter object as a variant control variable in the subsystem. If you set the custom attribute `HeaderFile` to `'myHdr.h'`, when you generate the S-function, place the custom header file `myHdr.h` in the current folder. The generated S-function uses the macro value from your header file instead of the value from the `Value` property of the parameter object.

To use a macro that you define through a compiler option, for example by applying the storage class `CompilerFlag`, use the model configuration parameter **Code Generation** > **Custom Code** > **Code information** > **Defines** to specify the compiler option. For more information, see Code Generation Pane: Custom Code: Additional Build Information: Defines.

Checksums and the S-Function Target

The code generator creates a checksum for a model and uses the checksum during the build process for code reuse, model reference, and external mode features.

The code generator calculates a model checksum by

1. Calculating a checksum for each subsystem in the model. A subsystem's checksum is the combination of properties (data type, complexity, sample time, port dimensions, and so forth) of the subsystem blocks.
2. Combining the subsystem checksums and other model-level information.

An S-function can add additional information, not captured during the block property analysis, to a checksum by calling the function `ssSetChecksumVal`. For the S-Function target, the value that gets added to the checksum is the checksum of the model or subsystem from which the S-function is generated.

The code generator applies the subsystem and model checksums as follows:

- Code reuse — If two subsystems in a model have the same checksum, the code generator produces code for one function only.

- Model reference — If the current model checksum matches the checksum when the model was built, the build process does not rebuild referenced models.
- External mode — If the current model checksum does not match the checksum of the code that is running on the target hardware, the build process generates an error.

Generated S-Function Compatibility

When you build a MEX S-function from your model, the code generator builds a level 2 noninlined S-function. Cross-release usage limitations on the generated code and binary MEX file (for example, `*.mexw64`) include:

- S-function target generated code from previous MATLAB release software is not compatible with newer releases. Do not recompile the generated code from a previous release with newer MATLAB release software. Use the same MATLAB release software to generate code for the S-function target and compile the code into a MEX file.
- You can use binary S-function MEX files generated from previous MATLAB release software with the same or newer releases with the same compatibility considerations as handwritten S-functions. For more information, see “S-Function Compatibility”.
- The code generator can generate code and build an executable from a model that contains generated S-functions. This support requires that the S-functions are built with the same MATLAB release software that builds the model. It is not possible to incorporate a generated S-function MEX file from previous MATLAB release software into a model and build the model with newer releases.

S-Function Target Limitations

- “Inport and Outport Blocks with Continuous Sample Time” on page 68-45
- “Tunable Variables in Expressions” on page 68-45
- “Parameter Tuning” on page 68-45
- “Run-Time Parameters and S-Function Compatibility Diagnostics” on page 68-45
- “Goto and From Blocks” on page 68-45
- “Building and Updating S-Functions” on page 68-46
- “Unsupported Blocks” on page 68-47
- “Model Operating Point not Supported for Code Generation” on page 68-47
- “Nesting S-Functions” on page 68-47
- “User-Defined Data Types” on page 68-47
- “Right-Click Generation of an S-Function Target” on page 68-48
- “Bus Input and Output” on page 68-48
- “Subsystems with Function-Call Input and Output Signals” on page 68-48
- “Function-Call Subsystem Modeling Pattern” on page 68-48
- “Data Store Access” on page 68-49
- “Inport or Outport Block Parameters Through Subsystem Mask” on page 68-49
- “MEX S-Function Wrappers” on page 68-49

Inport and Outport Blocks with Continuous Sample Time

The right-click build method for generating an S-function does not preserve continuous sample time for Inport and Outport blocks. Because the sample time is not preserved, simulation of the generated S-Function block can be affected.

Tunable Variables in Expressions

Certain limitations apply to the use of tunable variables in expressions. When the code generator encounters an unsupported expression while producing code, a warning appears and the equivalent numeric value is generated in the code. For a list of the limitations, see “Tunable Expression Limitations” on page 36-122.

Parameter Tuning

The S-Function block does not support tuning of tunable parameters with:

- Complex values.
- Values or data types that are transformed to a constant (by setting the model configuration parameter **Optimization** > **Default parameter behavior** to `Inlined`).
- Data types that are not built-in.
- Floating-point data types that are not equivalent to a built-in type.

Run-Time Parameters and S-Function Compatibility Diagnostics

If you set model configuration parameter **S-function upgrades needed** to `warning` or `error`, the code generator instructs you to upgrade S-functions generated from subsystems. The S-function system target file does not register run-time parameters. Run-time parameters are only supported for inlined S-functions, and the generated S-function supports features that prevent it from being inlined, for example, it can call or contain other noninlined S-functions.

To work around this limitation, set parameter **S-function upgrades needed** to `none`.

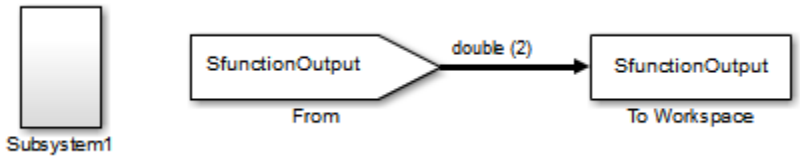
Goto and From Blocks

When using the S-function system target file, the code generator restricts input and output to correspond to the root model Inport and Outport blocks (or the Inport and Outport blocks of the Subsystem block from which the S-function target was generated). The code generator does not produce code for Goto or From blocks.

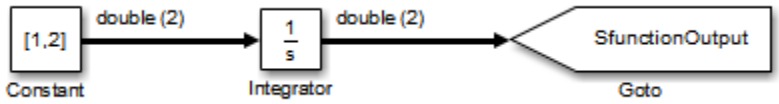
To work around this restriction, create your model and subsystem with the required Inport and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add Goto and From blocks.

Example Before Work Around

- Root model that includes a From block and subsystem, `Subsystem1`



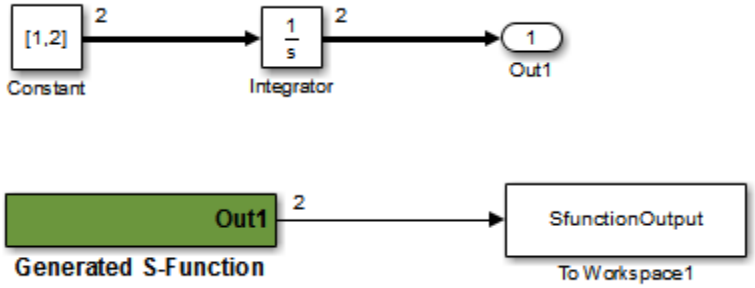
- `Subsystem1` that includes a Goto block that has global visibility and passes its input to the From block in the root model



- `Subsystem1` replaced with an S-function generated with the S-Function system target file — a warning results when you run the model because the generated S-function does not implement the Goto block

Example After Work Around

An Outport block replaces the GoTo block in `Subsystem1`. When you plug the generated S-function into the root model, its output connects directly to the To Workspace block.



Building and Updating S-Functions

The following limitations apply to building and updating S-functions using the S-function system target file:

- You cannot build models that contain Model blocks using the S-function system target file. This also means that you cannot build a subsystem by using the right-click context menu if the subsystem contains Model blocks. This restriction applies only to S-functions generated using the S-function target, not to ERT S-functions.
- You can build a model with:
  - A toolchain only if the toolchain supports MEX-file generation.
  - A template makefile only if the template makefile is associated with a toolchain on page 59-26 that supports MEX-file generation.
- If you modify the model that generated an S-Function block, the build process does not automatically rebuild models containing the generated S-Function block. This is in contrast to the

practice of automatically rebuilding models referenced by Model blocks when they are modified (depending on the Model Reference **Rebuild** configuration setting).

- Handwritten S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, see “Exception Free Code”.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- Interpreted MATLAB Function block
- S-Function blocks containing any of the following:
  - MATLAB language S-functions (unless you supply a TLC file for C code generation)
  - Fortran S-functions (unless you supply a TLC file for C code generation)
  - C/C++ MEX S-functions that call into the MATLAB environment
- Simulink Function block
- Function Caller block
- Scope block
- To Workspace block

The S-function format does not support blocks from the `embeddedtargetslib` block library.

Model Operating Point not Supported for Code Generation

You can write C-MEX and Level-2 MATLAB S-functions that leverage the model operating point, which is used to save and restore the simulation state, as described in “S-Function Compliance with the ModelOperatingPoint”. The model operating point is not supported for code generation, including with the S-function system target file.

When you generate code for an S-function that specifies default operating point compliance or no operating point support, the generated code does not include the model operating point functionality. You cannot generate code for an S-function that implements custom operating point functionality.

Nesting S-Functions

The following limitations apply to nesting a generated S-Function block in a model or subsystem from which you generate another S-function:

- The software does not support nonvirtual bus input and output for a nested S-function.
- Avoid nesting an S-function in a model or subsystem having the same name as the S-function (possibly several levels apart). In such situations, the S-function can be called recursively. The software currently does not detect such loops in S-function dependency, which can result in aborting or hanging your MATLAB session. To prevent this from happening, name the subsystem or model to be generated as an S-function target uniquely to avoid duplicating existing MEX filenames on the MATLAB path.

User-Defined Data Types

The S-function system target file does not support the `HeaderFile` property that can be specified on user-defined data types, including those based on `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType` objects. If a user-defined data type in your model uses the `HeaderFile`

property to specify an associated header file, code generation with the S-function system target file disregards the value and does not generate a corresponding include statement.

Right-Click Generation of an S-Function Target

If you generate an S-function target by right-clicking a Function-Call Subsystem block, the original subsystem and the generated S-function might not be consistent. An inconsistency occurs when the **States when enabling** parameter of the Trigger Port block inside the Function-Call Subsystem block is set to **inherit**. You must set the **States when enabling** parameter to **reset** or **held**, otherwise Simulink reports an error.

Bus Input and Output

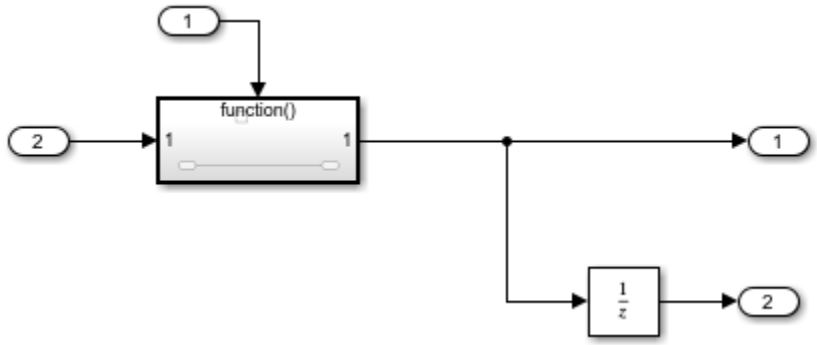
If an S-function generated using the S-function target has bus input or output, the generated bus data structures might include padding to align fields of the bus elements with the Simulink representation used during simulation. However, if you insert the S-function in a model and generate code using a model target such as `grt.tlc`, the bus structure alignment generated for the model build might be incompatible with the padding generated for the S-function and might impact the numerical results of code execution. To make the structure alignment consistent between model simulation and execution of the model code, for each `Simulink.Bus` object, you can modify the `HeaderFile` property to remove the unpadded bus structure header file. This will cause the bus typedefs generated for the S-function to be reused in the model code.

Subsystems with Function-Call Input and Output Signals

The S-function target does not support creating an S-Function block from a subsystem that has a function-call trigger input or a function-call output.

Function-Call Subsystem Modeling Pattern

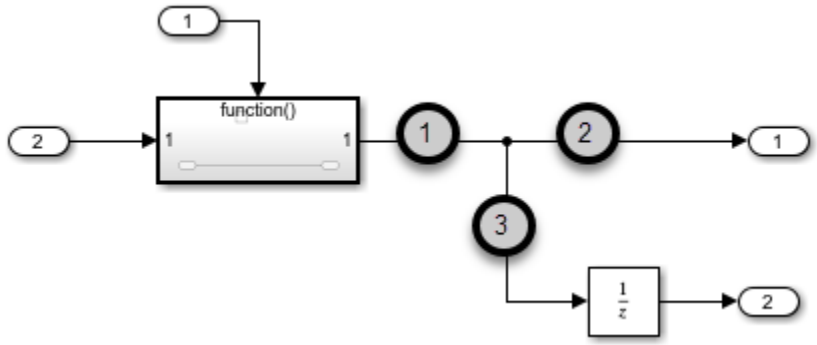
When you add a generated S-Function block to a model and then simulate that model, MATLAB might crash if the generated S-function uses this function-call subsystem modeling pattern.



The function-call subsystem connects directly to an Outport block. The signal line that connects the subsystem to the Outport block branches to a block that has an `Update` function, that is, a block that updates at each major time step, such as Unit Delay and Memory. The branch can be inside or outside of the subsystem.

In the model that includes the function-call subsystem, a Signal Conversion block in one of these locations can work around the problem with the generated S-function:

1. Between the subsystem block and the branch
2. Between the branch and the root Outport block
3. Between the branch and the block that has the `Update` function



The Signal Conversion block in the workaround has these parameter settings:

- **Output** set to `Signal copy`
- **Exclude this bock from 'Block reduction' optimization** selected

Data Store Access

When an S-Function block in your model accesses a data store during simulation, Simulink disables data store diagnostics.

- If you created the S-Function block from a model, the diagnostic is disabled for global data stores as well.
- If you created the S-Function block from a subsystem, the diagnostic is disabled for the following data stores:
  - Global data stores
  - Data stores placed outside the subsystem, but accessed by Data Store Read or Data Store Write blocks.

Inport or Outport Block Parameters Through Subsystem Mask

S-functions generated from a subsystem with Inport or Outport block parameters specified by subsystem mask variables produce an error when you try to run a simulation that uses the S-Function block.

```
Invalid setting in 'testSystem/Subsystem/__OutputSSForSFun__/Out2'
for parameter 'PortDimensions'
...
```

MEX S-Function Wrappers

Use a MEX S-function wrapper only in the MATLAB version in which the wrapper is created.

See Also

More About

- “Protect Models to Conceal Contents”
- “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File”
- “C/C++ S-Function Basics”
- “S-Functions and Code Generation” on page 13-2
- “Import Calls to External Code into Generated Code with Legacy Code Tool” on page 13-6
- “Use a Bus with S-Function Builder to Create an S-Function”

Desktops in Embedded Coder

Package Generated Code as Shared Libraries

If you have an Embedded Coder license, you can package generated source code from a model component for easy distribution and shared use by building the code as a shared library—Windows dynamic link library (`.dll`), UNIX shared object (`.so`), or Macintosh OS X dynamic library (`.dylib`). You or others can integrate the shared library into an application that runs on a Windows, UNIX, or Macintosh OS X development computer. The generated `.dll`, `.so`, or `.dylib` file is shareable among different applications and upgradeable without having to recompile the applications that use it.

About Generated Shared Libraries

You build a shared library by configuring the code generator to use the system target file `ert_shrlib.tlc`. Code generation for that system target file exports:

- Variables and signals of type `ExportedGlobal` as data
- Real-time model structure (*`model`*`_M`) as data
- Functions essential to executing your model code

To view a list of symbols contained in a generated shared library:

- On Windows, use the Dependency Walker utility, downloadable from https://www.dependencywalker.com
- On UNIX, use `nm -D` *`model`*`.so`
- On Macintosh OS X , use `nm -g` *`model`*`.dylib`

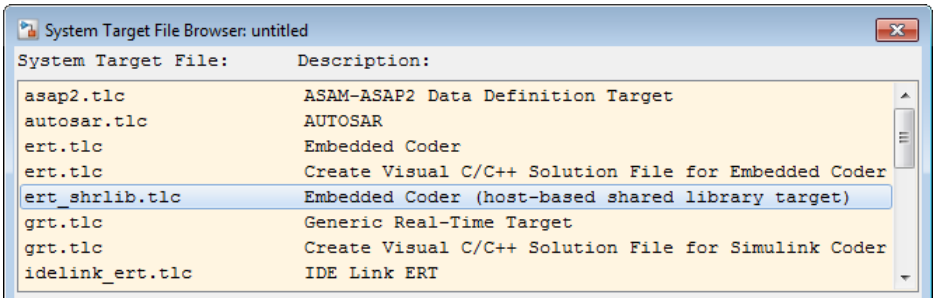
To generate and use a shared library:

1. Generate a shared library version of your model code
2. Create application code to load and use your shared library file

Generate Shared Library Version of Model Code

To generate a shared library version of your model code:

1. Open your model and configure it to use the `ert_shrlib.tlc` system target file.



   Selecting the `ert_shrlib.tlc` system target file causes the build process to generate a shared library version of your model code into your current working folder. The selection does not change the code that the code generator produces for your model.

2. Build the model.
3. After the build completes, examine the generated code in the model subfolder and examine the `.dll`, `.so`, or `.dylib` file in your current folder.

Create Application Code to Use Shared Library

To illustrate how application code can load a shared library file and access its functions and data, MathWorks provides the model `rtwdemo_shrlib`.

Note Navigate to a writable working folder before running the `rtwdemo_shrlib` script.

In the model, click the blue button to run a script. The script:

1. Builds a shared library file from the model (for example, `rtwdemo_shrlib_win64.dll` on 64-bit Windows).
2. Compiles and links an example application, `rtwdemo_shrlib_app`, that loads and uses the shared library file.
3. Executes the example application.

Tip Explicit linking is preferred for portability. However, on Windows systems, the `ert_shrlib` system target file generates and retains the `.lib` file to support implicit linking.

To use implicit linking, the generated header file needs a small modification for you to use it with the generated C file. For example, if you are using Visual C++, declare `__declspec(dllimport)` in front of data to be imported implicitly from the shared library file.

The model uses the following example application files, which are located in the folder *`matlabroot`*`/toolbox/rtw/rtwdemos/shrlib_demo` (open).

File	Description
`rtwdemo_shrlib_app.h`	Example application header file
`rtwdemo_shrlib_app.c`	Example application that loads and uses the shared library file generated for the model
`run_rtwdemo_shrlib_app.m`	Script to compile, link, and execute the example application

You can view each of these files by clicking white buttons in the model window. Additionally, running the script places the relevant source and generated code files in your current folder. The files can be used as templates for writing application code for your own ERT shared library files.

The following sections present key excerpts of the example application files.

Example Application Header File

The example application header file `rtwdemo_shrlib_app.h` contains type declarations for the model's external input and output.

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_
```

```
typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /*_APP_MAIN_HEADER_*/
```

Example Application C Code

The example application `rtwdemo_shrlib_app.c` includes the following code for dynamically loading the shared library file. Notice that, depending on platform, the code invokes Windows or UNIX library commands.

```
#if (defined(_WIN32)||defined(_WIN64)) /* WINDOWS */
#include <windows.h>
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

#else /* UNIX */
#include <dlfcn.h>
#define GETSYMBOLADDR dlsym
#define LOADLIB dlopen
#define CLOSELIB dlclose

#endif

int main()
{
    void* handleLib;
...
#if defined(_WIN64)
    handleLib = LOADLIB("./rtwdemo_shrlib_win64.dll");
#else
#if defined(_WIN32)
    handleLib = LOADLIB("./rtwdemo_shrlib_win32.dll");
#else /* UNIX */
    handleLib = LOADLIB("./rtwdemo_shrlib.so", RTLD_LAZY);
#endif
#endif
...
    return(CLOSELIB(handleLib));
}
```

The following code excerpt shows how the C application accesses the model's exported data and functions. Notice the hooks for adding user-defined initialization, step, and termination code.

```
    int32_T i;
 ...
    void (*mdl_initialize)(boolean_T);
    void (*mdl_step)(void);
    void (*mdl_terminate)(void);

    ExternalInputs_rtwdemo_shrlib (*mdl_Uptr);
    ExternalOutputs_rtwdemo_shrlib (*mdl_Yptr);

    uint8_T (*sum_outptr);
...
#if (defined(LCCDLL)||defined(BORLANDCDLL))
    /* Exported symbols contain leading underscores when DLL is linked with
       LCC or BORLANDC */
    mdl_initialize =(void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
                      "_rtwdemo_shrlib_initialize");
    mdl_step       =(void(*)(void))GETSYMBOLADDR(handleLib ,
                      "_rtwdemo_shrlib_step");
    mdl_terminate  =(void(*)(void))GETSYMBOLADDR(handleLib ,
```

```
                    "_rtwdemo_shrlib_terminate");
    mdl_Uptr       =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                    "_rtwdemo_shrlib_U");
    mdl_Yptr       =(ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                    "_rtwdemo_shrlib_Y");
    sum_outptr     =(uint8_T*)GETSYMBOLADDR(handleLib , "_sum_out");
#else
    mdl_initialize =(void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
                    "rtwdemo_shrlib_initialize");
    mdl_step       =(void(*)(void))GETSYMBOLADDR(handleLib ,
                    "rtwdemo_shrlib_step");
    mdl_terminate  =(void(*)(void))GETSYMBOLADDR(handleLib ,
                    "rtwdemo_shrlib_terminate");
    mdl_Uptr       =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                    "rtwdemo_shrlib_U");
    mdl_Yptr       =(ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                    "rtwdemo_shrlib_Y");
    sum_outptr     =(uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

    if ((mdl_initialize && mdl_step && mdl_terminate && mdl_Uptr && mdl_Yptr &&
         sum_outptr)) {
        /* === user application initialization function === */
        mdl_initialize(1);
        /* insert other user defined application initialization code here */

        /* === user application step function === */
        for(i=0;i<=12;i++){
            mdl_Uptr->Input = i;
            mdl_step();
            printf("Counter out(sum_out): %d\tAmplifier in(Input): %d\tout(Output): %d\n",
                   *sum_outptr, i, mdl_Yptr->Output);
            /* insert other user defined application step function code here */
        }

        /* === user application terminate function === */
        mdl_terminate();
        /* insert other user defined application termination code here */
    }
    else {
        printf("Cannot locate the specified reference(s) in the shared library.\n");
        return(-1);
    }
```

Example Application Script

The application script `run_rtwdemo_shrlib_app` loads and rebuilds the model, and then compiles, links, and executes the model's shared library target file. You can view the script source file by opening `rtwdemo_shrlib` and clicking a white button to view source code. The script constructs platform-dependent command character vectors for compilation, linking, and execution that may apply to your development environment. To run the script, click the blue button.

Note To run the `run_rtwdemo_shrlib_app` script without first opening the `rtwdemo_shrlib` model, navigate to a writable working folder and issue the following MATLAB command:

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','shrlib_demo'))
```

Note It is invalid to invoke the terminate function twice in a row. The terminate function clears pointers and sets them to NULL. Invoking the function a second time dereferences null pointers and results in a program failure.

Shared Library Limitations

The following limitations apply to building shared libraries:

- Code generation for the `ert_shrlib.tlc` system target file exports the following as data:
  - Variables and signals of type `ExportedGlobal`
  - Real-time model structure (*`model`*`_M`)
- For a model that contains a function-call subsystem, code generation for the `ert_shrlib.tlc` system target file exports to the shared library only symbols associated with the initialize and terminate entry-point functions.
- Code generation for the `ert_shrlib.tlc` system target file supports the C language only (not C `++`). When you select `ert_shrlib.tlc`, model configuration parameter **Language** is greyed out.
- To reconstruct a model simulation using a generated shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing needs to be consistent so that you can compare the simulation and integration results. Additional simulation considerations apply if generating a shared library from a model that enables model configuration parameters **Support: continuous time** and **Single output/update function**. For more information, see Single output/update function dependencies.

See Also

More About

- “Map Model Elements to Code Interfaces” on page 1-29
- “Configure a System Target File” on page 50-2
- “Model Protection”

Real-Time Systems in Simulink Coder

- “Deploy Algorithm Model for Real-Time Rapid Prototyping” on page 70-2
- “Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation” on page 70-4
- “Untrusted Custom Code, Custom Targets, and Callbacks” on page 70-6

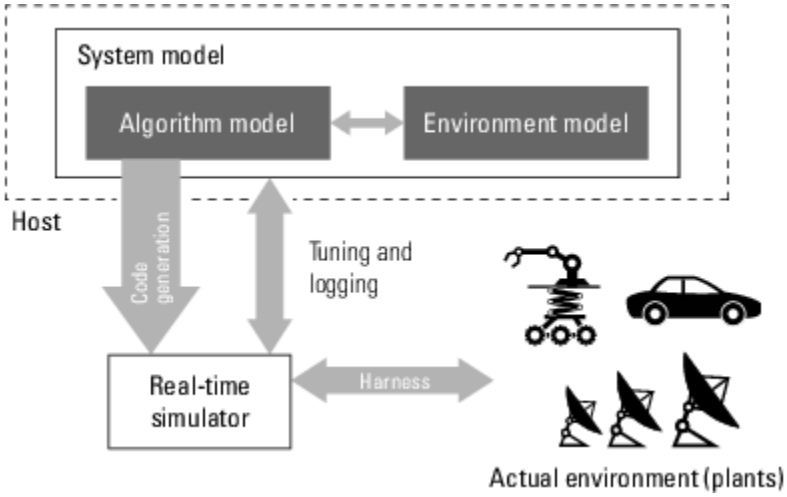
Deploy Algorithm Model for Real-Time Rapid Prototyping

Use the code generator to deploy algorithm models for real-time rapid prototyping.

About Real-Time Rapid Prototyping

Real-time rapid prototyping requires the use of a real-time simulator, potentially connected to system hardware (for example, physical plant or vehicle) being controlled. You generate, deploy, and tune code as it runs on the real-time simulator or embedded microprocessor. This design step is crucial for verifying whether a component can adequately control the system, and allows you to assess, interact with, and optimize code.

The following figure shows a typical approach for real-time rapid prototyping.



Goals of Real-Time Rapid Prototyping

Assuming that you have documented functional requirements, refined concept models, system hardware for the physical plant or vehicle being controlled, and access to target products you intend to use (for example, for example, the Simulink Real-Time or Simulink Desktop Real-Time product), you can use real-time prototyping to:

- Refine component and environment model designs by rapidly iterating between algorithm design and prototyping
- Validate whether a component can adequately control the physical system in real time
- Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design
- Test hardware

Refine Code With Real-Time Rapid Prototyping

To perform real-time rapid prototyping:

1. Create or acquire a real-time system that runs in real time on rapid prototyping hardware. The Simulink Real-Time product facilitates real-time rapid prototyping. This product provides a real-

time operating system that makes PCs run in real time. It also provides device driver blocks for numerous hardware I/O cards. You can then create a rapid prototyping system using inexpensive commercial-off-the-shelf (COTS) hardware. In addition, third-party vendors offer products based on the Simulink Real-Time product or other code generation technology that you can integrate into a development environment.

2. Use provided system target files to generate code that you can deploy onto a real-time simulator. See the following information.

Engineering Tasks	Related Product Information	Examples
Generate code for real-time rapid prototyping	"Compare System Target File Support Across Products" "Event-Based Scheduling" Embedded Coder "Support for Standards and Guidelines" on page 23-2	`rtwdemo_counter` `rtwdemo_counter_msvc`
Generate code for rapid prototyping in hard real time, using PCs	Simulink Real-Time "Simulink Real-Time Options Pane" (Simulink Real-Time)	`slrt_ex_osc` (and others)
Generate code for rapid prototyping in soft real time, using PCs	Simulink Desktop Real-Time "Simulink Desktop Real-Time Pane" (Simulink Desktop Real-Time)	`sldrtex_vdp` (and others)

3. Monitor signals, tune parameters, and log data.

See Also

More About

- "Access Signal, State, and Parameter Data During Execution" on page 36-17
- "Real-Time Simulation and Testing" (Simulink Real-Time)

Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation

In this section...

"About Hardware-In-the-Loop Simulation" on page 70-4
"Set Up and Run HIL Simulations" on page 70-4

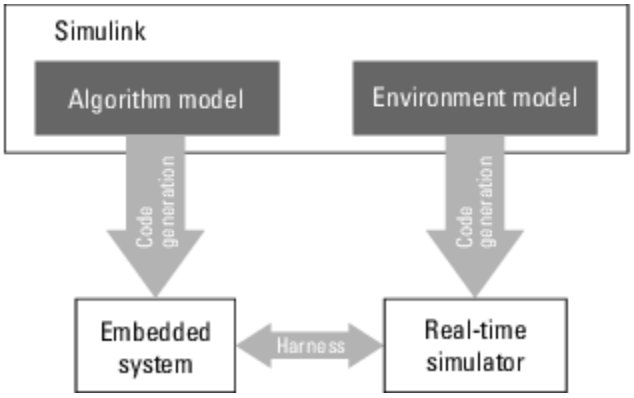
About Hardware-In-the-Loop Simulation

Hardware-in-the-loop (HIL) simulation tests and verifies an embedded system or control unit in the context of a software test platform. Examples of test platforms include real-time target systems and instruction set simulators (IISs). You use Simulink software to develop and verify a model that represents the test environment. Using the code generator, you produce, build, and download an executable program for the model to the HIL simulation platform. After you set up the environment, you can run the executable to validate the embedded system or control unit in real time.

During HIL simulation, you gradually replace parts of a system environment with hardware components as you refine and fabricate the components. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

The code that you build for the system simulator provides real-time system capabilities. For example, the code can include VxWorks from Wind River or another real-time operating system (RTOS).

The following figure shows a typical HIL setup.



The HIL platform available from MathWorks is the Simulink Real-Time product. Several third-party products are also available for use as HIL platforms. The Simulink Real-Time product offers hard real-time performance for PCs with Intel or AMD® 32-bit processors functioning as your real-time target. The Simulink Real-Time product enables you to add I/O interface blocks to your models and automatically generate code with code generation technology. The Simulink Real-Time product can download the code to a second PC running the Simulink Real-Time real-time kernel. System integrator solutions that are based on Simulink Real-Time are also available.

Set Up and Run HIL Simulations

To set up and run HIL simulations iterate through the following steps:

1. Develop a model that represents the environment or system under development.

   For more information, see “Compare System Target File Support Across Products”.
2. Generate an executable for the environment model.
3. Download the executable for the environment model to the HIL simulation platform.
4. Replace software representing a system component with corresponding hardware.
5. Test the hardware in the context of the HIL system.
6. Repeat steps 4 and 5 until you can simulate the system after including components that require testing.

See Also

More About

- “Access Signal, State, and Parameter Data During Execution” on page 36-17
- “Real-Time Simulation and Testing” (Simulink Real-Time)

Untrusted Custom Code, Custom Targets, and Callbacks

You can use custom code in:

- Simulations on your development computer.
- Applications that you deploy on external hardware.

Untrusted custom code, custom targets, or callbacks have risks. For example, unintended modification of your design, unauthorized acquisition of sensitive code and data by third parties, and unapproved destruction of data.

Best practices for exchanging sensitive code and data with a supplier or customer are:

- For code packaging, use a secure container that supports encryption such as an Open Packaging Conventions (OPC) container.
- For file transfer, use a secure transfer protocol such as HTTPS or SFTP.

Real-Time and Embedded Systems in Embedded Coder

- “Deploy Applications to Target Hardware” on page 71-2
- “Deploy Component Algorithm as Component Model Library by Using CMake” on page 71-24
- “Generate Main Program for Deployment to Bare-Metal Target Hardware” on page 71-32
- “Deploy Generated Component Software to Application Target Platforms” on page 71-34

Deploy Applications to Target Hardware

By default, the Embedded Coder software generates application code that does not require an external real-time executive or operating system. Generated application code requires minimal modification to be adapted to the target hardware. The architecture of the application code supports execution of models with single or multiple sample rates.

In this section...

"Generate Code for an Application" on page 71-2
"Generated Application Code" on page 71-2
"Main Program" on page 71-3
"rt_OneStep and Scheduling Considerations" on page 71-4
"Static Main Program Module" on page 71-8
"Rate Grouping Compliance and Compatibility Issues" on page 71-12
"Generate Code That Dereferences Data from a Literal Memory Address" on page 71-15

Generate Code for an Application

To generate code for an application:

1. Select the model configuration parameter **Generate an example main program**. This enables the **Target operating system** menu.
2. From the **Target operating system** menu, select `BareBoardExample`.
3. Generate code.

Different code is generated for multirate models depending on these factors:

- Whether the model executes in single-tasking or multitasking mode.
- Whether or not reusable code is being generated.

These factors can impact the scheduling algorithms used in generated code, and in some cases impact the API for the model entry-point functions. The following sections discuss these variants.

Generated Application Code

The core of generated application code is the main loop. On each iteration, the main loop executes a background or null task and checks for a termination condition.

The main loop is periodically interrupted by a timer. The function `rt_OneStep` is installed as a timer interrupt service routine (ISR) or is called from a timer ISR at each clock step.

The execution driver, `rt_OneStep`, sequences calls to *`model`*`_step` functions. The operation of `rt_OneStep` differs depending on whether the model is single-rate or multirate. For a single-rate model, `rt_OneStep` simply calls the *`model`*`_step` function. For a multirate model, `rt_OneStep` prioritizes and schedules execution of blocks according to the rates at which they run.

Main Program

- "Overview of Operation" on page 71-3
- "Guidelines for Modifying Main Program" on page 71-3

Overview of Operation

This pseudocode shows the execution of a generated example main program.

```
main()
{
  Initialization (including installation of rt_OneStep as an
    interrupt service routine for a real-time clock)
  Initialize and start timer hardware
  Enable interrupts
  While(not Error) and (time < final time)
    Background task
  EndWhile
  Disable interrupts (Disable rt_OneStep from executing)
  Complete any background tasks
  Shutdown
}
```

The pseudocode is a design for a harness program to drive code generated for your model. The main program implements this design partially. You must modify it according to target environment software specifications.

Guidelines for Modifying Main Program

Make these changes in the production version of the main program module that implements your harness program:

1. Call `model_initialize`.
2. Initialize target-specific data structures and hardware, such as ADCs or DACs.
3. Install `rt_OneStep` as a timer ISR.
4. Initialize timer hardware.
5. Enable timer interrupts and start the timer.

   `rtModel` is not in a valid state until the main program calls `model_initialize`. The main program should not start servicing timer interrupts until after calling `model_initialize`.
6. Optionally, insert background task calls in the main loop.
7. On termination of the main loop (if applicable):

   - Disable timer interrupts.
   - Perform target-specific cleanup such as zeroing DACs.
   - Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions, such as timer interrupt overruns.

     You can use the macros `rtmGetErrorStatus` and `rtmSetErrorStatus` to detect and signal errors.

rt_OneStep and Scheduling Considerations

- “Overview of Operation” on page 71-4
- “Single-Rate Single-Tasking Operation” on page 71-4
- “Multirate Multitasking Operation” on page 71-5
- “Multirate Single-Tasking Operation” on page 71-7
- “Guidelines for Modifying rt_OneStep” on page 71-7

Overview of Operation

The operation of `rt_OneStep` depends upon

- Whether the model is single-rate or multirate. In a single-rate model, the sample times of blocks in the model and the model fixed step size are the same. A model for which the sample times and step size do not meet these conditions is a multirate model.
- The model solver mode (`SingleTasking` versus `MultiTasking`)

Permitted Solver Modes for Embedded Real-Time System Target Files summarizes the valid solver modes for single-rate and multirate models. For a single-rate model, only `SingleTasking` solver mode is allowed.

Permitted Solver Modes for Embedded Real-Time System Target Files

Mode	Single-Rate	Multirate
`SingleTasking`	Allowed	Allowed
`MultiTasking`	Disallowed	Allowed
`Auto`	Allowed (defaults to `SingleTasking`)	Allowed (defaults to `MultiTasking`)

The generated code for `rt_OneStep` (and associated timing data structures and support functions) is tailored to the number of rates in the model and to the solver mode.

Single-Rate Single-Tasking Operation

The only valid solver mode for a single-rate model is `SingleTasking`. Such models run in “single-rate” operation.

This pseudocode shows the design of `rt_OneStep` in a single-rate program.

```
rt_OneStep()
{
  Check for interrupt overflow or other error
  Enable "rt_OneStep" (timer) interrupt
  Model_Step()  -- Time step combines output,logging,update
}
```

For the single-rate case, the generated *`model`*`_step` function is

```
void model_step(void)
```

Single-rate `rt_OneStep` executes *`model`*`_step` within a single clock period. To enforce this timing constraint, `rt_OneStep` maintains and checks a timer overrun flag. On entry, the main program disables timer interrupts until after checking the overrun flag and for other error conditions. If the overrun flag is clear, `rt_OneStep` sets the flag, and proceeds with timer interrupts enabled.

The main program clears the overrun flag if *`model`*`_step` returns without being reinterrupted by the timer. If `rt_OneStep` is reinterrupted by the timer before completing *`model`*`_step`, the main program detects the reinterruption through the overrun flag and an error condition results. If the main program detects this condition, `rt_OneStep` signals an error and returns immediately. You can change this behavior if you want to handle the condition differently.

The design of `rt_OneStep` assumes that the main program disables interrupts before calling `rt_OneStep`. The main program should not interrupt `rt_OneStep` until after checking the setting of the interrupt overflow flag.

Multirate Multitasking Operation

In a multirate multitasking system, the generated example main program uses a prioritized, preemptive multitasking scheme to execute the different sample rates in a model.

This pseudocode shows the design of `rt_OneStep` in a multirate multitasking program.

```
rt_OneStep()
{
  Check for base-rate interrupt overrun
  Enable "rt_OneStep" interrupt
  Determine which rates need to run this time step

  Model_Step0()        -- run base-rate time step code

  For N=1:NumTasks-1   -- iterate over sub-rate tasks
    If (sub-rate task N is scheduled)
    Check for sub-rate interrupt overrun
      Model_StepN()    -- run sub-rate time step code
    EndIf
  EndFor
}
```

Task Identifiers

The generated main program groups the execution of blocks that use different sample rates into different tasks. The code generator assigns each block that executes at a given sample rate a *task identifier* (`tid`), which associates the code for that block with a task that executes at that rate. Where there are `NumTasks` tasks in the system, the range of task identifiers is 0..`NumTasks`-1.

Prioritization of Base-Rate and Subrate Tasks

The scheduler prioritizes tasks in descending order by rate. The *base-rate* task runs at the fastest rate in the system (the hardware clock rate). The base-rate task has highest priority (`tid` 0). The next fastest task (`tid` 1) has the next highest priority, and so on down to the slowest, lowest priority task (`tid` `NumTasks`-1).

The slower tasks, running at multiples of the base rate, are *subrate* tasks.

Rate Grouping and Rate-Specific model_step Functions

For a single-rate model, the main program performs block output computations by calling a single entry-point function, *`model`*`_step`. For multirate, multitasking models, the main program applies *rate*

grouping. The generated code includes a separate *model*_step entry-point function for the base rate task and each subrate task in the model. The function naming convention for these functions is as follows, where *N* is a task identifier:

*model*_step*N*

For example, for a model named my_model that has three rates, the code generator produces these entry-point functions:

```
void my_model_step0 (void);
void my_model_step1 (void);
void my_model_step2 (void);
```

Each *model*_step*N* function executes the code generated for blocks that share tid *N*. The code generator groups block code that executes within task *N* into the associated *model*_step*N* function.

Scheduling model_stepN Execution

On each clock tick, rt_OneStep maintains scheduling counters and *event flags* for each subrate task. The code generator implements:

- Counters as taskCounter arrays indexed on tid
- Event flags as arrays indexed on tid.

The scheduler uses rt_OneStep to maintain scheduling counters and task flags for subrates. The scheduling counters are clock rate dividers that count up the sample period associated with each subrate task. A pair of tasks that exchanges data maintains an interaction flag at the faster rate. Task interaction flags indicate that fast and slow tasks are scheduled to run.

The event flags indicate whether a given task is scheduled for execution. rt_OneStep maintains the event flags based on a task counter that is maintained by code in the main program for the model. When a counter indicates that the sample period for a task has elapsed, the main code sets the event flag for that task.

On each invocation, rt_OneStep updates its scheduling data structures and steps the base-rate task (rt_OneStep calls *model*_step0 because the base-rate task must execute on every clock step). Then, rt_OneStep iterates over the scheduling flags in tid order, unconditionally calling *model*_step*N* for each task for which the flag is set. The scheduler executes the tasks in priority order.

Preemption

The design of rt_OneStep assumes that the main program disables interrupts before calling rt_OneStep. The main program should not interrupt rt_OneStep until after checking the base-rate interrupt overflow flag (see pseudocode above).

The event flag array and loop variables that rt_OneStep uses are stored as local stack variables, making rt_OneStep reentrant. If rt_OneStep is reinterrupted, higher priority tasks preempt lower priority tasks. Upon return from interrupt, lower priority tasks resume in the previously scheduled order.

Overrun Detection

Multirate rt_OneStepmaintains an array of timer overrun flags. rt_OneStep detects timer overrun, per task, by the same logic as single-rate rt_OneStep.

If have developed multirate S-functions, or if you use a customized static main program module, see “Rate Grouping Compliance and Compatibility Issues” on page 71-12 for information about how to adapt your code for rate grouping compatibility. This adaptation produces more efficient code for multirate, multitasking models.

Multirate Single-Tasking Operation

In a multirate single-tasking program, sample times in the model must be an integer multiple of the fixed-step size configured for the model.

In a multirate single-tasking program, blocks execute at different rates, but under the same task identifier. The operation of `rt_OneStep`, is a simplified version of multirate multitasking operation. Rate grouping is not used. The only task is the base-rate task. The code generator produces one *`model`*`_step` entry-point function:

```
void model_step(void)
```

On each clock tick, `rt_OneStep` checks the overrun flag and calls *`model`*`_step`. The scheduling function for a multirate single-tasking program is `rate_scheduler` (rather than `rate_monotonic_scheduler`). The scheduler maintains scheduling counters on each clock tick. There is one counter for each sample rate in the model. The counters are implemented in an array (indexed on `tid`) within the `Timing` structure of `rtModel`.

The counters are clock rate dividers that count the sample period associated with each subrate task. When a counter indicates that a sample period for a given rate has elapsed, `rate_scheduler` clears the counter. This condition indicates that blocks running at that rate should execute on the next call to *`model`*`_step`, which is responsible for checking the counters.

Guidelines for Modifying rt_OneStep

`rt_OneStep` does not require extensive modification. The only required change is to reenable interrupts after the overrun flags and error conditions are checked. If applicable:

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling *`model`*`_step0`.
- Get model outputs associated with the base rate after calling *`model`*`_step0`. If you modify `rt_OneStep` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline below.
- In a multirate, multitasking model, set model inputs associated with subrates before calling *`model`*`_step`*`N`* in the subrate loop.
- In a multirate, multitasking model, get model outputs associated with subrates after calling *`model`*`_step`*`N`* in the subrate loop.

Comments in `rt_OneStep` indicate the place to add your code.

In multirate `rt_OneStep`, you can optimize the generated code by unrolling `for` and `while` loops.

In addition, you can modify the overrun behavior to continue execution after error recovery is complete.

Observe these guidelines:

- You should not modify the way in which the counters, event flags, or other timing data structures are set in `rt_OneStep`, or in functions called from `rt_OneStep`. The `rt_OneStep` timing data structures (including `rtModel`) and logic are critical to the operation of the generated program.
- If you customize the main program module to read model outputs after each base-rate model step and select model configuration parameters **Support: continuous time** and **Single output/update function** can cause output values read from `main` for a continuous output port to differ from the corresponding output values in data logged for the model. While logged data is a snapshot of output at major time steps, output read from `main` after the base-rate model step function executes reflects intervening minor time steps. To eliminate the discrepancy, separate the generated output and update functions (clear model configuration parameter **Single output/update function**) or place a Zero-Order Hold block before the continuous output port.
- You might observe a mismatch between results from simulation and logged MAT file results from generated code if you do not set model inputs before each time you call the model step function. In the generated example main program, the following comments show the locations for setting the inputs and stepping the model with your code:

  ```
  /* Set model inputs here */
  /* Step the model */
  ```

  If your model applies signal reuse and you are using `MatFileLogging` for comparing results from simulation against generated code, modify `rt_OneStep` to write model inputs in every time step as directed by these comments. Alternatively, you use a SIL or PIL approach for verification (see “Choose a SIL or PIL Approach” on page 88-11).

Static Main Program Module

- “Overview” on page 71-8
- “Rate Grouping and Static Main Program” on page 71-9
- “Modify the Static Main Program” on page 71-10
- “Modify Static Main to Allocate and Access Model Instance Data” on page 71-10

Overview

A strategy for deploying generated application code is to use model configuration parameter **Generate an example main program option** to generate an example main program module (`ert_main.c` or `.cpp`). For more information, see “Generate Code for an Application” on page 71-2.

Alternatively, you can clear the **Generate an example main program** parameter and use a generated static main program module as an example or template for developing your embedded application. Static main program modules provided by MathWorks include:

- *`matlabroot`*`/rtw/c/src/common/rt_main.c` — Supports the `Nonreusable function` setting for model configuration parameter **Code interface packaging**.
- *`matlabroot`*`/rtw/c/src/common/rt_malloc_main.c` — Supports the `Reusable function` setting for parameter **Code interface packaging**. Select model configuration parameter **Use dynamic memory allocation for model initialization** and set parameter **Pass root-level I/O as** to `Part of model data structure`.
- *`matlabroot`*`/rtw/c/src/common/rt_cppclass_main.cpp` — Supports the `C++ class` setting for parameter **Code interface packaging**.

Static main programs are not part of the generated code. The programs are provided as a basis for developing custom modifications and for use in simulation.

A static main program contains:

- `rt_OneStep`—A timer interrupt service routine (ISR) that calls the entry-point function *`model`*`_step` to execute processing for one clock period of the model.
- Skeletal `main` function—As provided, `main` is useful for simulation only. You must modify `main` for real-time interrupt-driven execution.

For single-rate models, operation of `rt_OneStep` and the `main` function in the static main program are the same as they are in the automatically generated version described in “Deploy Applications to Target Hardware” on page 71-2. For multirate, multitasking models, the static main program and generated code differ as described in the next section.

To develop an application by using a static main program:

1. Copy the program module to your working folder.
2. Rename the file. For example, rename `rt_main.c` to *`model`*`_rt_main.c`.
3. Customize the program module content.
4. If you are using a template makefile build approach, modify the template makefile settings such that the build process creates a corresponding object file in the build folder. For example, depending on the target platform, *`model`*`_rt_main.obj` or *`model`*`_rt_main.o`.

   If you are using a makefile-based toolchain approach:

   a. Use the `setTargetProvidesMain` function to disable the inclusion of the shipped static `main` program in the build process.
   b. Use the `addSourceFiles` function to add the source file for your `main` program.

If your existing application depends upon a static `ert_main.c` developed in releases before R2012b, `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`, you might need to continue using that static main program.

Rate Grouping and Static Main Program

ERT-based system target files can use a static main program module and disallow use of the model configuration parameter **Generate an example main program**. A custom system target file does not disallow use of that model configuration parameter when target-specific changes have been made to the static main program and those modifications must be preserved. A generated example main program does not preserve the changes to the static main program.

For models configured for multitasking (**Treat each discrete rate as a separate task** is selected) or concurrency (**Allow task to execute concurrently on target is** selected), the code generator uses a rate grouping scheme, producing a step entry-point function for each rate (*`model`*`_step`*`N`*, where *`N`* identifies the rate). The Code Interface Report lists the individual entry-point functions that a main program can call.

To specify that only the rate-based step functions are generated without the wrapper function, use the TLC variable `RateBasedStepFcn`. If your target calls the rate grouping compatible function directly, set `RateBasedStepFcn` to `1`. In this case, the wrapper function is not generated.

In your system target file, prior to the `%include "codegenentry.tlc"` statement, set the TLC variable `RateBasedStepFcn` to `1`. Alternatively, set `RateBasedStepFcn` to `1` in your `target_settings.tlc` file.

Modify the Static Main Program

Make few modifications to the main loop and the `rt_OneStep` function. See "Guidelines for Modifying Main Program" on page 71-3 and "Guidelines for Modifying rt_OneStep" on page 71-7.

Replace the `rt_OneStep` call in the main loop with a background task call or null statement.

Other modifications you might need to make are:

- If applicable, follow comments in the code regarding where to add code for reading and writing model I/O and saving and restoring FPU context.
- If you modify `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp` to read a value from a continuous output port after each base-rate model step, see the relevant guideline in "Guidelines for Modifying rt_OneStep" on page 71-7.
- When you clear the model configuration parameter **Generate an example main program**, the code generator produces `rtmodel.h` to provide an interface between the main program module and generated model code. If you create your own static main program module, include `rtmodel.h`.

  Alternatively, you can suppress generation of `rtmodel.h` and include *`model`*`.h` directly in your main program module. To suppress generation of `rtmodel.h`, add this statement to your system target file:

  ```
  %assign AutoBuildProcedure = 0
  ```
- If you cleared model configuration parameter **Terminate function required**, remove or comment out these items in your production version of `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`:
  - The `#if TERMFCN...` compile-time error check
  - The call to `MODEL_TERMINATE`
- For only `rt_main.c`: If you do *not* want to combine output and update functions, clear model configuration parameter **Single output/update function** and make these changes in your production version of `rt_main.c`:
  - Replace calls to `MODEL_STEP` with calls to `MODEL_OUTPUT` and `MODEL_UPDATE`.
  - Remove the `#if ONESTEPFCN...` error check.
- The static program module `rt_main.c` does not support `Reusable function` code interface packaging. If you try to use it, a compile-time error occurs.

  ```
  #if MULTI_INSTANCE_CODE==1
  ```

Modify Static Main to Allocate and Access Model Instance Data

If you are using a static main program module and your model is configured for `Reusable function` code interface packaging, but the model configuration parameter **Use dynamic memory allocation for model initialization** is cleared, allocate model instance data statically or dynamically by the calling main code. Pointers to the individual model data structures, such as Block IO, DWork, and Parameters, must be set up in the top-level real-time model data structure.

To support main modifications, the build process generates a subset of these real-time model (RTM) macros, based on the data requirements of your model, into *model*.h.

RTM Macro Syntax	Description
rtmGetBlockIO(rtm)	Get the block I/O data structure
rtmSetBlockIO(rtm,val)	Set the block I/O data structure
rtmGetContStates(rtm)	Get the continuous states data structure
rtmSetContStates(rtm,val)	Set the continuous states data structure
rtmGetDefaultParam(rtm)	Get the default parameters data structure
rtmSetDefaultParam(rtm,val)	Set the default parameters data structure
rtmGetPrevZCSigState(rtm)	Get the previous zero-crossing signal state data structure
rtmSetPrevZCSigState(rtm,val)	Set the previous zero-crossing signal state data structure
rtmGetRootDWork(rtm)	Get the DWork data structure
rtmSetRootDWork(rtm,val)	Set the DWork data structure
rtmGetU(rtm)	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
rtmSetU(rtm,val)	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
rtmGetY(rtm)	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
rtmSetY(rtm,val)	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

To access individual model data structures within the RTM data structure, use these macros in your static main program. For example, consider the example model Reusable.

```
openExample('ecoder/ReusableCodeInterfaceExample', 'supportingFile', 'Reusable');
```

Suppose the model is configured with Reusable function code interface packaging, **Use dynamic memory allocation for model initialization** cleared, **Pass root-level I/O as** set to Individual arguments, and **Optimization** pane option **Remove root level I/O zero initialization** is cleared. Building the model generates these model data structures and model entry-points into Reusable.h:

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T Delay_DSTATE;                  /* '<Root>/Delay' */
} D_Work;

/* Parameters (auto storage) */
struct Parameters_ {
  real_T k1;                            /* Variable: k1
                                         * Referenced by: '<Root>/Gain'
                                         */
};

/* Model entry point functions */
extern void Reusable_initialize(RT_MODEL *const rtM, real_T *rtU_In1,
  real_T *rtU_In2, real_T *rtY_Out1);
extern void Reusable_step(RT_MODEL *const rtM, real_T rtU_In1, real_T
  rtU_In2, real_T *rtY_Out1);
```

If you do not select the model configuration parameter **Generate an example main program** for the model, `Reusable.h` contains definitions for the RTM macros `rtmGetDefaultParam`, `rtmsetDefaultParam`, `rtmGetRootDWork`, and `rtmSetRootDWork`.

For reference, the generated `rtmodel.h` file contains an example parameter definition that has initial values (nonexecuting code):

```
#if 0

/* Example parameter data definition with initial values */
static Parameters rtP = {
  2.0                                   /* Variable: k1
                                         * Referenced by: '<Root>/Gain'
                                         */
};                                      /* Modifiable parameters */

#endif
```

In the definitions section of your static main file, you can use this code to statically allocate the real-time model data structures and arguments for the `Reusable` model:

```
static RT_MODEL rtM_;
static RT_MODEL *const rtM = &rtM_;    /* Real-time model */
static Parameters rtP = {
  2.0                                   /* Variable: k1
                                         * Referenced by: '<Root>/Gain'
                                         */
};                                      /* Modifiable parameters */

static D_Work rtDWork;                  /* Observable states */

/* '<Root>/In1' */
static real_T rtU_In1;

/* '<Root>/In2' */
static real_T rtU_In2;

/* '<Root>/Out1' */
static real_T rtY_Out1;
```

In the body of your main function, you can use these RTM macro calls to set up the model parameters and DWork data in the real-time model data structure:

```
int_T main(int_T argc, const char *argv[])
{
...
/* Pack model data into RTM */

rtmSetDefaultParam(rtM, &rtP);
rtmSetRootDWork(rtM, &rtDWork);

/* Initialize model */
Reusable_initialize(rtM, &rtU_In1, &rtU_In2, &rtY_Out1);
...
}
```

Follow a similar approach to set up multiple instances of model data, where the real-time model data structure for each instance has its own data. You must initialize the parameter structure (`rtP`) for each instance, to the values that you want, statically as part of the `rtP` data definition or at run time.

Rate Grouping Compliance and Compatibility Issues

- “Main Program Compatibility” on page 71-13
- “Make Your S-Functions Rate Grouping Compliant” on page 71-13

Main Program Compatibility

When you clear model configuration parameter **Generate an example main program**, the code generator produces slightly different rate grouping code, for compatibility with the older static `ert_main.c` module. See “Rate Grouping and Static Main Program” on page 71-9 for details.

Make Your S-Functions Rate Grouping Compliant

Built-in Simulink blocks and DSP System Toolbox blocks are compliant with the requirements for generating rate grouping code. However, user-written multirate inlined S-functions might not be rate grouping compliant. Noncompliant blocks generate less efficient code, but are otherwise compatible with rate grouping. To take full advantage of the efficiency of rate grouping, upgrade multirate inlined S-functions to be fully rate grouping compliant. You should upgrade your TLC S-function implementations, as described in this section.

Use of noncompliant multirate blocks to generate rate-grouping code generates dead code. This can cause two problems:

- Reduced code efficiency.
- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code does not run there is no impact to ,the run-time behavior of the generated code.

To make your S-functions rate grouping compliant, use these TLC functions to generate `ModelOutputs` and `ModelUpdate` code, respectively:

```
OutputsForTID(block, system, tid)
UpdateForTID(block, system, tid)
```

These code listings illustrate generation of output computations without rate grouping (Listing 1) and with rate grouping (Listing 2).

- The `tid` argument is a task identifier (`0..NumTasks-1`).
- Only code guarded by the `tid` passed into `OutputsForTID` is generated. The `if (%<LibIsSFcnSampleHit(portName)>)` test is not used in `OutputsForTID`.
- When generating rate grouping code, `OutputsForTID` and/or `UpdateForTID` is called during code generation. When generating non-rate-grouping code, `Outputs` and/or `Update` is called.
- In rate grouping compliant code, the top-level `Outputs` and/or `Update` functions call `OutputsForTID` and/or `UpdateForTID` functions for each rate (`tid`) involved in the block. The code returned by `OutputsForTID` and `UpdateForTID` must be guarded by the corresponding `tid` guard as in lListing 2:

  ```
  if (%<LibIsSFcnSampleHit(portName)>)
  ```

Listing 1: Outputs Code Generation Without Rate Grouping

```
%% multirate_blk.tlc

%implements "multirate_blk" "C"


%% Function: mdlOutputs ======================================================
%% Abstract:
%%
%%  Compute the two outputs (input signal decimated by the
%%  specified parameter). The decimation is handled by sample times.
%%  The decimation is only performed if the block is enabled.
```

```
%%  Each port has a different rate.
%%
%%  Note, the usage of the enable should really be protected such that
%%  each task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
%%
  %function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign enable = LibBlockInputSignal(0, "", "", 0)
  {
    int_T *enabled = &%<LibBlockIWork(0, "", "", 0)>;

    %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
      %% Only check the enable signal on a major time step.
      if (%<LibIsMajorTimeStep()> && ...
                          %<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %else
      if (%<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %endif

    if (*enabled) {
      %assign signal = LibBlockInputSignal(1, "", "", 0)
      if (%<LibIsSFcnSampleHit("OutputPortIdx0")>) {
        %assign y = LibBlockOutputSignal(0, "", "", 0)
        %<y> = %<signal>;
      }
      if (%<LibIsSFcnSampleHit("OutputPortIdx1")>) {
        %assign y = LibBlockOutputSignal(1, "", "", 0)
        %<y> = %<signal>;
      }
    }
  }

  %endfunction
%% [EOF] sfun_multirate.tlc
```

Listing 2: Outputs Code Generation With Rate Grouping

```
%% example_multirateblk.tlc

%implements "example_multirateblk" "C"


  %% Function: mdlOutputs ====================================================
  %% Abstract:
  %%
  %% Compute the two outputs (the input signal decimated by the
  %% specified parameter). The decimation is handled by sample times.
  %% The decimation is only performed if the block is enabled.
  %% All ports have different sample rate.
  %%
  %% Note: the usage of the enable should really be protected such that
  %% each task has its own enable state. In this example, the enable
  %% occurs immediately which may or may not be the expected behavior.
  %%
  %function Outputs(block, system) Output



  %assign portIdxName = ["InputPortIdx0","OutputPortIdx0","OutputPortIdx1"]
  %assign portTID     = [%<LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")>, ...
                         %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")>, ...
                         %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")>]
  %foreach i = 3
    %assign portName = portIdxName[i]
    %assign tid      = portTID[i]
    if (%<LibIsSFcnSampleHit(portName)>) {
                       %<OutputsForTID(block,system,tid)>
```

```
    }
  %endforeach

  %endfunction

  %function OutputsForTID(block, system, tid) Output
  /* %<Type> Block: %<Name> */
  %assign enable = LibBlockInputSignal(0, "", "", 0)
  %assign enabled = LibBlockIWork(0, "", "", 0)
  %assign signal = LibBlockInputSignal(1, "", "", 0)

  %switch(tid)
    %case LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")
                          %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
                            %% Only check the enable signal on a major time step.
                            if (%<LibIsMajorTimeStep()>) {
                              %<enabled> = (%<enable> > 0.0);
                            }
                          %else
                            %<enabled> = (%<enable> > 0.0);
                          %endif
                          %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")
                          if (%<enabled>) {
                            %assign y = LibBlockOutputSignal(0, "", "", 0)
                            %<y> = %<signal>;
                          }
                          %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")
                          if (%<enabled>) {
                            %assign y = LibBlockOutputSignal(1, "", "", 0)
                            %<y> = %<signal>;
                          }
                          %break
    %default
                          %% error it out
  %endswitch

  %endfunction

%% [EOF] sfun_multirate.tlc
```

Generate Code That Dereferences Data from a Literal Memory Address

This example shows how to generate code that reads the value of a signal by dereferencing a memory address that you specify. With this technique, you can generate a control algorithm that interacts with memory that your hardware populates (for example, memory that stores the output of an analog-to-digital converter in a microcontroller).

In this example, you generate an algorithm that acquires input data from a 16-bit block of memory at address `0x8675309`. Assume that a hardware device asynchronously populates only the lower 10 bits of the address. The algorithm must treat the address as read-only (`const`), volatile (`volatile`) data, and ignore the upper 6 bits of the address.

The generated code can access the data by defining a macro that dereferences `0x8675309` and masks the unnecessary bits:

```
#define A2D_INPUT ((*(volatile const uint16_T *)0x8675309)&0x03FF)
```

To configure a model to generate code that defines and uses this macro:

1 Create the example model.

2. Create package to contain definitions of data class and storage class.
3. Use Custom Storage Class Designer to create a storage class.
4. Define a class to store property settings for the storage class.
5. Write Target Language Compiler (TLC) code that emits the correct C code for the storage class.
6. Define a class for the inport data.
7. Load the package into the Embedded Coder® Dictionary.
8. Configure the model root-level inport to use the storage class.
9. Generate and inspect the code.

For an example that shows how to use the Custom Storage Class Designer without writing TLC code, see “Create and Apply a Storage Class” on page 40-14.

As an alternative to writing TLC code, you can use memory sections to generate code that includes pragmas. Depending on your build toolchain, you can use pragmas to specify a literal memory address for storing a global variable. For more information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas” on page 45-2.

Derivation of Macro Syntax

In this example, you configure the generated code to define and use the dereferencing macro. To determine the correct syntax for the macro, start by recording the target address.

```
0x8675309
```

Cast the address as a pointer to a 16-bit integer. Use the Simulink® Coder™ data type name `uint16_T`.

```
(uint16_T *)0x8675309
```

Add the storage type qualifier `const` because the generated code must not write to the address. Add `volatile` because the hardware can populate the address at an arbitrary time.

```
(volatile const uint16_T *)0x8675309
```

Dereference the address.

```
*(volatile const uint16_T *)0x8675309
```

After the dereference operation, apply a mask to retain only the 10 bits that the hardware populates. Use explicit parentheses to control the order of operations.

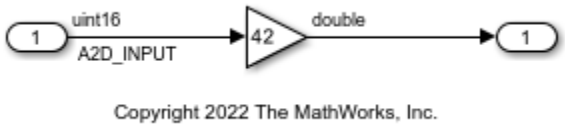
```
(*(volatile const uint16_T *)0x8675309)&0x03FF
```

As a safe coding practice, wrap the entire construct in another layer of parentheses.

```
((*(volatile const uint16_T *)0x8675309)&0x03FF)
```

Create Example Model

Create the example model `ex_memmap_simple`.



For the Inport block, set the output data type to `uint16`. Name the signal as `A2D_INPUT`. The `Inport` block and the signal line represent the data that the hardware populates.

For the Gain block, set the output data type to `double`.

Create Package to Contain Definitions of Data Class and Storage Class

In your current folder, create a folder named `+MemoryMap`. The folder defines a package named `MemoryMap`.

To make the package available for use outside of your current folder, you can add the folder containing the `+MemoryMap` folder to the MATLAB® path.

Create Storage Class

To generate code that defines and reads `A2D_INPUT` as a macro, you must create a storage class that you can apply to the model root-level inport. Later, you write TLC code that complements the storage class.

Open the Custom Storage Class designer in advanced mode. To design a storage class that operates through custom TLC code, you must use the advanced mode.

```
cscdesigner('MemoryMap','-advanced');
```

In the Custom Storage Class Designer, click **New**. A new custom storage class, `NewCSC_1`, appears in the list of storage class definitions.

Rename the storage class to `MemoryMappedAddress`.

For `MemoryMappedAddress`, on the **General** tab, set:

- **Type** to `Other`. The storage class can operate through custom TLC code that you write later.
- **Data scope** to `Exported`. For data items that use this custom storage class, Simulink Coder generates the definition (for example, the `#define` statement that defines a macro).
- **Data initialization** to `None`. Simulink Coder does not generate code that initializes the data item. Use this setting because this custom storage class represents read-only data. You do not select `Macro` because the Custom Storage Class Designer does not allow you to use `Macro` for signal data.
- **Definition file** to `Specify` (leave the text box empty). For data items that consume memory in the generated code, **Definition file** specifies the `.c` source file that allocates the memory. This storage class yields a macro, which does not require memory. Typically, header files (`.h`), not `.c` files, define macros. Setting **Definition file** to `Specify` instead of `Instance specific` prevents users of the storage class from unnecessarily specifying a definition file.
- **Header file** to `Instance specific`. To control the file placement of the macro definition, the user of the storage class must specify a header file for each data item that uses this storage class.
- **Owner** to `Specify` (leave the text box empty). **Owner** applies only to data items that consume memory.

After you finish selecting the settings, click **Apply** and **Save**.

Now, when you apply the storage class to a data item, such as inport `In1`, you can specify a header file to contain the generated macro definition. You cannot yet specify a memory address for the data item. To enable specification of a memory address, create a custom attributes class that you can associate with the `MemoryMappedAddress` storage class.

Define Class to Store Property Settings for Storage Class

Define a MATLAB class to store additional information for data items that use the storage class. In this case, the additional information is the memory address.

In the `MemoryMap` package (the `+MemoryMap` folder), create a folder named `@MemoryMapAttribs`.

In the `@MemoryMapAttribs` folder, create a file named `MemoryMapAttribs`. The file defines a class that derives from the built-in class `Simulink.CustomStorageClassAttributes`.

```
classdef MemoryMapAttribs < Simulink.CustomStorageClassAttributes
    properties( PropertyType = 'char' )
        MemoryAddress = '';
    end
end
```

Later, you associate this MATLAB class with the `MemoryMappedAddress` storage class. Then, when you apply the storage class to a data item, you can specify a memory address.

Write TLC Code That Emits Correct C Code

Write TLC code that uses the attributes of the storage class, such as `HeaderFile` and `MemoryAddress`, to generate correct C code for each data item.

In the `+MemoryMap` folder, create a folder named `tlc`.

Navigate to the new folder.

Inspect the built-in template TLC file, `TEMPLATE_v1.tlc`.

```
edit(fullfile(matlabroot,...
    'toolbox','rtw','targets','ecoder','csc_templates','TEMPLATE_v1.tlc'))
```

Save a copy of `TEMPLATE_v1.tlc` in the `tlc` folder. Rename the copy `memory_map_csc.tlc`.

In `memory_map_csc.tlc`, find the portion that controls the generation of C-code data declarations.

```
%case "declare"

  %% LibDefaultCustomStorageDeclare is the default declare function to
  %% declares a global variable whose identifier is the name of the data.
  %return "extern %<LibDefaultCustomStorageDeclare(record)>"
  %%break

%% ============================================================================
```

The `declare` case (`%case`) constructs a return value (`%return`), which the code generator emits into the header file that you specify for each data item. To control the C code that declares each data item, adjust the return value in the `declare` case.

Replace the existing `%case` content with this new code, which specifies a different return value:

```
%case "declare"

        %% In TLC code, a 'record' is a data item (for example, a signal line).
        %% 'LibGetRecordIdentifier' returns the name of the data item.
        %assign id = LibGetRecordIdentifier(record)

        %assign dt = LibGetRecordCompositeDataTypeName(record)

        %% The 'CoderInfo' property of a data item stores a
        %% 'Simulink.CoderInfo' object, which stores code generation settings
        %% such as the storage class or custom storage class that you specify
        %% for the item.
        %assign ci = record.Object.ObjectProperties.CoderInfo
        %% The 'ci' variable now stores the 'Simulink.CoderInfo' object.

        %% By default, the 'CustomAttributes' property of a 'Simulink.CoderInfo'
        %% object stores a 'Simulink.CustomStorageClassAttributes' object.
        %% This nested object stores specialized code generation settings
        %% such as the header file and definition file that you specify for
        %% the data item.
        %%
        %% The 'MemoryMap' package derives a new class,
        %% 'MemoryMapAttribs', from 'Simulink.CustomStorageClassAttributes'.
        %% The new class adds a property named 'MemoryAddress'.
        %% This TLC code determines the memory address of the data item by
        %% acquiring the value of the 'MemoryAddress' property.
        %assign ca = ci.Object.ObjectProperties.CustomAttributes
        %assign address = ca.Object.ObjectProperties.MemoryAddress

        %assign width = LibGetDataWidth(record)

        %% This TLC code constructs the full macro, with correct C syntax,
        %% based on the values of TLC variables such as 'address' and 'dt'.
        %% This TLC code also asserts that the data item must be a scalar.
        %if width == 1
                %assign macro = ...
                    "#define %<id> ((*(volatile const %<dt>*)%<address>) & 0x03FF)"
        %else
                %error( "Non scalars are not supported yet." )
        %endif

        %return "%<macro>"
      %%break

    %% ============================================================================
```

The new TLC code uses built-in, documented TLC functions, such as `LibGetRecordIdentifier`, and other TLC commands and operations to access information about the data item. Temporary variables such as `dt` and `address` store that information. The TLC code constructs the full macro, with the correct C syntax, by expanding the variables, and stores the macro in the variable `macro`.

In the same file, find the portion that controls the generation of data definitions.

```
%case "define"

  %% LibDefaultCustomStorageDefine is the default define function to define
  %% a global variable whose identifier is the name of the data.  If the
  %% data is a parameter, the definition is also statically initialized to
  %% its nominal value (as set in MATLAB).
  %return "%<LibDefaultCustomStorageDefine(record)>"
  %%break

%% ============================================================================
```

The `define` case derives a return value that the code generator emits into a `.c` file, which defines data items that consume memory.

Replace the existing `%case` content with this new content:

```
%case "define"
  %return ""
  %%break

%% ============================================================================
```

`MemoryMappedAddress` yields a macro in the generated code, so you use the `declare` case instead of the `define` case to construct and emit the macro. To prevent the `define` case from emitting a duplicate macro definition, the new TLC code returns an empty string.

Find the portion that controls the generation of code that initializes data.

```
%case "initialize"

  %% LibDefaultCustomStorageInitialize is the default initialization
  %% function that initializes a scalar element of a global variable to 0.
  %return LibDefaultCustomStorageInitialize(record, idx, reim)
  %%break

%% ============================================================================
```

The `initialize` case generates code that initializes data items (for example, in the *`model`*`_initialize` function).

Replace the existing `%case` content with this new content:

```
%case "initialize"
  %return ""
  %%break

%% ============================================================================
```

`MemoryMappedAddress` yields a macro, so the generated code must not attempt to initialize the value of the macro. The new TLC code returns an empty string.

Complete the Definition of the Storage Class

Your new MATLAB class, `MemoryMapAttribs`, can enable users of your storage class, `MemoryMappedAddress`, to specify a memory address for each data item. To allow this specification, associate `MemoryMapAttribs` with `MemoryMappedAddress`. To generate correct C code based on the information that you specify for each data item, associate the customized TLC file, `memory_map_csc.tlc`, with `MemoryMappedAddress`.

Navigate to the folder that contains the `+MemoryMap` folder.

Open the Custom Storage Class Designer again.

```
cscdesigner('MemoryMap','-advanced');
```

For `MemoryMappedAddress`, on the **Other Attributes** tab, set:

- **TLC file name** to `memory_map_csc.tlc`.
- **CSC attributes class** to `MemoryMap.MemoryMapAttribs`.

Click **Apply** and **Save**.

Define Signal Data Class

To apply the storage class to a data element in a model, in the `MemoryMap` package, you must create a MATLAB class that derives from `Simulink.Signal`. When you configure the signal in the model, you select this new data class instead of the default class, `Simulink.Signal`.

In the `MemoryMap` package, create a folder named `@Signal`.

In the `@Signal` folder, create a file named `Signal.m`.

```
classdef Signal < Simulink.Signal
    methods
        function setupCoderInfo( this )
            useLocalCustomStorageClasses( this, 'MemoryMap' );
            return;
        end
    end
end
```

The file defines a class named `MemoryMap.Signal`. The class definition overrides the `setupCoderInfo` method, which the `Simulink.Signal` class already implements. The new implementation specifies that objects of the `MemoryMap.Signal` class use custom storage classes from the `MemoryMap` package (instead of storage classes from the `Simulink` package). When you configure a signal in a model by selecting the `MemoryMap.Signal` class, you can select the new custom storage class, `MemoryMappedAddress`.

Load Package into Embedded Coder Dictionary

Navigate to the folder that contains the example model and open the model.

Open the **Embedded Coder** app.

Open the Embedded Coder Dictionary. On the **C Code** tab, select **Code Interface** > **Embedded Coder Dictionary**.

In the Embedded Coder Dictionary, click **Manage Packages**.

In the Manage Package dialog box, click **Refresh**. When the refresh is complete, select the package `MemoryMap`. Click **Load**.

Close the Embedded Coder Dictionary.

Configure Root-Level Inport to Use Storage Class

In the **Embedded Coder** app, use the Code Mappings editor and Property Inspector to configure Inport `In1` to use the storage class that you defined.

Open the Code Mappings editor. On the **C Code** tab, select **Code Interface** > **Individual Element Code Mappings**.

In the Code Mappings Editor, on the **Inports** tab, select inport `In1`. Set the storage class for `In1` to `MemoryMappedAddress`.

In the Property Inspector, under **Code**, set the **HeaderFile** property to `memory_mapped_addresses.h` and the **MemoryAddress** property to `0x8675309`.

Save the model.

Generate and Inspect Code

Generate code from the model.

```
### Starting build procedure for: ex_memmap_simple
### Successful completion of build procedure for: ex_memmap_simple

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
========================================================================================
ex_memmap_simple  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.063s
```

Inspect the generated header file `memory_mapped_addresses.h`. The file defines the macro `A2D_INPUT`, which corresponds to the signal line in the model.

```
/* Declaration of data with custom storage class MemoryMappedAddress */
#define A2D_INPUT                      ((*(volatile const uint16_T*)0x8675309) & 0x03FF)
```

Inspect the generated file `ex_memmap_simple.c`. The generated algorithmic code (which corresponds to the Gain block) calculates the model output, `rtY.Out1`, by operating on `A2D_INPUT`.

```
/* Model step function */
void step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
```

```
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 = 42.0 * (real_T)A2D_INPUT;
}
```

See Also

More About

- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3
- “How Generated Code Exchanges Data with an Environment” on page 36-35
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

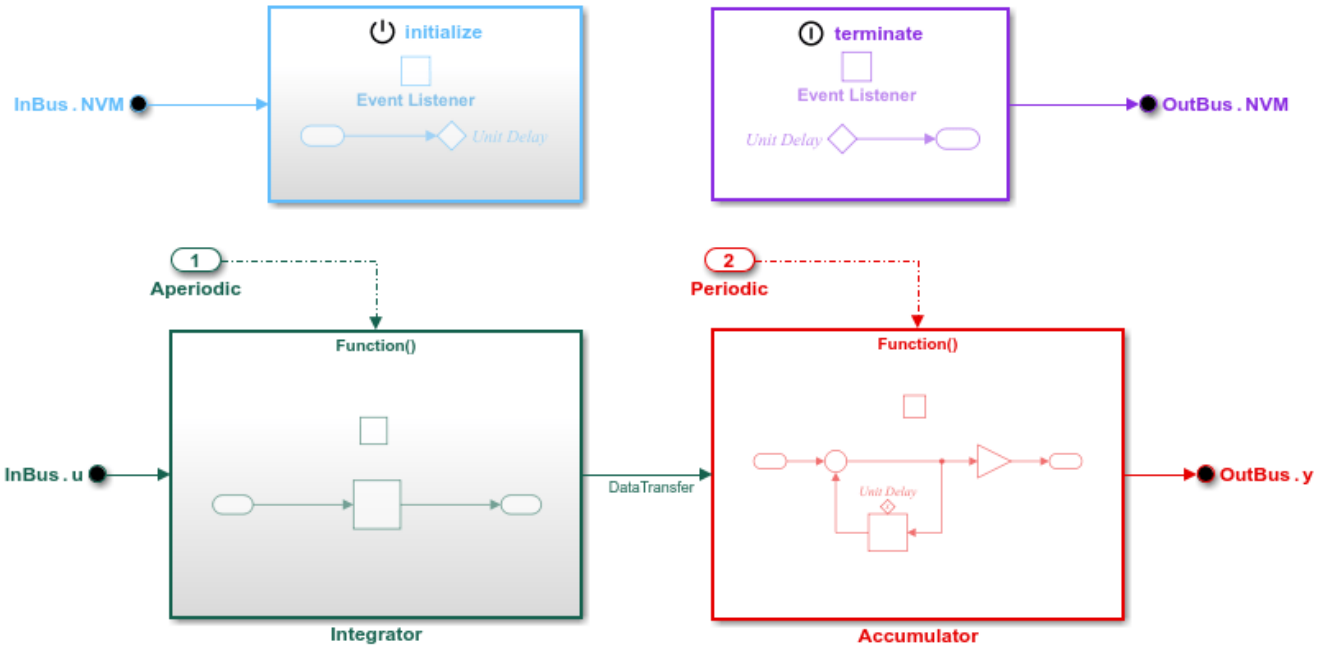
Deploy Component Algorithm as Component Model Library by Using CMake

This example shows how to use the CMake approach of building code from a component algorithm model for deploying the algorithm code as a component model library. CMake is a third-party, open-source tool for build process management. CMake uses configuration files (`CMakeLists.txt`) to generate standard build files for native build environments such as, makefiles, Ninja files, or Microsoft® Visual Studio® projects. The example guides you through these steps:

1. Open the model.
2. Generate component code from the model.
3. Implement handwritten integration code.
4. Configure CMake for your target environment.
5. Generate build artifacts and build integrated code.
6. Run the executable program.

Open Model

Open example model `ComponentDeploymentFcn.slx`.




```
open_system('ComponentDeploymentFcn');
```

Generate Component Code from Model

1. Configure your target platform for the code generator.
2. Generate source code.
3. Create a `CMakeLists.txt` file for the component model library.

4 Optionally, package generated code files and `CMakeLists.txt` files into a ZIP file that you can use to relocate the files.

Configure Model Target Environment Device Type

Inform the code generator about the C language constraints for your target platform by defining a workspace variable (*`prodHWDeviceType`*) that you can use for setting model configuration parameter `ProdHWDeviceType`.

```
switch computer('arch')
    case 'win64'
        prodHWDeviceType = 'Intel->x86-64 (Windows64)';
    case 'glnxa64'
        prodHWDeviceType = 'Intel->x86-64 (Linux 64)';
    case 'maci64'
        prodHWDeviceType = 'Intel->x86-64 (Mac OS X)';
    case 'maca64'
        prodHWDeviceType = 'Apple->ARM64';
    otherwise
        assert(false, 'Platform not supported');
end
set_param('ComponentDeploymentFcn', 'ProdHWDeviceType', prodHWDeviceType);
```

Generate Code and CMakeLists.txt for Component Model Library

To generate a platform-independent `CMakeLists.txt` file, specify the `CMake` toolchain.

```
set_param('ComponentDeploymentFcn', 'Toolchain', 'CMake');
```

Specify the creation of a ZIP file after code generation. The `packNGo` function packages the generated source code and CMake configuration file in the ZIP file.

```
set_param('ComponentDeploymentFcn', 'PackageGeneratedCodeAndArtifacts', 'on');
```

Generate code.

```
slbuild('ComponentDeploymentFcn')
```

```
### Starting build procedure for: ComponentDeploymentFcn
### Successful completion of code generation for: ComponentDeploymentFcn

Build Summary

Top model targets built:

Model                    Action           Rebuild Reason
========================================================================================
ComponentDeploymentFcn  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 26.364s
```

In the `services` subfolder beneath the build folder, the software generates a header file `services.h`. The file defines interfaces to services that you must implement for an application containing the generated component.

```
dir(fullfile('ComponentDeploymentFcn_ert_rtw', 'services', 'services.h'))
```

```
services.h
```

In the `services/lib` subfolder beneath the build folder, the software generates a `CMakeLists.txt` file that you can use to build the generated code into a static library.

```
dir(fullfile('ComponentDeploymentFcn_ert_rtw', 'services', 'lib', 'CMakeLists.txt'))
```

```
CMakeLists.txt
```

(Optional) Relocate Generated Files to Another Development Environment

You can relocate the ZIP file, which contains the generated code, build artifact, and `CMakeLists.txt` files to another development environment.

```
%% Optional
copyfile('ComponentDeploymentFcn.zip', 'path-for-destination');
cd path-for-destination
unzip('ComponentDeploymentFcn.zip');
```

Implement Handwritten Integration Code

main function

Implement handwritten code for a `main` function in your code generation folder (*`codeGenerationFolder/)`*. For this example, you can use the function in file `ComponentDeploymentFcnIntegration.c`.

```
open('ComponentDeploymentFcnIntegration.c');
```

```
#include "ComponentDeploymentFcn.h"
#include <stddef.h>
#include <stdio.h>
extern uint32_T tick;   // Variable used by timer service implementation
const size_t nIter = 3;
int main(void) {

    // Run model initialize function
    printf("\nRunning model initialize function\n");
    CD_initialize();

    for (size_t k = 0; k < nIter; ++k) {

        // Run periodic Accumulator function at every iteration
        printf("\nRunning Accumulator function (iteration %llu)\n", k);
        CD_accumulator();

        if (k == 1) {
            // The asynchronous Integrator function only runs when k == 1
            printf("\nRunning Integrator function (iteration %llu)\n", k);
            CD_integrator();
        }

        // Increment timer tick count
        tick++;
    }

    // Run model terminate function
    printf("\nRunning model terminate function\n");
    CD_terminate();
}
```

services.c

In your code generation folder (*codeGenerationFolder/*), create handwritten code that provides implementations of the target platform services declared in `services.h`. For this example, you can use the code in file `services.c`.

```
open('services.c');
```

```
#include "services.h"

#include <stdio.h>

uint32_T tick = 0;

static real_T DataTransfer_storage = 0;

static real_T input_storage = 0;
static real_T NVM_storage = 100;
static real_T output_storage = 0;


/* timer services */
uint32_T get_tick_outside_tick_CD_integrator(void) {
    printf("Service function called: get_tick_outside_tick_CD_integrator.\n");
    return tick;
}

/* data transfer services */
const real_T * get_CD_accumulator_DataTransfer(void) {
    printf("Service function called: get_CD_accumulator_DataTransfer.\n");
    return &DataTransfer_storage;
}

real_T * set_CD_integrator_DataTransfer(void) {
    printf("Service function called: set_CD_integrator_DataTransfer.\n");
    return &DataTransfer_storage;
}

real_T * set_ModelInitialize_DataTransfer(void) {
    printf("Service function called: set_ModelInitialize_DataTransfer.\n");
    return &DataTransfer_storage;
}

/* receiver services */
const real_T * get_CD_integrator_InBus_u(void) {
    printf("Service function called: get_CD_integrator_InBus_u.\n");
    return &input_storage;
}

const real_T * get_CD_initialize_InBus_NVM(void) {
    printf("Service function called: get_CD_initialize_InBus_NVM.\n");
    return &NVM_storage;
}

/* sender services */
real_T * getref_CD_accumulator_OutBus_y(void) {
    printf("Service function called: getref_CD_accumulator_OutBus_y.\n");
    return &output_storage;
```

```
}

real_T * getref_CD_terminate_OutBus_NVM(void) {
    printf("Service function called: getref_CD_terminate_OutBus_NVM.\n");
    return &NVM_storage;
}
```

CMakeLists.txt

To produce an executable application that incorporates the handwritten main function and service implementations, and links with the library generated from the Simulink model, in your code generation folder (*codeGenerationFolder*), create a handwritten `CMakeLists.txt` file. The file must perform these tasks:

- Import the previously generated `CMakeLists.txt` file that contains the rules to build the static library from the component code.
- Add a rule to build the handwritten source files `ComponentDeploymentFcnIntegration.c` and `services.c` into an executable program.
- Link the executable program with the static library that contains the compiled component code.

For this example, you can use the provided `CMakeLists.txt` file.

```
cmake_minimum_required(VERSION 3.12)

# Set a name for the project
project(ComponentDeploymentFcnIntegration)

# Use the "add_subdirectory" command to reference the CMakeLists.txt file
# that was generated by codebuild for the code generated from
# the Simulink model.  This has defined a static library target
# named "ComponentDeploymentFcn" which we will link into
# the application further down.
add_subdirectory(ComponentDeploymentFcn_ert_rtw/services/lib)

# Use the "add_executable" command to define an executable target named
# "ComponentDeploymentFcnIntegration" that is built from the hand-written
# source files.
add_executable(ComponentDeploymentFcnIntegration
    ComponentDeploymentFcnIntegration.c
    services.c)

# Specify that the executable target "ComponentDeploymentFcnIntegration"
# incorporating the handwritten code should be linked against the the
# "ComponentDeploymentFcn" library target containing the code
# generated from the Simulink model.
target_link_libraries(ComponentDeploymentFcnIntegration ComponentDeploymentFcn)
```

Configure CMake

Configure the CMake command and arguments for your target environment. In the MATLAB Command Window, run one of these examples.

Windows - Microsoft Visual Studio 2017 Solution

```
cmakeCommand = ['"', fullfile(matlabroot, 'bin', 'win64', 'cmake', 'bin', 'cmake.exe'), '"'];
cmakeArguments = '-G "Visual Studio 15 2017"';
```

Windows - Microsoft Visual Studio 2019 Solution

```
cmakeCommand = ['"', fullfile(matlabroot, 'bin', 'win64', 'cmake', 'bin', 'cmake.exe'), '"'];
cmakeArguments = '-G "Visual Studio 16 2019"';
```

Windows - Microsoft Visual Studio with NMake

```
[status, msvcInstall] = system('"%ProgramFiles(x86)%\Microsoft Visual Studio\Installer\vswhere" -
msvcInstall = strtrim(msvcInstall);
assert(status == 0 && ~isempty(msvcInstall), 'Could not find location of Microsoft Visual Studio
vcvarsAll = fullfile(msvcInstall, 'VC', 'Auxiliary', 'Build', 'vcvarsall.bat');
envSetupCommand = ['"' vcvarsAll '" amd64 && '];
cmakeCommand = [envSetupCommand, '"', fullfile(matlabroot, 'bin', 'win64', 'cmake', 'bin', 'cmake
cmakeArguments = '-G "NMake Makefiles"';
```

Windows - MinGW

```
mingwLoc = getenv('MW_MINGW64_LOC');
assert(~isempty(mingwLoc), 'Could not find location of MinGW installation');
mingwLocBin = [mingwLoc, '/bin'];
envSetupCommand = ['set PATH=', mingwLocBin, ';%PATH% && '];
cmakeCommand = [envSetupCommand, '"', fullfile(matlabroot, 'bin', 'win64', 'cmake', 'bin', 'cmake
cmakeArguments = ['-G "MinGW Makefiles" ', ...
    '-DCMAKE_C_COMPILER="gcc.exe" ', ...
    '-DCMAKE_CXX_COMPILER="g++.exe"'];
```

Linux - GCC and GMake

```
cmakeCommand = ['"', fullfile(matlabroot, 'bin', 'glnxa64', 'cmake', 'bin', 'cmake'), '"'];
cmakeArguments = '-G "Unix Makefiles"';
```

Mac - XCode with Clang

```
cmakeCommand = ['"', fullfile(matlabroot, 'bin', computer('arch'), 'cmake', 'bin', 'cmake'), '"'
cmakeArguments = '-G "Xcode"';
```

Generate Build Artifacts and Build Integrated Code

To generate required build artifacts and build the integrated code, run CMake. The CMake tool processes the `CMakeLists.txt` file in the folder specified through the `-S` flag, and generates the configuration files for your build system in the folder that you specify through the `-B` flag. In *`codeGenerationFolder`*, run this commmand:

```
[status1, cmdout1] = system([cmakeCommand ' -S . -B build ', cmakeArguments], '-echo');
```

Using the `--build` flag, build the application.

```
[status2, cmdout2] = system([cmakeCommand ' --build build'], '-echo');
```

The location of the executable program file that the CMake generator produces depends on the CMake generator that you use.

- Makefile based CMake generator - Relative to your current working folder, places the executable program file in subfolder `build`.
- Project-based CMake generator - Relative to your current working folder, places the executable program file in folder `build/Debug`.

Windows - Visual Studio Solution

```
exeLocation = fullfile('build', 'Debug', 'ComponentDeploymentFcnIntegration.exe');
```

Windows - NMake Makefiles or GMake Makefiles

```
exeLocation = fullfile('build', 'ComponentDeploymentFcnIntegration.exe');
```

Linux - GMake Makefiles

```
exeLocation = fullfile('build', 'ComponentDeploymentFcnIntegration');
```

Mac - Xcode project

```
exeLocation = fullfile('build', 'Debug', 'ComponentDeploymentFcnIntegration');
```

Run Executable Program

Use this command to run the compiled executable program:

```
[status3, cmdout3] = system(exeLocation, '-echo');
```

Example output:

```
Running model initialize function
Service function called: set_ModelInitialize_DataTransfer.
Service function called: get_CD_initialize_InBus_NVM.

Running Accumulator function (iteration 0)
Service function called: get_CD_accumulator_DataTransfer.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.

Running Accumulator function (iteration 1)
Service function called: get_CD_accumulator_DataTransfer.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.

Running Integrator function (iteration 1)
Service function called: set_CD_integrator_DataTransfer.
Service function called: get_tick_outside_CD_integrator.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
```

```
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.
Service function called: get_CD_integrator_InBus_u.

Running Accumulator function (iteration 2)
Service function called: get_CD_accumulator_DataTransfer.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.
Service function called: getref_CD_accumulator_OutBus_y.

Running model terminate function
Service function called: getref_CD_terminate_OutBus_NVM.
```

```
close_system('ComponentDeploymentFcn',0)
```

See Also

slbuild

Related Examples

- “Configure CMake Build Process” on page 59-13

Generate Main Program for Deployment to Bare-Metal Target Hardware

This example shows how to configure a model such that the code generator produces an example main program that you can customize for deployment on bare-metal target hardware (does not have an operating system). When you select the model configuration parameter **Generate an example main program**, the code generator produces the example `ert_main.c` or `ert_main.cpp` file. The file includes:

- `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

Operation of the `main` program and the scheduling algorithm depend primarily on:

- Whether the model is a single-rate or multirate model
- Whether the model solver mode is set to single-tasking or multitasking

For more information, see “Deploy Applications to Target Hardware” on page 71-2.

Alternatively, you can configure a model to generate an example main program for deployment on target hardware that is running threaded code that runs on your host operating system. The example file `ert_main.c` that the code generator produces, shows how to deploy the generated example code.

You can customize a generated main program by using a custom file processing (CFP) template. Consider using a template file to:

- Assemble generated code in buffers
- Call an API to obtain buffered code into specific sections of generated source and header files

For more information, see File customization template.

Open Example Model

Open the example model `rtwdemo_examplemain`.

```
open_system('rtwdemo_examplemain');
```

Configure Model

1. Open the **Embedded Coder** app.

2. Select model configuration parameter **Generate an example main program**. When you select this parameter, you enable parameter **Target operating system**.

3. Set parameter **Target operating system** to `BareBoardExample` or `NativeThreadsExample`. For more information, see Target operating system.

4. Consider whether you want to specify a custom file processing template. To configure a template, specify the file name and extension for your template TLC file as a string for model configuration parameter **File customization template**.

Generate Code

Generate and inspect the code.

Add Hand-Coded Customizations

Augment the generated code with hand code as needed.

Deploy Generated Component Software to Application Target Platforms

The code generator supports integration of generated code with operating systems and processors. For details, see “Embedded Coder Supported Hardware”.

Interface to an Example Real-Time Multitasking System

This example shows how to simulate and generate code for asynchronous events on a real-time multitasking system.

The operating system integration techniques that the example demonstrates use one or more blocks in the `vxlib1` library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Example Model

Open the `rtwdemo_vxworks` model.

```
model = 'rtwdemo_vxworks';
open_system(model);
%
```

Model Description

The example model contains two asynchronously executed subsystems, `Count` and `Algorithm`. `Count` executes at interrupt level. `Algorithm` executes in an asynchronous task. The generated code for these blocks is tailored for the VxWorks® operating system. However, you can modify the `Async Interrupt` and `Task Sync` blocks to generate code for your run-time environment whether you are using an operating system or not.

Data Transfer Assumptions

This example assumes the following about the target platform environment:

- Data transitions occur between a single task that reads data and a single task that writes data.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact during a data transfer, only one of the tasks can preempt the other task.
- For periodic tasks, the task that executes at the faster rate has a higher priority than the tasks that executes at the slower rate. The task executing at the faster rate preempts the tasks executing at the slower rate.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash and restart during a data transfers.

Related Information

- Async Interrupt
- Task Sync
- “Generate Interrupt Service Routines”

- “Timers in Asynchronous Tasks”
- “Create a Customized Asynchronous Library”
- “Import Asynchronous Event Data for Simulation”
- “Load Data to Root-Level Input Ports”
- “Asynchronous Events”
- “Rate Transitions and Asynchronous Blocks”
- “Asynchronous Support Limitations”

See Also

More About

- “Map Embedded System Architecture to Simulink Modeling Environment” on page 2-3

Code Replacement Customization for Simulink Models in Embedded Coder

- “What Is Code Replacement Customization?” on page 72-3
- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Quick Start Code Replacement Library Development - Simulink” on page 72-21
- “Semaphore and Mutex Function Replacement” on page 72-30
- “Identify Code Replacement Library Requirements” on page 72-35
- “Define Code Replacement Library Optimizations” on page 72-36
- “Register Code Replacement Library” on page 72-51
- “Verify Code Replacement Library” on page 72-55
- “Deploy Code Replacement Library” on page 72-57
- “Troubleshoot Code Replacement Library Registration” on page 72-58
- “Troubleshoot Code Replacement Misses” on page 72-59
- “Math Function Code Replacement” on page 72-64
- “Buffer Index Calculation Code Replacement” on page 72-71
- “Memory Function Code Replacement” on page 72-78
- “Nonfinite Function Code Replacement” on page 72-83
- “Algorithm-Based Code Replacement” on page 72-88
- “Lookup Table Function Code Replacement” on page 72-94
- “Data Alignment for Code Replacement” on page 72-105
- “Replace MATLAB Functions with Custom Code by Using the coder.replace Function” on page 72-112
- “Replace coder.ceval Calls to External Functions” on page 72-118
- “Custom Function Code Replacement” on page 72-125
- “Reserved Identifiers and Code Replacement” on page 72-131
- “Customize Match and Replacement Process” on page 72-132
- “Scalar Operator Code Replacement” on page 72-145
- “Addition and Subtraction Algorithm Types for Code Replacement” on page 72-150
- “Replace Addition and Subtraction Operator Code” on page 72-153
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 72-160
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 72-170
- “Small Matrix Operation to Processor Code Replacement” on page 72-177
- “Remap Operator Output to Function Input” on page 72-185
- “Fixed-Point Operator Code Replacement” on page 72-188

- “Binary-Point-Only Scaling Code Replacement” on page 72-194
- “Slope Bias Scaling Code Replacement” on page 72-199
- “Net Slope Scaling Code Replacement” on page 72-205
- “Equal Slope and Zero Net Bias Code Replacement” on page 72-212
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 72-218
- “Shift Left Operations and Code Replacement” on page 72-224
- “Optimize Generated Code by Using Multiple Code Replacement Libraries” on page 72-229
- “Compare Rounding and Overflow Modes in C/C++ Code and Simulink” on page 72-233

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the **Code Replacement Tool** or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator's match and replacement process for functions

To get started, "Quick Start Code Replacement Library Development - Simulink" on page 72-21.

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

1. Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
2. Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
   - Conceptual name or key
   - Arguments, including quantity, type, type qualifiers, and complexity
   - Algorithm (computation method)
   - Fixed-point saturation and rounding modes
   - Priority
3. When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
4. Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See "Verify Code Replacement Library" on page 72-55.
- Code replacement for matrices — Code replacement libraries do not support the replacement of functions that have variable-size inputs.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is

not available from the Code Replacement Tool. See Specify build information in “Define Code Replacement Library Optimizations” on page 72-36.

- Addition and subtraction operation replacements—See “Replace Addition and Subtraction Operator Code” on page 72-153 for relevant limitations.
- Data alignment—
  - Not supported for
    - Arguments associated with a built-in storage class with `DataScope` set to `Exported` or the imported built-in storage class `GetSet`
    - Software-in-the-loop (SIL)
    - Processor-in-the-loop (PIL)
    - Model reference parameters
    - Exported functions in Stateflow charts
    - Replaced functions that are generated with C function prototype control or C++ class I/O arguments step method and that use root-level I/O variables
    - Replaced functions that are generated with the AUTOSAR system target file and that use root-level I/O or AUTOSAR inter-runnable access functions
  - If the following conditions exist, the code generator includes data alignment directives for root-level I/O variables in the `ert_main.c` or `ert_main.cpp` file it produces:
    - Compiler supports global variable alignment
    - Generate an example main program (select **Configuration Parameters** > **Generate an example main program**)
    - Generate a reusable function interface for the model (set **Configuration Parameters** > **Code Generation** > **Interface** > **Code interface packaging** to `Reusable function`)
    - Function uses root-level I/O variables that are passed in as individual arguments (set **Configuration Parameters** > **Code Generation** > **Interface** > **Pass root-level I/O as**to `Individual arguments`)
    - Replaced function uses a root-level I/O variable
    - Replaced function imposes alignment requirements
  - If you discard the generated example main program, align used root-level I/O variables correctly.
  - If you choose not to generate an example main program in this case, the code generator does not replace the function.
  - If a replacement imposes alignment requirements on the shared utility interface arguments, the code generator does not honor data alignment. Under these conditions, replacement does not occur. Replacement is allowed if the registered data alignment type specification supports alignment of local variables, and the replacement involves only local variables.
  - For `Simulink.Bus`:
    - If user registered alignment specifications do not support structure field alignment, aligning `Simulink.Bus` objects is not supported unless the `Simulink.Bus` is imported.
    - When aligning a `Simulink.Bus` data object, the elements in the bus object are aligned on the same boundary. The boundary is the lowest common multiple of the alignment requirements for each individual bus element.

- When you specify alignment for functions that occur in a model reference hierarchy, and multiple models in the hierarchy operate on the same function data, the bottommost model dictates alignment for the rest of the hierarchy. If the alignment requirement for a function in a model higher in the hierarchy cannot be honored due to the alignment set by a model lower in the hierarchy, the replacement in the higher model does not occur. In some cases, an error message is generated. To work around this issue, if the shared data is represented by a bus or signal object, manually set the alignment property on the shared data by setting the alignment property of the `Simulink.Bus` or `Simulink.Signal` object.
- It is your responsibility to honor the `Alignment` property setting for storage classes that you create.

See “Data Alignment for Code Replacement” on page 72-105.

- `coder.replace` function — See `coder.replace` for relevant limitations.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Quick Start Code Replacement Library Development - Simulink” on page 72-21
- “What Is Code Replacement?” on page 56-2

Code You Can Replace From Simulink Models

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

For information on how to explore functions and operators that a code replacement library supports, see “Choose a Code Replacement Library” on page 56-6.

Math Functions – Simulink Support

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`acos`	Floating point	Scalar	Real Complex input/complex output Real input/complex output
`acosd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acosh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`acot`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acotd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acoth`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acsc`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acscd`[2]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`acsch`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asec`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asecd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asech`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asin`	Floating point	Scalar	Real Complex input/complex output Real input/complex output
`asind`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asinh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`atan`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`atan2`[3]	Floating point	Scalar Vector Matrix	Real
`atan2d`[2]	Floating point	Scalar Vector Matrix	Real
`atand`[2]	Floating point	Scalar Vector Matrix	Real Complex
`atanh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`ceil`	Floating-point Scalar	Floating-point Scalar	Floating-point Scalar
`circularIndex` — Delay block with **Use circular buffer for state** selected.	Integer	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`cos`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`cosd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`cosh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`cot`[2]	Floating point	Scalar Vector Matrix	Real Complex
`cotd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`coth`[2]	Floating point	Scalar Vector Matrix	Real Complex
`csc`[2]	Floating point	Scalar Vector Matrix	Real Complex
`cscd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`csch`[2]	Floating point	Scalar Vector Matrix	Real Complex
`exactrSqrt` — Sqrt block with: • **Function** set to `rSqrt` • **Method** set to `Exact`	Integer Floating point	Scalar	Real
`exp`	Floating point	Scalar Vector Matrix	Real
`fix`	Floating point	Scalar	Real
`floor`	Floating-point Scalar	Floating-point Scalar	Floating-point Scalar
`fmod`[4]	Floating point	Scalar	Real
`frexp`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`hypot`	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real
`ln`	Floating point	Scalar	Real
`log`	Floating point	Scalar Vector Matrix	Real
`log10`	Floating point	Scalar Vector Matrix	Real
`log2`[2]	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point Fixed point	Scalar	Real
`min`	Integer Floating point Fixed point	Scalar	Real
`mod`	Integer Floating point	Scalar Vector Matrix	Real
`pow`	Floating point	Scalar Vector Matrix	Real
`rem`	Floating point	Scalar Vector Matrix	Real
`round`	Floating point	Scalar	Real
`rSqrt`	Integer Floating point	Scalar Vector Matrix	Real
`saturate`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`sec`[2]	Floating point	Scalar Vector Matrix	Real Complex
`secd`[2]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`sech`[2]	Floating point	Scalar Vector Matrix	Real Complex
`sign`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`signPow` — Math Function block with: • **Function** set to `pow` • **Signed power** enabled	Floating point	Scalar Vector Matrix	Real
`sin`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`sincos`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`sind`[2]	Floating point	Scalar Vector Matrix	Real Complex
`sinh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`sqrt`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`tan`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`tand`[2]	Floating point	Scalar Vector Matrix	Real Complex
`tanh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

[1] Wrap on integer overflow only. Clear block parameter **Saturate on integer overflow**.

[2] Only when used with the MATLAB Function block.

[3] Supports the CORDIC approximation method and the Lookup approximation method.

[4] Stateflow support only.

Math Functions – Stateflow Support

When generating C/C++ code from Stateflow charts, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Integer Floating point	Scalar	Real
`acos`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`acosd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acot`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acotd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acoth`[3,5]	Floating point	Scalar Vector Matrix	Real Complex
`acsc`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acscd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acsch`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asec`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asecd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asech`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`asin`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`asind`[3]	Floating point	Scalar Vector Matrix	Real Complex
`atan`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`atan2`[2]	Floating point	Scalar Vector Matrix	Real
`atan2d`[3]	Floating point	Scalar Vector Matrix	Real
`atand`[3]	Floating point	Scalar Vector Matrix	Real Complex
`ceil`	Floating-point Scalar	Floating-point Scalar	Floating-point Scalar
`cos`[3]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`cosd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`cosh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`cot`[3]	Floating point	Scalar Vector Matrix	Real Complex
`cotd`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`coth`[3]	Floating point	Scalar Vector Matrix	Real Complex
`csc`[3]	Floating point	Scalar Vector Matrix	Real Complex
`cscd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`csch`[3]	Floating point	Scalar Vector Matrix	Real Complex
`exp`	Floating point	Scalar	Real
`floor`	Floating-point Scalar	Floating-point Scalar	Floating-point Scalar
`fmod`	Floating point	Scalar	Real
`hypot`[3]	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real
`log`[2]	Floating point	Scalar Vector Matrix	Real Complex
`log10`[2]	Floating point	Scalar Vector Matrix	Real Complex
`log2`[3]	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point	Scalar	Real
`min`	Integer Floating point	Scalar	Real
`pow`	Floating point	Scalar	Real
`sec`[3]	Floating point	Scalar Vector Matrix	Real Complex
`secd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`sech`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`sin`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sind`[3]	Floating point	Scalar Vector Matrix	Real Complex
`sinh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sqrt`	Floating point	Scalar	Real
`tan`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`tand`[3]	Floating point	Scalar Vector Matrix	Real Complex
`tanh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

[1] Wrap on integer overflow only.

[2] For models involving vectors or matrices, the code generator replaces only functions coded in the MATLAB action language.

[3] The code generator replaces only functions coded in the MATLAB action language.

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`memcmp`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`memcpy`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset2zero`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Nonfinite Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following nonfinite functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`getInf`	Floating point	Scalar	Real
`getMinusInf`	Floating point	Scalar	Real
`getNaN`	Floating point	Scalar	Real
`rtIsInf`	Floating point	Scalar	Real Complex
`rtIsNaN`	Floating point	Scalar	Real Complex

Mutex and Semaphore Functions

Mutex and semaphore functions control access to resources shared by multiple processes in multicore target environments. MathWorks provides code replacement libraries that support mutex and semaphore replacement for Rate Transition and Task Transition blocks on Windows, Linux, Mac, and VxWorks platforms.

Generated mutex and semaphore code typically consists of:

- In model initialization code, an initialization function call to create a mutex or semaphore to control entry to a critical section of code.
- In model step code:
  - Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls to reserve a critical section of code.
  - After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls to release the critical section of code.

- In model termination code, an optional destroy function call to explicitly delete the mutex or semaphore.

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following mutex and semaphore functions with application-specific implementations.

Function	Key
Mutex Destroy	`RTW_MUTEX_DESTROY`
Mutex Init	`RTW_MUTEX_INIT`
Mutex Lock	`RTW_MUTEX_LOCK`
Mutex Unlock	`RTW_MUTEX_UNLOCK`
Semaphore Destroy	`RTW_SEM_DESTROY`
Semaphore Init	`RTW_SEM_INIT`
Semaphore Post	`RTW_SEM_POST`
Semaphore Wait	`RTW_SEM_WAIT`

Operators

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates that you can specify different data types for different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)[1]	`RTW_OP_ADD`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)[1]	`RTW_OP_MINUS`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*)[2]	`RTW_OP_MUL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	`RTW_OP_DIV`	Integer Floating point Fixed-point Mixed	Scalar	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Data type conversion (cast)	`RTW_OP_CAST`	Integer Floating point[3] Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	`RTW_OP_SL`	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real
Shift right arithmetic (>>)[5]	`RTW_OP_SRA`	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real
Shift right logical (>>)	`RTW_OP_SRL`	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real
Element-wise matrix multiplication (`.*`)[6]	`RTW_OP_ELEM_MUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix right division (`/`)	`RTW_OP_RDIV`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix left division (`\`)	`RTW_OP_LDIV`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix inversion (`inv`)	`RTW_OP_INV`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	`RTW_OP_CONJUGATE`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (`.'`)	`RTW_OP_TRANS`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (`'`)	`RTW_OP_HERMITIAN`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Multiplication with transposition[2]	`RTW_OP_TRMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition[2]	`RTW_OP_HMMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication followed by shift right arithmetic (`u1*u2>>u3`)[7]	`RTW_OP_MUL_SRA`	Integer Fixed-point	Scalar	Real
Multiplication followed by division (`u1*u2/u3`)[8]	`RTW_OP_MULDIV`	Integer Fixed-point	Scalar	Real
Greater than (>)	`RTW_OP_GREATER_THAN`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal (>=)	`RTW_OP_GREATER_THAN_OR_EQUAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	`RTW_OP_LESS_THAN`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (<=)	`RTW_OP_LESS_THAN_OR_EQUAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	`RTW_OP_EQUAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (`!=`)	`RTW_OP_NOT_EQUAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
[1] See “Replace Addition and Subtraction Operator Code” on page 72-153 for details to consider when defining mappings for addition and subtraction code replacements. [2] Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions. [3] Scaled floating point is not supported. [4] Shift operator replacement with matrix data is supported for shift values that you specify with an input port. Replacement is not supported for shift values that you specify in a block parameter dialog. [5] The code generator converts some arithmetic shift rights to logical shift rights. To avoid unexpected results, when creating a code replacement library that includes a table entry for an arithmetic shift right implementation, also include an entry for a logical shift right implementation. [6] Use the multiplication (*) operator (`RTW_OP_MUL`) for scalar multiplication. [7] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; shift operand is an unsigned integer; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope` and `Mul_output_slope == output_slope_of_shift_operation`). [8] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope == U3_slope * Div_output_slope`).				

See Also

More About

- “Lookup Table Function Code Replacement” on page 72-94
- “Develop a Code Replacement Library” on page 72-20
- “Quick Start Code Replacement Library Development - Simulink” on page 72-21
- “What Is Code Replacement?” on page 56-2

Develop a Code Replacement Library

Iterate through the following steps, as necessary, to develop a code replacement library:

1. “Identify Code Replacement Library Requirements” on page 72-35
2. “Define Code Replacement Library Optimizations” on page 72-36
3. “Register Code Replacement Library” on page 72-51
4. “Verify Code Replacement Library” on page 72-55
5. “Deploy Code Replacement Library” on page 72-57

To experiment with the process and tools, see “Quick Start Code Replacement Library Development - Simulink” on page 72-21.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Identify Code Replacement Library Requirements” on page 72-35
- “Quick Start Code Replacement Library Development - Simulink” on page 72-21
- “What Is Code Replacement Customization?” on page 72-3

Quick Start Code Replacement Library Development - Simulink

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the Code Replacement Tool.

Prerequisites

To complete this example, install the following software:

- MATLAB®
- MATLAB Coder™
- Simulink®
- Simulink Coder™
- Embedded Coder®
- Compiler

For instructions on installing MathWorks® products, see the MATLAB installation documentation. If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

For a list of supported compilers, see `https://www.mathworks.com/support/compilers/current_release/`.

Open the Code Replacement Tool

1. Start a MATLAB session.

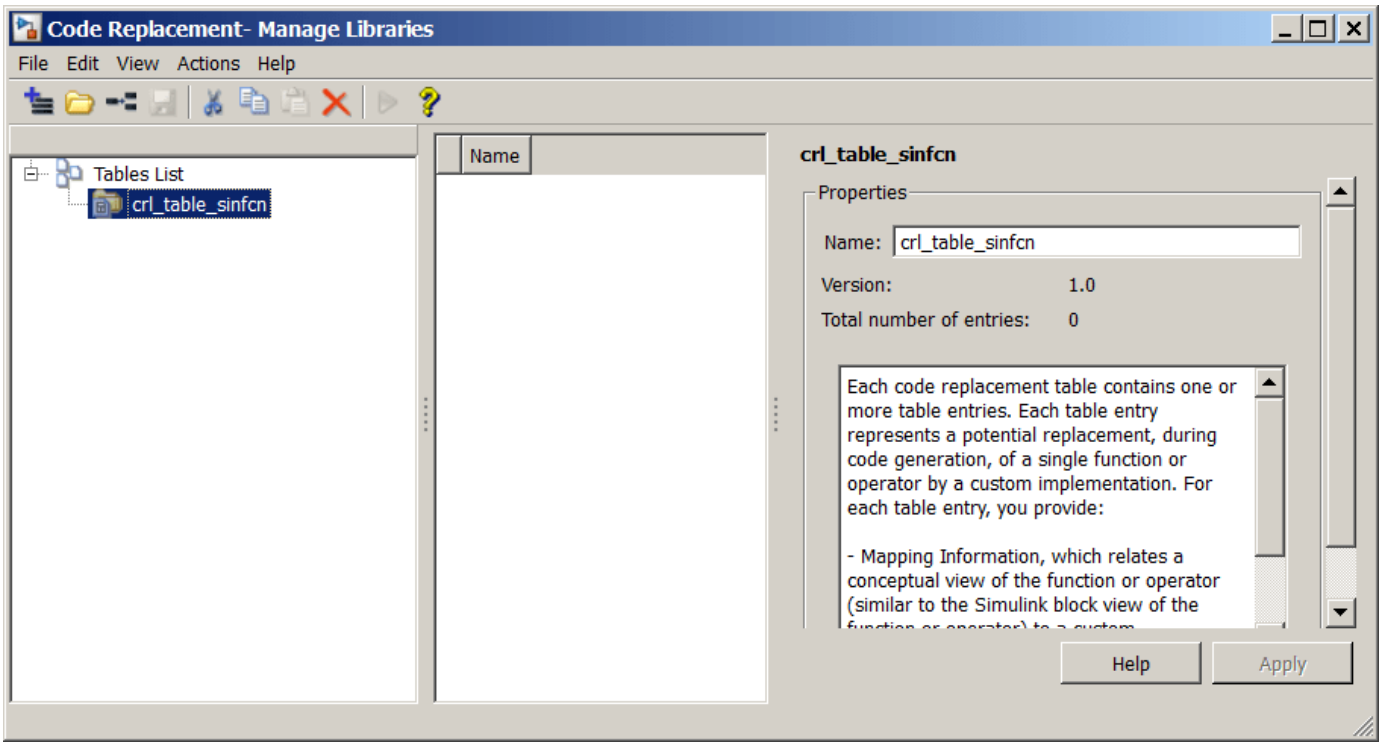
2. Create or navigate (`cd`) to an empty folder.

3. At the command prompt, enter the `crtool` command. The Code Replacement Tool window opens.

Create Code Replacement Table

1. In the Code Replacement Tool window, select **File > New table**

2. In the right pane, name the table `crl_table_sinfcn` and click **Apply**. When you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.



Create Table Entry

Create a table entry that maps a `sin` function with `double` input and `double` output to a custom implementation function.

1. In the left pane, select table `crl_table_sinfcn`. Then, select **File > New entry > Function**. The entry appears in the middle pane, initially without a name.

2. In the middle pane, select the new entry.

3. In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.

4. Leave **Algorithm** set to `Unspecified`, and leave parameters in the **Conceptual function** group set to default values.

5. In the **Replacement function** group, name the replacement function `sin_dbl`.

6. Leave the remaining parameters in the **Replacement function** group set to default values.

7. Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.

8. Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.

Mapping Information | Build Information

Function: sin

Entry information

Algorithm: Unspecified

Conceptual function

Used by code generation process for matching purposes

Conceptual arguments

y1
u1

Argument properties

Data type: double

Complex

Argument type: Scalar

Make conceptual and implementation argument types the same

Replacement function

Function prototype

Name: sin_dbl

C++ namespace:

Function returns void

Function arguments

y1(return arg)
u1

Argument properties

Data type: double

I/O type: OUTPUT

Const

Pointer

Complex

+

Function signature preview

double *sin_dbl* (double u1);

Implementation attributes

Integer saturation mode: Unspecified Saturation

Rounding mode:

Unspecified Rounding
Floor
Ceil

Allow expressions as inputs

Function modifies internal or global state

Click here to add Build Information

Validation

Validate entry

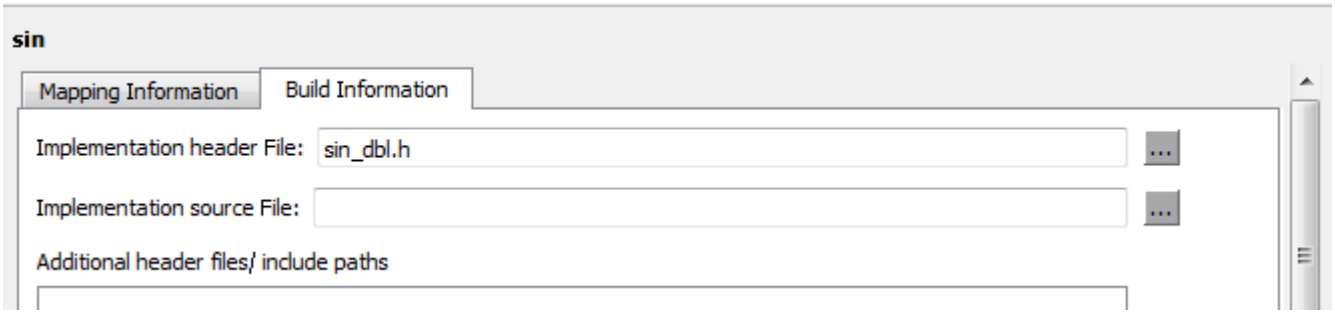
Status: *Validated*

Specify Build Information for Replacement Code

1. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.

2. Leave the remaining parameters set to default values.

3. Click **Apply**.



4. Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

1. In the middle pane, select the `sin_dbl` entry.

2. Select **Edit > Copy**.

3. Select **Edit > Paste**.

4. On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.

5. In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.

6. Click **Apply**. Note the changes that appear for the **Function signature preview**.

7. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

1. Select **Actions > Validate table**.

2. If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.

Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
sin	sin_dbl	1	double		double		100
sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File > Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `crl_table_sinfcn.m`.

Review the Code Replacement Table Definition

Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

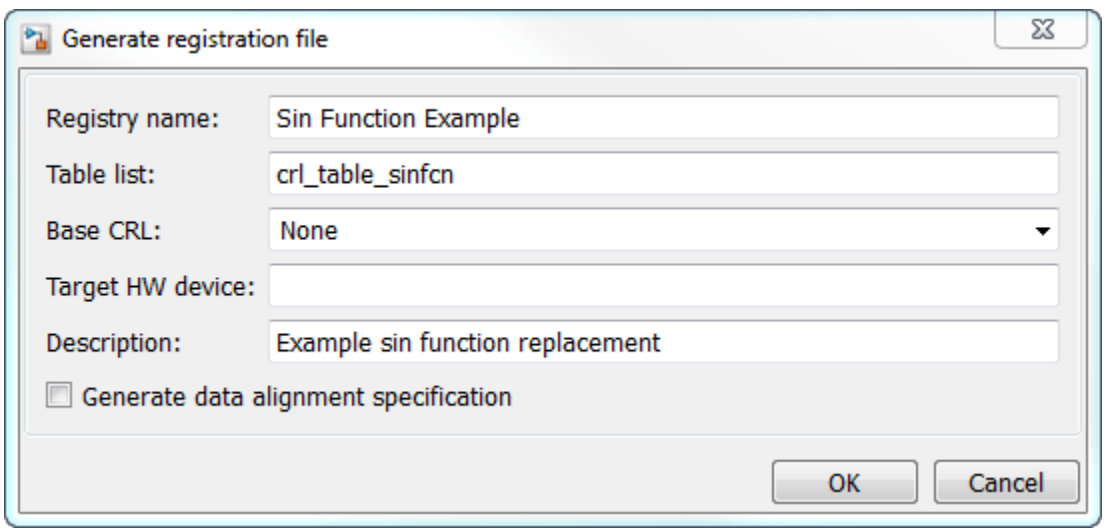
To review it, in MATLAB or another text editor, open the file `crl_table_sinfcn.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

1. In the Code Replacement Tool, select **File > Generate registration file**.

2. In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and click **OK**.



3. In the **Select location to save the registration file** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

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

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

1. Check for errors. At the command line, invoke the table definition file. For example:

tbl = crl_table_sinfcn

tbl =

```
TflTable with properties:

               Version: '1.0'
       ReservedSymbols: []
   StringResolutionMap: []
            AllEntries: [2x1 RTW.TflCFunctionEntry]
           EnableTrace: 1|
```

If an error exists in the definition file, the invocation triggers a message. Fix the error and try again.

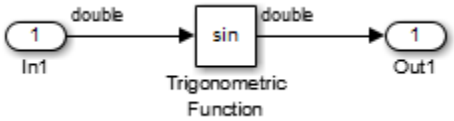
2. Use the Code Replacement Viewer to check your code replacement entries. For example:

```
crviewer('Sin Function Example')
```

In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

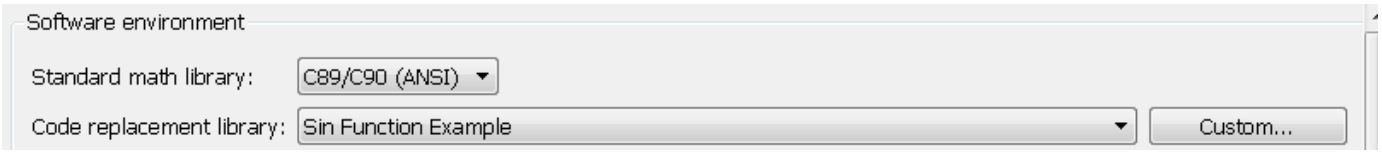
- Incorrect argument order.
- Conceptual argument names that do not match what the code generator expects.
- Incorrect priority settings.

3. Identify existing model or create a model that includes a Trigonometric block that is set to the `sin` function. For example:



4. Open the model and configure it for code generation with an Embedded Coder (ERT-based) target.

5. See whether your library is listed as an available option for the **Code Generation > Interface > Code replacement library** model configuration parameter. If it is, select it.



If it is not listed, open the registration file, `rtwTargetInfo.m`. See whether you entered the correct code replacement table name when you created the file. If you hover the cursor over the selected library, a tool tip appears. This tip contains information derived from your code replacement library registration file, such as the library description and the list of tables it contains.

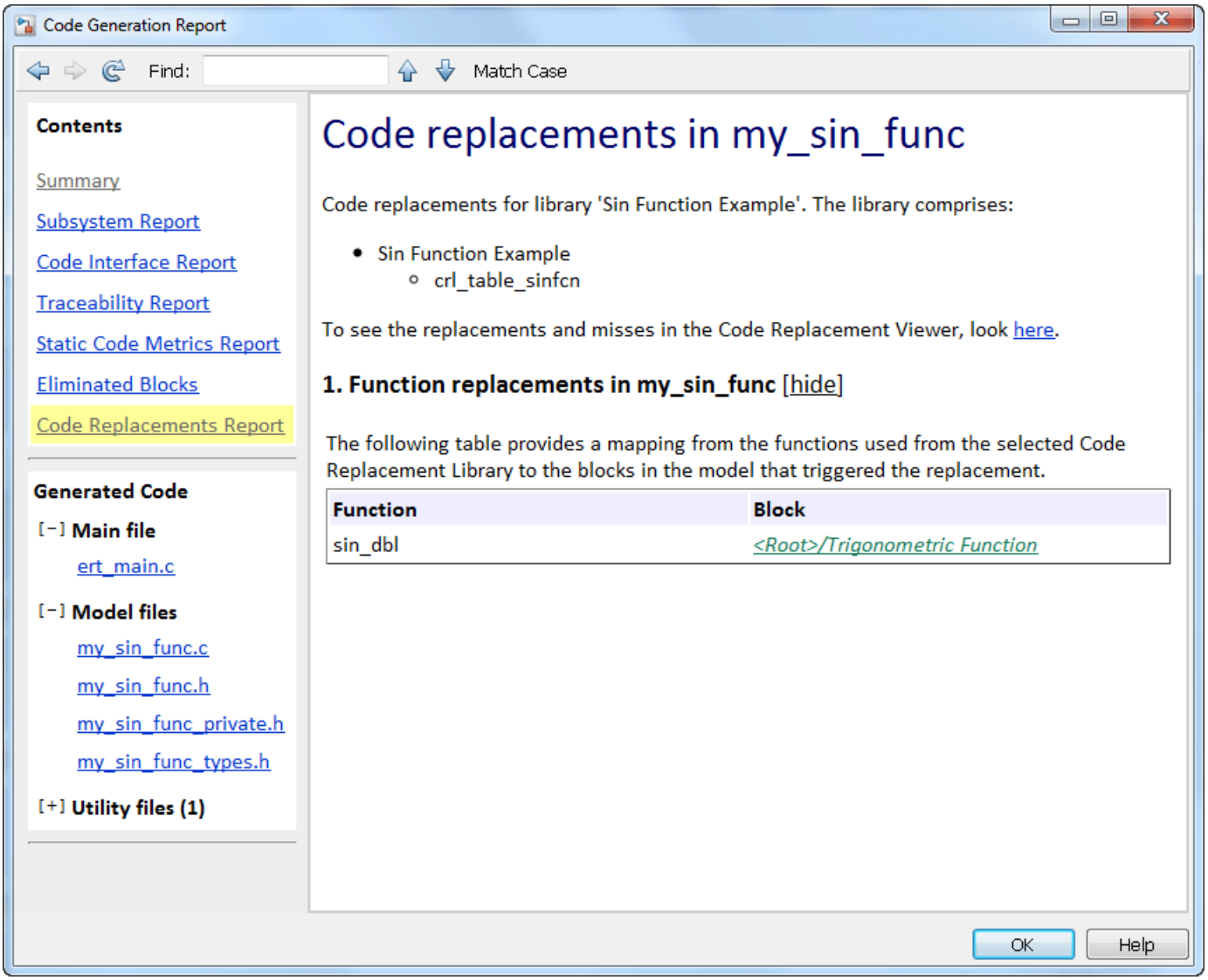
6. To find parameters quickly, in the Configuration Parameters dialog box **Search** field, type the parameter name. Configure the code generation report for code replacement analysis by selecting the following parameters:

- **Create code generation report**
- **Open report automatically**
- **Code-to-model**
- **Model-to-code**
- **Summarize which blocks triggered code replacements**
- **Include comments**
- **Simulink block comments**
- **Simulink block descriptions**

7. Configure the model to generate code only. Before you build an executable program, confirm that the code generator is replacing code as expected.

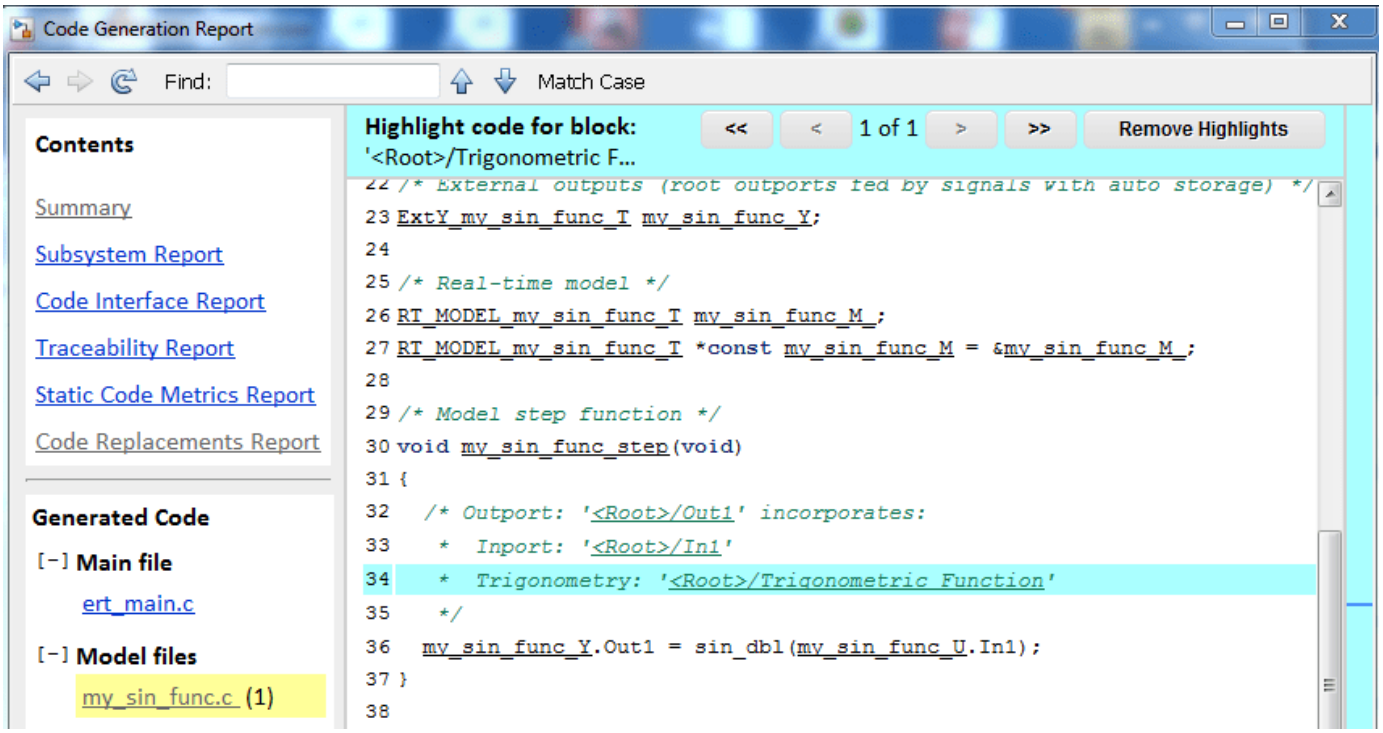
8. Generate code for the model.

9. Review code replacement results in the Code Replacement Report section of the code generation report.



The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.

10. Review the code replacements. In the model window, right-click the Trigonometric Function block. Select **C/C++ Code > Navigate to C/C++ Code**. The code generation report opens and highlights the code replacement in `my_sin_func.c`. In this case, the code generator replaced `sin` with `sin_dbl`.



More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

Semaphore and Mutex Function Replacement

You can create a code replacement table for a custom target that supports concurrent execution. Create table entries that specify custom implementations of semaphore or mutex operations. The table must have four semaphore entries, four mutex entries, or both, and include the table in a custom code replacement library. (The semaphore or mutex entries are mutually dependent. Provide them in complete sets of four.)

Note A custom target that supports concurrent multitasking must set the target configuration parameter `ConcurrentExecutionCompliant`. For more information, see "Support Concurrent Execution of Multiple Tasks".

If the build process generates semaphore or mutex function calls for data transfer between tasks during code generation for a multicore target environment, use a custom library. The library can specify code replacements for custom semaphore or mutex implementations that are optimal for your target environment. Using the Code Replacement Tool (`crtool`) or equivalent code replacement functions, you can:

- Configure code replacement table entries for custom semaphore or mutex functions. During system startup, execution of the code for data transfer between tasks, and system shutdown the generated code calls these functions.
- Configure DWork arguments that represent global data, which the semaphore or mutex functions access. A DWork pointer is passed to the model entry functions.

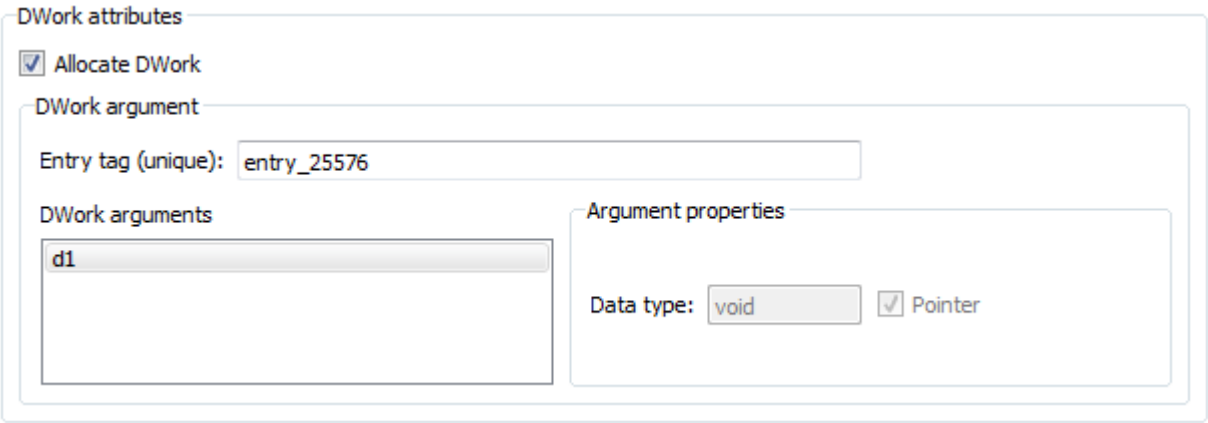
Generated mutex and semaphore code typically consists of these elements:

Code	Generated Code
Model initialization	Initialization function call that creates a mutex or semaphore function to control entry to a critical section of code.
Model step	• Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls reserve the critical section of code. • After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls release the critical section of code.
Model termination	Optional destroy function call to delete the mutex or semaphore explicitly.

This example shows how to create code replacement table entries for a mutex replacement scenario. You configure a multicore target model for concurrent execution and for data transfer between tasks of differing rates, which Rate Transition blocks handle. In the generated code for the model, each Rate Transition block has a separate, unique mutex. Mutex lock and unlock operations within the Rate Transition block generated code share access to the same global data. They achieve this by using the unique mutex created for that Rate Transition block.

1. Open the **Code Replacement Tool**.
2. Create and open a new table.

3. Name the table `crl_table_rt_mutex`.
4. Create an entry for a mutex initialization function replacement.
   a. Select **File** > **New entry** > **Semaphore entry** to open a new table entry for configuring a semaphore or mutex replacement.
   b. In the **Mapping Information** tab, use the **Function** parameter to select `Mutex Init`. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.
   c. In the **DWork attributes** section, the **Allocate DWork** option is selected. The dialog box provides a unique entry tag for the DWork argument `d1`.



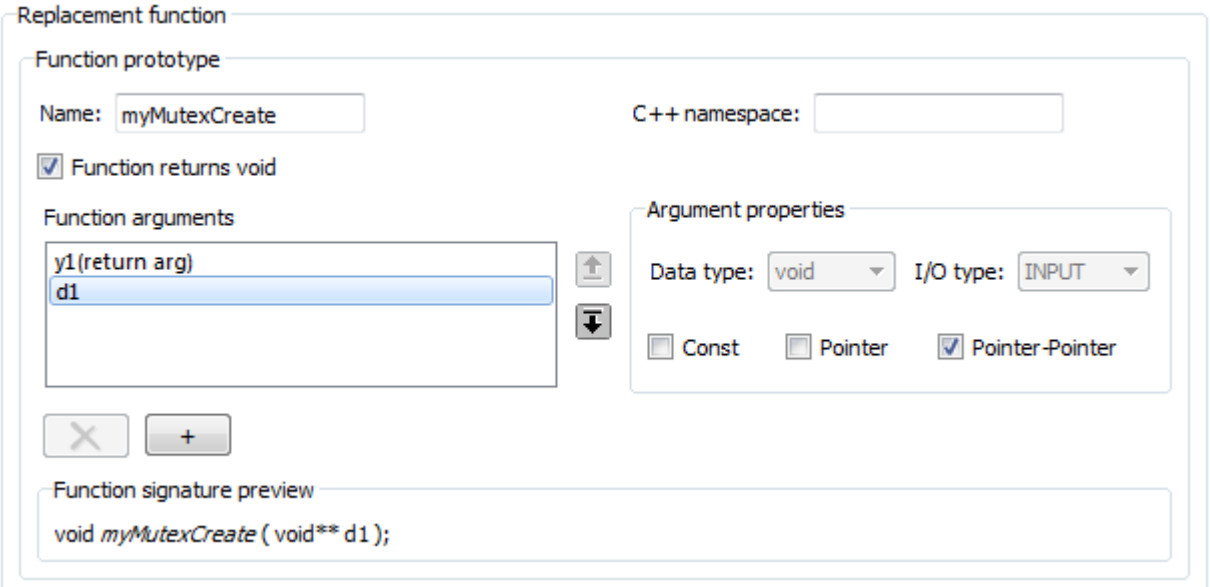
   On the **DWork attributes** pane, configure a DWork argument to the replacement function. The DWork argument supports sharing of a semaphore or mutex between:

   - Code that creates the semaphore or mutex
   - Code that requests and relinquishes access
   - Code that deletes the semaphore or mutex

   In this example, the DWork argument for the `Mutex Init` function defines a unique entry tag, `entry_25576`. That function also defines DWork arguments for `Mutex Lock`, `Mutex Unlock`, and `Mutex Destroy`, which reference the entry tag to share the DWork data.

   The only data type supported for the DWork **Data type** parameter is `void*`.

   d. In the **Replacement function** section, enter a function name in the **Name** field. This example uses `myMutexCreate`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void**`.



The C function signature preview is:

```
void myMutexCreate (void** d1);
```

e In the **Replacement function** section, select **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values. Click **Apply**. Optionally, you can click **Validate entry** to validate the information entered in the **Mapping Information** tab.

To create a sample table entry, configure the replacement function signature without the replacement function and its build information. If header and source files for these functions are available, select the **Build Information** table to specify them.

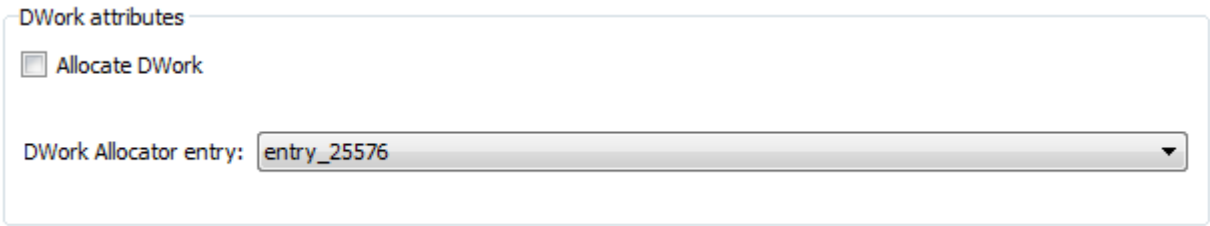
f The `Mutex Init` table entry is complete. Optionally, you can save the table to a file, and inspect the MATLAB code created for the table definition so far.

5 Repeat the following sequence to create the table entries for the mutex lock, unlock, and destroy function replacements. Each table entry references the DWork unique tag entry, `entry_25576`, defined in the `Mutex Init` table entry.

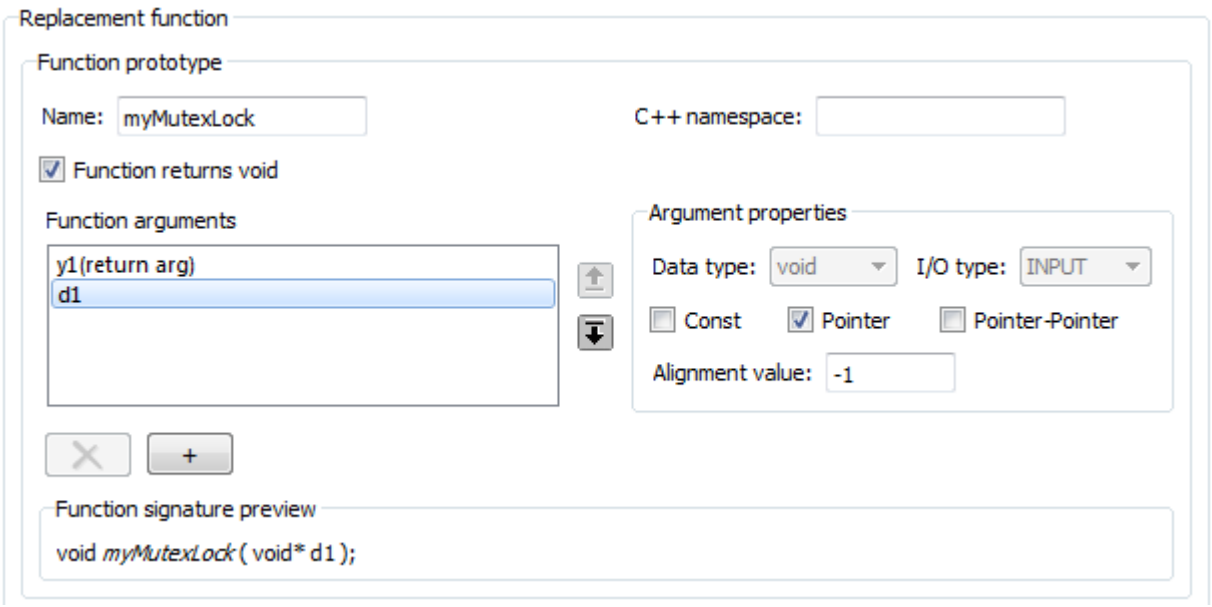
a Select **File** > **New entry** > **Semaphore entry**.

b In the **Mapping Information** tab, use the **Function** parameter to select `Mutex Lock`, `Mutex Unlock`, or `Mutex Destroy`. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.

c For a Rate Transition block mutex, the wait, post, and destroy functions operate on the DWork allocated at system startup by the mutex initialization function. In the **DWork attributes** section, verify that the **Allocate DWork** option is cleared. From the **DWork Allocator entry** drop-down list, select the entry tag matching the value in the `Mutex Init` table entry. In this example, the entry tag is `entry_25576`.



d In the **Replacement function** section, **Name** field, enter a function name. This example uses `myMutexLock`, `myMutexUnlock`, and `myMutexDelete`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void*`.

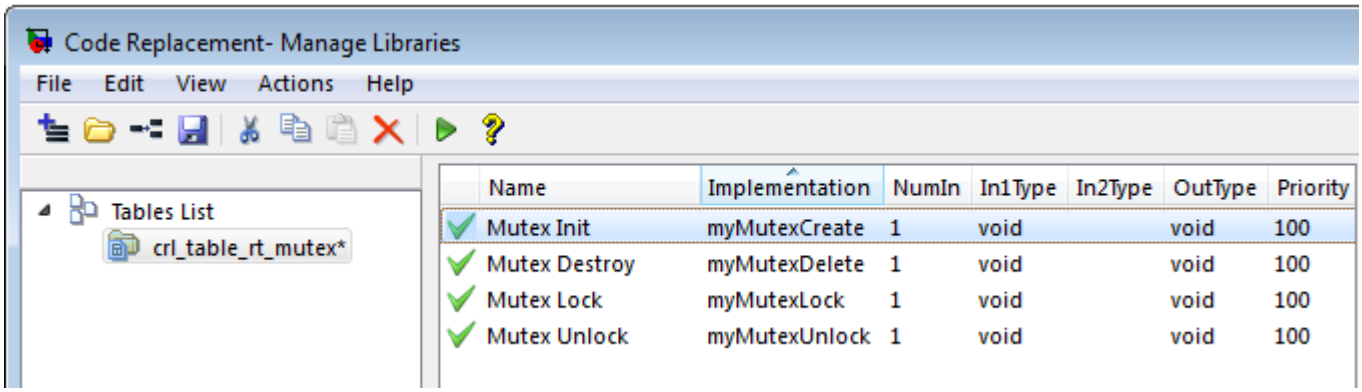


e In the **Implementation attributes** section, select the option **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values.

f Optionally, supply build information for the replacement function on the **Build Information** tab.

g Click **Apply**. In the middle pane, right-click the table entry and select **Validate entry(s)**.

6 When you have added the table entries for `Mutex Lock`, `Mutex Unlock`, and `Mutex Destroy` to the entry for `Mutex Init`, the rate transition mutex replacement table is complete. In the left-most pane, right-click the table name and select **Validate table**. Address errors and repeat the table validation.

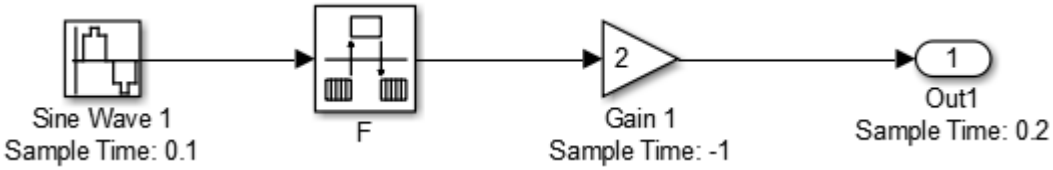


7 Save the table to a MATLAB file in your working folder, for example, using **File** > **Save table**. The name of the saved file is the table name, `crl_table_rt_mutex`, with an `.m` extension. Optionally, you can open the saved file and examine the MATLAB code for the code replacement table definition.

To test this example:

1 Register the code replacement mapping.

2 Create a model that contains a rate transition for which the build process generates mutex function calls. For example:



3 Configure the model for a multicore target environment and the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your mutex entry.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Define Code Replacement Library Optimizations” on page 72-36
- “Data Alignment for Code Replacement” on page 72-105
- “Reserved Identifiers and Code Replacement” on page 72-131
- “Customize Match and Replacement Process” on page 72-132
- “Develop a Code Replacement Library” on page 72-20

Identify Code Replacement Library Requirements

What Is a Code Replacement Library?

A code replacement library defines optimizations for generated C or C++ code from Simulink models or MATLAB code for specific targets. You can configure the code generator to use your code replacement library to replace specified functions in your model or MATLAB code with optimized implementations.

What Functions Can Be Optimized with a Code Replacement Library?

You can use a code replacement library to optimize the performance of functions defined either by Simulink models or MATLAB code. To look up functions that are available to optimize, see “Code You Can Replace From Simulink Models” on page 72-6 and “Code You Can Replace from MATLAB Code” on page 73-5.

How to Develop a Code Replacement Library

You can develop a code replacement library either interactively, using the **Code Replacement Tool** (crtool), or programmatically, using the MATLAB programming interface. The Code Replacement Tool is quick, easy, and ideal for first time library development, while the MATLAB Programming interface provides more customization and control options. To determine which approach to use, identify your use case below:

Interactive Approach:

- Developing a code replacement library for the first time
- Rapid prototyping

Programmatic Approach:

- Specifying code replacement attributes not available with the Code Replacement Tool
- Creating or modifying code replacement library entries by copying, pasting, and editing
- Modifying a registration file by copying and pasting content

Identify Basic Requirements

- What model or MATLAB code do you want to optimize by using a code replacement library?
- What functions and operations within your model or MATLAB code do you want to replace? What is the signature for each function or operation that you want to replace?
- What build files (source, header, object) do you need to provide to the code generator?

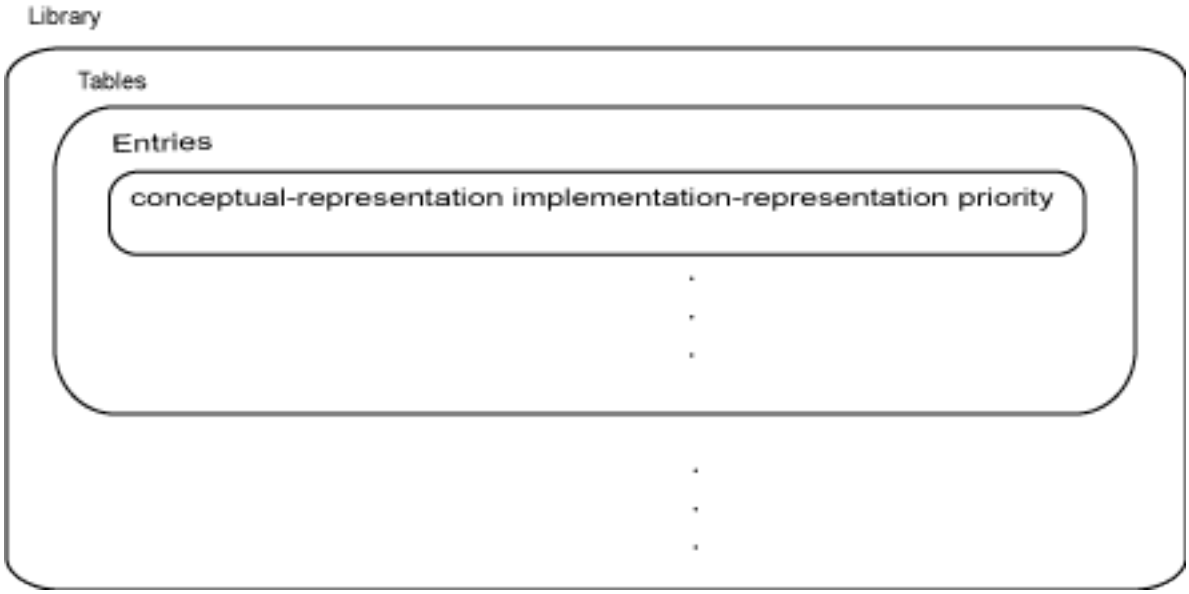
See Also

More About

- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

Define Code Replacement Library Optimizations

A code replacement library contains one or more code replacement tables. Each table contains one or more code replacement entries. Each entry contains a mapping and entry parameters. The entry maps the conceptual representation of an automatically generated MATLAB or Simulink function to an implementation representation of an optimized C or C++ function. The entry parameters describe additional properties for the code generator to honor during code replacement.



To define a code replacement library, first define code replacement tables in a customization file. Then, during the registration step, create a code replacement library by selecting tables to include in your library. You can define your code replacement tables both interactively or programmatically using the same work flow.

The interactive and programmatic approaches both follow these 8 steps:

1. Open the environment.
2. Create a table.
3. Create an entry.
4. Create entry parameters.
5. Create the conceptual representation.
6. Create the implementation representation.
7. Specify build information.
8. Validate and save.

Interactively Develop a Code Replacement Library

1. Open the environment.

   To interactively create a code replacement library, open the **Code Replacement Tool** (crtool) from the MATLAB command line with the command:

```
>> crtool
```

2. Create a table.

   A code replacement table organizes code replacement entries into groupings that are easier to manage.

   To create a code replacement table:

   a. From the crtool menu, select **File** > **New table**.
   b. In the right pane, name the table and click **Apply**.

3. Create an entry.

   A code replacement table entry contains entry parameters and a mapping of a conceptual representation that describes a generated function to an implementation representation that describes an optimized implementation of that function.

   To create a code replacement entry, use one of these methods:

   - From the crtool toolbar, click the **entry-type** icon the select the **entry-type**.
   - From the crtool menu, select **File** > **New entry** > **entry-type**.
   - Right-click on a table and select **New entry** > **entry-type**.

   The new entry appears without a name in the middle pane.

4. Create entry parameters.

   Entry parameters specify optimization and build requirements for the code generator to honor during code replacement. These parameters vary and automatically appear in various sections of the crtool interface depending on the type of function replacement.

   To create entry parameters:

   - In the **Function** or **Operation** drop-down list, select the name of the function you want to replace.

   This selection determines the entry parameters that appear in the crtool.

5. Create the conceptual representation.

   The conceptual representation describes the signature of the function you want to replace. The conceptual representation consists of the function name, return arguments, and parameters. The return arguments are referred to as output arguments, and the parameters are referred to as input arguments. Specify each conceptual argument with a name, I/O type, and, depending on the type of replacement, other properties that appear in the crtool.

   To define your conceptual representation:

   - In the **Conceptual Function** or **Conceptual Operation** subsection of the crtool, specify the input and output arguments of the function you want the code generator to replace.

6. Create the implementation representation.

   The implementation representation describes the signature of the replacement function defined in the code replacement library. The implementation representation consists of a function name, return arguments, and parameters. Specify each implementation argument with a type, type qualifier, and complexity.

To define your implementation representation:

- In the **Implementation Function** or **Operation Function** subsection of the crtool, specify the input and output arguments for the optimized function defined in the code replacement library.

7 Specify build information.

Build information provides the code generator with the files to perform function replacement.

To specify build information:

- In the crtool, click the **Build Information** tab to open the build requirements pane. Specify necessary build information in these fields:

Parameter	Description
Implementation header file	Name of header file needed for code replacement.
Implementation source file	Name of source file needed for code replacement.
Additional header files/include paths	Name and path of additional header files needed for code replacement.
Additional source files/paths	Name and path of additional source files needed for code replacement.
Additional object files/paths	Name and path of additional object files needed for code replacement.
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build folder	Flags whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as packNGo might find duplicate instances of files.

8 Validate and save.

Save your code replacement library with the same name as your code replacement table on your MATLAB path. Saving your library automatically validates unvalidated content.

To validate and save your library:

- From the crtool menu, select **File** > **Save Table** > **Save**.

Programmatically Develop a Code Replacement Library

1. Open the environment.

   To programmatically create a code replacement library, open a MATLAB function file. From the MATLAB menu select **New** > **Function**.

2. Create a table.

   A code replacement table organizes code replacement entries into groupings that are easier to manage.

   To create a code replacement table:

   a. Create a function you can use to call your code replacement library table. The function should not have arguments and return a table object.

   b. Create a table object with a call to `RTW.TflTable`.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

end
```

3. Create an entry.

   A code replacement table entry contains entry parameters and a mapping of a conceptual representation of a generated function to the implementation representation of an optimized implementation of that function.

   To create a code replacement table:

   a. Identify the type of function you want to replace.

   b. Call the corresponding Entry Function to create an entry in your table.

Type of Function You Want to Replace	Entry Function
Math Operation	`RTW.TflCOperationEntry`
Function	`RTW.TflCFunctionEntry`
BLAS operation	`RTW.TflBlasEntryGenerator`
CBLAS operation	`RTW.TflCBlasEntryGenerator`
Fixed-point addition and subtraction operations	`RTW.TflCOperationEntryGenerator`
Net slope fixed-point operation	`RTW.TflCOperationEntryGenerator_NetSlope`
Semaphore or mutex	`RTW.TflCSemaphoreEntry`
Custom function	`RTW.TflCFunctionEntryML`
Custom operation	`RTW.TflCOperationEntryML`

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table
```

```
%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

end
```

4 Create entry parameters.

Entry parameters specify optimization and build requirements for the code generator to honor during code replacement. These parameters vary depending on the type of function replacement.

To create your entry parameters:

- Call the Set Parameters function that corresponds with the type of function you want to replace identified in the previous step (step 3).

Type of Function You Want to Replace	Set Parameters function
Math operation, BLAS operation, CBLAS operation, Fixed-Point, Net Slope Fixed-Point, Custom operation	`setTflCOperationEntryParameters`
Function, Custom function	`setTflCFunctionEntryParameters`
Semaphore or Mutex	`setTflCSemaphoreEntryParameters`

Each Set Parameter function is shown with all available properties set to their default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;
```

```
%% Create entry parameters
% Pick one from list below
hEntry.setTflCOperationEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ...
        'AllowShapeAgnostic',           false, ...
        'ArrayLayout',                  'COLUMN_MAJOR', ...
        'EntryInfoAlgorithm',           'RTW_CAST_BEFORE_OP', ...
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'MustHaveZeroNetBias',          false, ...
        'NetFixedExponet',              0, ...
        'NetAdjustmentFactor', 1,       ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
        'SlopesMustBeTheSame',          false, ...
        'StoreFcnReturnInLocalVar',     false);

hEntry.setTflCFunctionEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ...
        'ArrayLayout',                  'COLUMN_MAJOR', ...
        'EntryInfoAlgorithm',           'RTW_DEFAULT', ...
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
```

```
        'StoreFcnReturnInLocalVar',     false);

hEntry.setTflCSemaphoreEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ....
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
        'StoreFcnReturnInLocalVar',     false);

end
```

5 Create the conceptual representation.

The conceptual representation describes the signature of the function you want to replace. It consists of the function name, return arguments, and parameters. The return arguments are referred to as output arguments, and the parameters are referred to as input arguments. Specify each conceptual argument with the required properties, and, depending on the type of replacement, other optional properties.

Required properties:

- **Name**- Defined as y1..yn for output arguments and u1..un for input arguments
- **I/O type**- Defined as `RTW_IO_OUTPUT` or `RTW_IO_INPUT`

To define your conceptual representation, use one of these approaches:

Approach	Function	Goal
1	`createAndAddConceptualArg`	Customization and control. If you want to explicitly specify arguments as scalar or matrix and other properties.
2	`getTflArgFromString`	Rapid prototyping. If you want to quickly specify arguments with built-in data types.

Programming approaches are shown with all available properties set to default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

%% Create entry parameters
% Pick one from list below
% To view properties, see step 4
hEntry.setTflCOperationEntryParameters( );
hEntry.setTflCFunctionEntryParameters( );
hEntry.setTflCSemaphoreEntryParameters( );

%% Create the conceptual representation

% Approach 1: createAndAddConceptualArg
% arg = createAndAddConceptualArg(hEntry, argType, varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type

% Define Output Arguments
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'y1', ...
        'IOType',                       'RTW_IO_OUTPUT', ...
        'IsSigned',                     true, ...
        'WordLength',                   32, ...
        'CheckSlope',                   true, ...
        'CheckBias',                    true, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'BaseType',                     'double', ...
        'DimRange',                     [2,2]);

% Define Input Arguments
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'u1', ...
        'IOType',                       'RTW_IO_INPUT', ...
```

```
        'IsSigned',                     true, ...
        'WordLength',                   32, ...
        'CheckSlope',                   true, ...
        'CheckBias',                    true, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'BaseType',                     'double', ...
        'DimRange',                     [2,2]);


% Approach 2: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, I/O type)
% name- y1..yn | u1..un
% I/O type- RTW_IO_OUTPUT | RTW_IO_INPUT

% Define Output Arguments
arg = getTflArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

% Define Input Arguments
arg = getTflArgFromString('u1', 'double');
arg.IOType = 'RTW_IO_INPUT';
hEntry.addConceptualArg(arg);

end
```

6 Create the implementation representation.

The implementation representation describes the signature of the replacement function defined by the code replacement library. It consists of the function name, return arguments, and parameters. Specify the implementation arguments with a type, type qualifier, and complexity. After you define your implementation representation, add your completed entry to the code replacement table with a call to the function `addEntry`.

To define your implementation representation, use one of these approaches:

Approach	Function	Goal
1	`copyConceptualArgsToImplementation`	Identical arguments. Specify implementation arguments and conceptual arguments have the same properties.
2	`createAndAddImplementationArg` `createAndSetCImplementationReturn`	Same order, different properties. Specify implementation arguments have the same order as conceptual arguments but have different properties.

Approach	Function	Goal
3	`getTflArgFromString`	Add constant value. Specify implementation arguments with a constant value.
4	`getTflDWorkFromString`	Mutex/Semaphore replacement. Specify implementation arguments for mutex or semaphore functions.
5	`createCRLEntry`	Create entry with one function. Declare and define a code replacement table entry (conceptual and implementation arguments) in one function.

Programming approaches are shown with all available properties set to their default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

%% Create entry parameters
% Pick one from list below
% To view properties, see step 4
hEntry.setTflCOperationEntryParameters( );
hEntry.setTflCFunctionEntryParameters( );
hEntry.setTflCSemaphoreEntryParameters( );

%% Create the conceptual representation

% Approach 1: createAndAddConceptualArg
% arg = createAndAddConceptualArg(hEntry, argType, varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type, to view properties, see step 5

% Define Output Arguments
hEntry.createAndAddConceptualArg( );
```

```
% Define Input Arguments
hEntry.createAndAddConceptualArg( );



% Approach 2: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, I/O type)
% name- y1..yn | u1..un
% I/O type- RTW_IO_OUTPUT | RTW_IO_INPUT

% Define Output Arguments
arg = getTflArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

% Define Input Arguments
arg = getTflArgFromString('u1', 'double');
arg.IOType = 'RTW_IO_INPUT';
hEntry.addConceptualArg(arg);



%% Create the implementation representation

% Approach 1: copyConceptualArgsToImplementation
copyConceptualArgsToImplementation(hEntry);


% Approach 2: createAndSetCImplementation, createAndSetCImplementationArg
% arg = createAndAddImplementation(hEntry, argType,varargin),
% arg = createAndSetCImplementationReturn(hEntry, argType,varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type

% Define Output Arguments
hEntry.createAndSetCImplementationReturn(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'y1', ...
        'IOType',                       'RTW_IO_OUTPUT', ...
        'IsSigned',                     true, ...
        'WordLength',                   16, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1.0, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'Value',                        0);

% Define Input Arguments
hEntry.createAndAddImplementationArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'u1', ...
        'IOType',                       'RTW_IO_INPUT', ...
```

```
        'IsSigned',                     true, ...
        'WordLength',                   16, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1.0, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'Value',                        0);


% Approach 3: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, datatype)
% name- y1..yn | u1..un, datatype - built-in data type

% Define Output arguments
arg = getTflArgFromString(hEntry, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

% Define Input arguments
arg = getTflArgFromString(hEntry, 'u1', 'double', 0);
hEntry.Implementation.addArgument(arg);



% Approach 4: getTflDWorkFromString
% arg = getTflDWorkFromString(hEntry, name, datatype)

% Define arguments
arg = getTflDWorkFromString('d1', 'void*');
hEntry.addDWorkArg(arg);



% Approach 5: createCRLEntry
% tableEntry = createCRLEntry(crTable,conceptualSpecification,implementationSpecification)

% Define code replacement conceptual and implementation arguments
hEntry = createCRLEntry(...
        hTable, ...
        'conceptualSpecification', ...
        'implementationSpecification');


%% Add code replacement library entry to table
hTable.addEntry(hEntry);

end
```

7 Specify build information.

Specify build information for your code replacement library. Build information provides the code generator with the files necessary to perform function replacement.

To specify build information:

- Specify your build parameters in the Entry Parameters function you defined in step 4 or with the specified functions. Specify build information with the following properties:

Parameter	Description
Implementation header file Set properties `ImplementationHeaderFile` and `ImplementationHeaderPath`.	Name of header file needed for code replacement.
Implementation source file Set properties `ImplementationSourceFile` and `ImplementationSourcePath`.	Name of source file needed for code replacement.
Additional header files/include paths For each file, specify the file name and path in calls to the functions `addAdditionalHeaderFile` and `addAdditionalIncludePath`.	Name and path of additional header files needed for code replacement.
Additional source files/paths For each file, specify the file name and path in calls to the functions `addAdditionalSourceFile` and `addAdditionalSourcePath`.	Name and path of additional source files needed for code replacement.
Additional object files/paths For each file, specify the file name and path in calls to the functions `addAdditionalLinkObj` and `addAdditionalLinkObjPath`.	Name and path of additional object files needed for code replacement.
Additional link flags Set the entry property `AdditionalLinkFlags` to a cell array of character vectors or string array representing the required compile flags.	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags Set the entry property `AdditionalCompileFlags` to a cell array of character vectors or string array representing the required link flags.	Flags the compiler needs to generate object code for the replacement code.

Parameter	Description
Copy files to build folder. Set property `GenCallback` to `'RTW.copyFileToBuildDir'`.	Flags whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. If a match occurs for a table entry, a call to the function `RTW.copyFileToBuildDir` copies required files to the build folder. If the configuration parameter `DataTypeReplacement` is set to `'CDataTypesFixedWidth'`, the function replaces Simulink Coder data types in the copied files with data types from the C99 language standard. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.

- If an entry uses header, source, or object files, consider whether to make the files accessible to the code generator. You can copy files to the build folder or you can specify individual file names and paths explicitly.
- If you specify *additional* header files/include paths or source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files (an instance in the build folder and an instance in the original folder).
- If you choose to copy files to the build folder and you are using the `packNGo` function to relocate static and generated code files to another development environment:
  - In the call to `packNGo`, specify the property-value pair `'minimalHeaders' true` (the default). That setting instructs the function to include the minimal header files required to build the code in the zip file.
  - Do not collocate files that you copy with files that you do not copy. If the `packNGo` function finds multiple instances of the same file, the function returns an error.
- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace that you enclose with dollar signs ($*variable*$). The code generator evaluates and replaces a token with the defined value. For example, consider the path `$myfolder$\folder1`, where `myfolder` is a character vector or string scalar variable defined in the MATLAB workspace as `'d:\work\source\module1'`. The code generator generates the custom path as `d:\work\source\module1\folder1`.

8 Validate and save.

Save your code replacement library customization file with the same name as your code replacement library table on your MATLAB path.

To save and validate your library:

a Save your file. From the MATLAB menu, select **File > Save**.

b Validate your library by calling it from the MATLAB command line. For example:

```
>> hTable = code_replacement_library_table()
```

See Also

More About

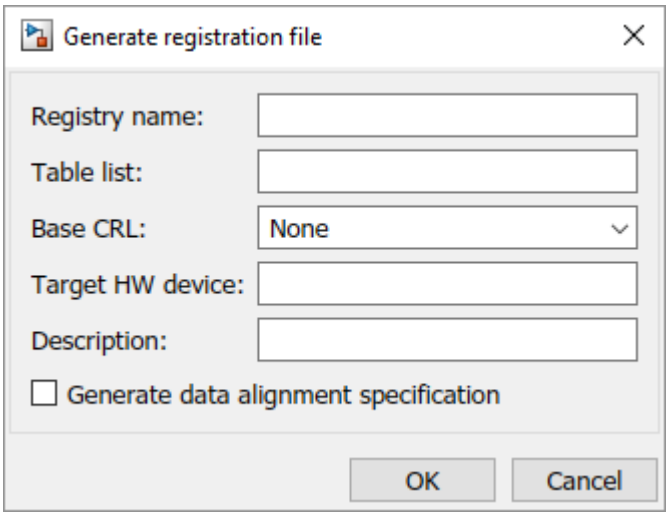
- “Code You Can Replace From Simulink Models” on page 72-6
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

Register Code Replacement Library

Create a code replacement library by defining in a registration file the library name, code replacement tables, and other information. After you define and register your library it becomes available to the code generator for code replacement.

Interactively Develop and Load a Registration file

1 In the crtool, select **File** > **Generate registration file**.



2 In the **Generate registration file** dialog box, specify registration information. Then click **OK**.

Parameter	Description
Registry name (Required)	Code replacement library name.
Table list (Required)	Comma-separated list of one or more code replacement tables that you want in the library. Specify tables by using either their names (if located on the MATLAB search path), absolute paths, or relative paths.
Base CRL	Enables you to specify library hierarchies. Provide the name of a registered library to serve as a base for the library that you are currently registering. For example, a TI device library can be a base for a TI C28x library.
Target HW device	Comma-separated list of one or more hardware devices that this library supports. Indicate support of all devices with an asterisk (*).
Description	Description of the purpose and content of the library.

Parameter	Description
Generate data alignment specification	Flag that enables data alignment.

3 To use your registered code replacement library, refresh your current MATLAB session with the command:

```
>> sl_refresh_customizations
```

Programmatically Develop and Load a Registration File

1 Open a MATLAB function file. From the MATLAB menu, select **New** > **Function**.

2 Save the function file with the name `rtwTargetInfo.m`.

3 Copy and paste the following code into your file to use as a template.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry; %instantiates a registration entry
this(1).Name = 'Example Code Replacement Library';
this(1).TableList = {'example_code_replacement_table.m'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'This registers an example library';

end
```

4 In the template, specify the following registration information:

Parameter	Description
Registry name (Required)	Code replacement library name.
Table list (Required)	Comma-separated list of one or more code replacement tables that you want in the library. Specify tables by using either their names (if located on the MATLAB search path), absolute paths, or relative paths.
Base CRL	Enables you to specify library hierarchies. Provide the name of a registered library to serve as a base for the library that you are currently registering. For example, a TI device library can be a base for a TI C28x library.
Target HW device	Comma-separated list of one or more hardware devices that this library supports. Indicate support of all devices with an asterisk (*).
Description	Description of the purpose and content of the library.
Generate data alignment specification	Flag that enables data alignment.

5. Save the file on your MATLAB path.
6. To use your registered code replacement library, refresh your current MATLAB session with the command:

```
>> sl_refresh_customizations
```

Examples

Register one code replacement library

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry; %instantiates a registration entry
this(1).Name = 'Example Code Replacement Library';
this(1).TableList = {'example_code_replacement_table.m'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'This registers an example library';

end
```

Register multiple code replacement libraries

```
function rtwTargetInfo(cm)
cm.registerTargetInfo(@loc_register_crl);
end

function thisCRL = loc_register_crl
  % Register a code replacement library for use with model: rtwdemo_crladdsub
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Addition & Subtraction Examples';
  thisCrl(1).Description = 'Example of addition/subtraction op replacement';
  thisCrl(1).TableList = {'crl_table_addsub'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlmuldiv
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'Multiplication & Division Examples';
  thisCrl(2).Description = 'Example of mult/div op repl for built-in integers';
  thisCrl(2).TableList = {'c:/work_crl/crl_table_muldiv'};
  thisCrl(2).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlfixpt
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'Fixed-Point Examples';
  thisCrl(3).Description = 'Example of fixed-point operator replacement';
  thisCrl(3).TableList = {fullfile('$(MATLAB_ROOT)', ...
     'toolbox','rtw','rtwdemos','crl_demo','crl_table_fixpt')};
   thisCrl(3).TargetHWDeviceType = {'*'};

end
```

Register a code replacement hierarchy

```
function rtwTargetInfo(cm)
```

```
cm.registerTargetInfo(@loc_register_crl);
end

function thisCRL = loc_register_crl
  % Register a code replacement library that includes common entries
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Common Replacements';
  thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
  thisCrl(1).TableList = {'crl_table_general'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for TI devices
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'TI Device Replacements';
  thisCrl(2).Description = 'Code replacement entries shared across TI devices';
  thisCrl(2).TableList = {'crl_table_TI_devices'};
  thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
  thisCrl(1).BaseTfl = 'Common Replacements'

  % Register a code replacement library for TI c6xx devices
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c6xx Device Replacements';
  thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
  thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
  thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

  % Register a code replacement library for the TI c67x device
  thisCrl(4) = RTW.TflRegistry;
  thisCrl(4).Name = 'TI c67x Device Replacements';
  thisCrl(4).Description = 'Code replacement entries for the TI C67x device';
  thisCrl(4).TableList = {'crl_table_TIC67x_device'};
  thisCrl(4).TargetHWDeviceType = {'TI 67x'};


end
```

See Also

More About

- “Troubleshoot Code Replacement Library Registration” on page 72-58
- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

Verify Code Replacement Library

Verify your code replacement library by examining its structure, replacement, and performance. Use the Code Replacement Viewer (crviewer) to verify the structure of your library. Use the code replacement report to verify the replacements made with your library in generated code. Use crviewer again to verify the performance of your library during code generation by examining the replacement hit and miss logs.

Verify Library Structure

Use the Code Replacement Viewer (crviewer) to open and view the structure of your code replacement library by examining its tables and table entries.

1. From the MATLAB command prompt, open your code replacement library using the crviewer with the command:

   ```
   >> crviewer('library')
   ```

2. The viewer opens your code replacement library and shows its code replacement tables in the left pane. Verify that your library lists all required tables in the correct relative order.
3. Open and view the entries for each code replacement table. In the crtool, click each table listed in the left pane to view its entries in the middle pane.

   If you are not using the crviewer, you can open each table directly from the MATLAB command prompt with the command:

   ```
   crviewer(table)
   ```

4. For each table in your library, verify that it lists all required entries in the correct relative order. Verify that the conceptual representation and implementation representation are correct for each entry in the table.

Verify Library Replacement

Use the Code Replacements Report to verify that when the code generator uses your code replacement library it creates replacements as expected in the generated code. For each instance of replacement, the Code Replacements Report provides a mapping between the Simulink block that triggered the replacement and its implementation in the generated code. Use this mapping to verify correct code replacement by tracing between the report and the code in either direction (from report to code or code to report).

To generate the Code Replacements Report:

1. Configure your model to use your code replacement library.
2. Configure your model to open the Code Generation Report. The Code Replacements Report is a tab within this report. In the Model Configuration Parameters dialog box, in the **Code Generation** > **Report** pane, select:

   - **Create code generation report**
   - **Open report automatically**
   - **Model-to-code**

- **Summarize which block triggered code replacements**

3. Configure the generated code to include comments. In the Model Configuration Parameters dialog box, in the **Code Generation** > **Comments** pane, select:
   - **Include comments**
   - Either or both of **Simulink block comments** and **Simulink block descriptions**
4. Build your model to generate code only. The Code Generation Report opens automatically. On the left pane, click to open the Code Replacements Report.
5. Verify that code replacements occur as expected.
   - **Navigate from report to model**: In the Code Replacements Report, click each block link to see the source that triggered replacement.
   - **Navigate from model to report**: In the Simulink model window, use model-to-code highlighting to trace code replacements. Right-click blocks where you expected code replacement to occur and select **C/C++ Code** > **Navigate to C/C++ Code**. The report appears with the corresponding replacement code highlighted.

Verify Library Performance

Verify the performance of your code replacement library by examining the hits and misses that occurred during code generation using your library. During code generation, when the code generator finds a match for code replacement from your code replacement library, it logs the hit in a hit cache. When it does not find a code replacement match, it logs the failed attempt in the miss cache. Each time that you generate code, the code generator overwrites its previous hit and miss cache logs.

To review trace information from the hit and miss cache data, use the Code Replacement Viewer (crviewer). From the hit cache, you can identify when code replacement successfully occurred, and from the miss cache, you can troubleshoot when code replacement did not take place. The miss cache trace information provides you information about the reason for the miss, a link to the relevant source location for the miss, and the call site object of the miss. For more information, see “Troubleshoot Code Replacement Misses” on page 72-59.

See Also

More About

- “Troubleshoot Code Replacement Misses” on page 72-59
- “Register Code Replacement Library” on page 72-51
- “Deploy Code Replacement Library” on page 72-57
- “What Is Code Replacement Customization?” on page 72-3

Deploy Code Replacement Library

Package and deploy your code replacement library.

1. Move your code replacement library customization and registration files to a location on the MATLAB® search path that is accessible to other users. If another registration file already exists in that location, copy and paste the information for your code replacement library into the existing registration file.
2. Refresh your session and verify that the library is available as a configuration option for the code generator.
3. Verify that code replacement occurs as expected in generated code.
4. Inform users that the code replacement library is available.

See Also

More About

- “Relocate or Share Generated Code”
- “Develop a Code Replacement Library” on page 72-20
- “What Is Code Replacement Customization?” on page 72-3

Troubleshoot Code Replacement Library Registration

If a code replacement library is not listed as a configuration option or does not appear in the Code Replacement Viewer:

- Refresh the library registration information within the current MATLAB session (`RTW.TargetRegistry.getInstance('reset');` or for the Simulink environment,`sl_refresh_customizations`).
- See whether the registration file, `rtwTargetInfo.m`, contains an error.

See Also

More About

- “Register Code Replacement Library” on page 72-51

Troubleshoot Code Replacement Misses

Use miss reason messages that appear in the **Code Replacement Viewer** to analyze and correct code replacement misses.

Miss Reason Messages

The Code Replacement Viewer displays miss reason messages in trace information for code replacement misses. A legend listing each message that appears in the miss report precedes the report details. A message consists of:

- Numeric identifier, which identifies the message in the report details.
- Message text, which in some cases includes placeholders for names of arguments, call site object values, table entry values, and property names.

For example:

```
1. Mismatched data types (argument name, CSO value, table entry value)
```

The parenthetical information represents placeholders for actual values that appear in the report details.

In the **Miss Source Locations** table that lists the miss details, the **Reason** column includes:

- The message identifier, as listed in the legend.
- The placeholder values for that instance of the miss reason message.

The following **Reason** details indicate a data type mismatch because the call site object specifies data type `int8` for arguments `y1`, `u1`, and `u2`, while the code replacement table entry specifies `uint32`.

```
1. y1, int8, uint32
   u1, int8, uint32
   u2, int8, uint32
```

Depending on your situation and the reported miss reason, troubleshoot reported misses by looking for instances of the following:

- A typo in the code replacement table entry definition or a source parameter setting.
- Information missing from the code replacement table entry or a source parameter setting.
- Invalid or incorrect information in the code replacement table entry definition or a source parameter setting.
- Arguments incorrectly ordered in the code replacement table entry definition or the source being replaced with replacement code.
- Failed algorithm classification for an addition or subtraction operation due to:
  - An ideal accumulator not being calculated because the type of an input argument is not fixed-point or the slope adjustment factors of the input arguments are not equal.
  - Input or output casts with a floating-point cast type.
  - Input or output casts with cast types that have different slope adjustment factors or biases.
  - Output casts not being convertible to a single output cast.

- Input casts resulting in loss of bits.

Analyze and Correct Code Replacement Misses

The following example shows how to use Code Replacement Viewer trace information to troubleshoot code replacement misses. You must have already reviewed and tested code replacements for your model.

1. Review the code generated for a model element, looking for expected code replacements. For this example, examine the code generated for block Sub32 in model `rtwdemo_crladdsub`. Right-click the block and select **C/C++ Code** > **Navigate to C/C++ Code**.

   The Code Generation Report opens to the location of the generated code for that block.

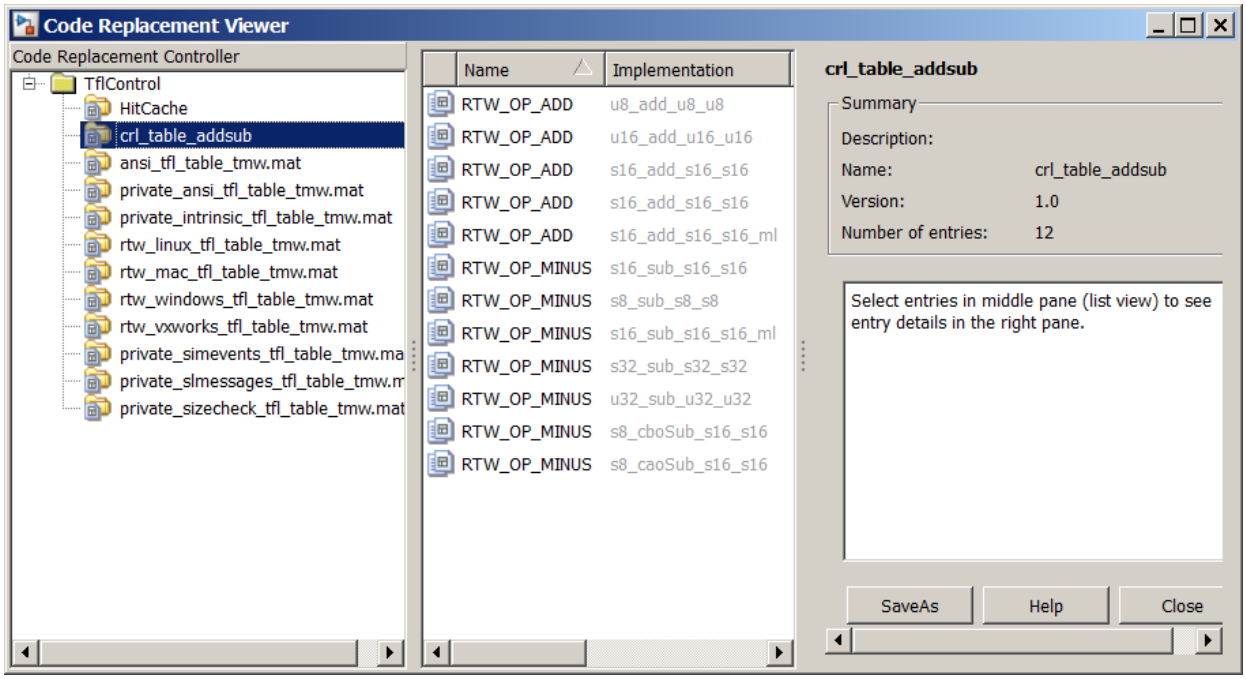
```
63   /* Outport: '<Root>/Out5' incorporates:
64    *  Inport: '<Root>/In10'
65    *  Inport: '<Root>/In9'
66    *  Sum: '<Root>/Sub32'
67    */
68   rtY.Out5 = s32_sub_s32_s32(rtU.In9, rtU.In10);
```

   The code generator replaced code, but the replacement was for the signed version of the 32-bit subtraction operation. You expected an unsigned operation.

2. Regenerate or reopen the Code Replacements Report for your model. If you already generated the code generation report that includes the Code Replacements Report for model `rtwdemo_crladdsub`, open the file `rtwdemo_crladdsub_ert_rtw/html/index.html`. For information on how to regenerate the report, see “Verify Code Replacement Library” on page 72-55.

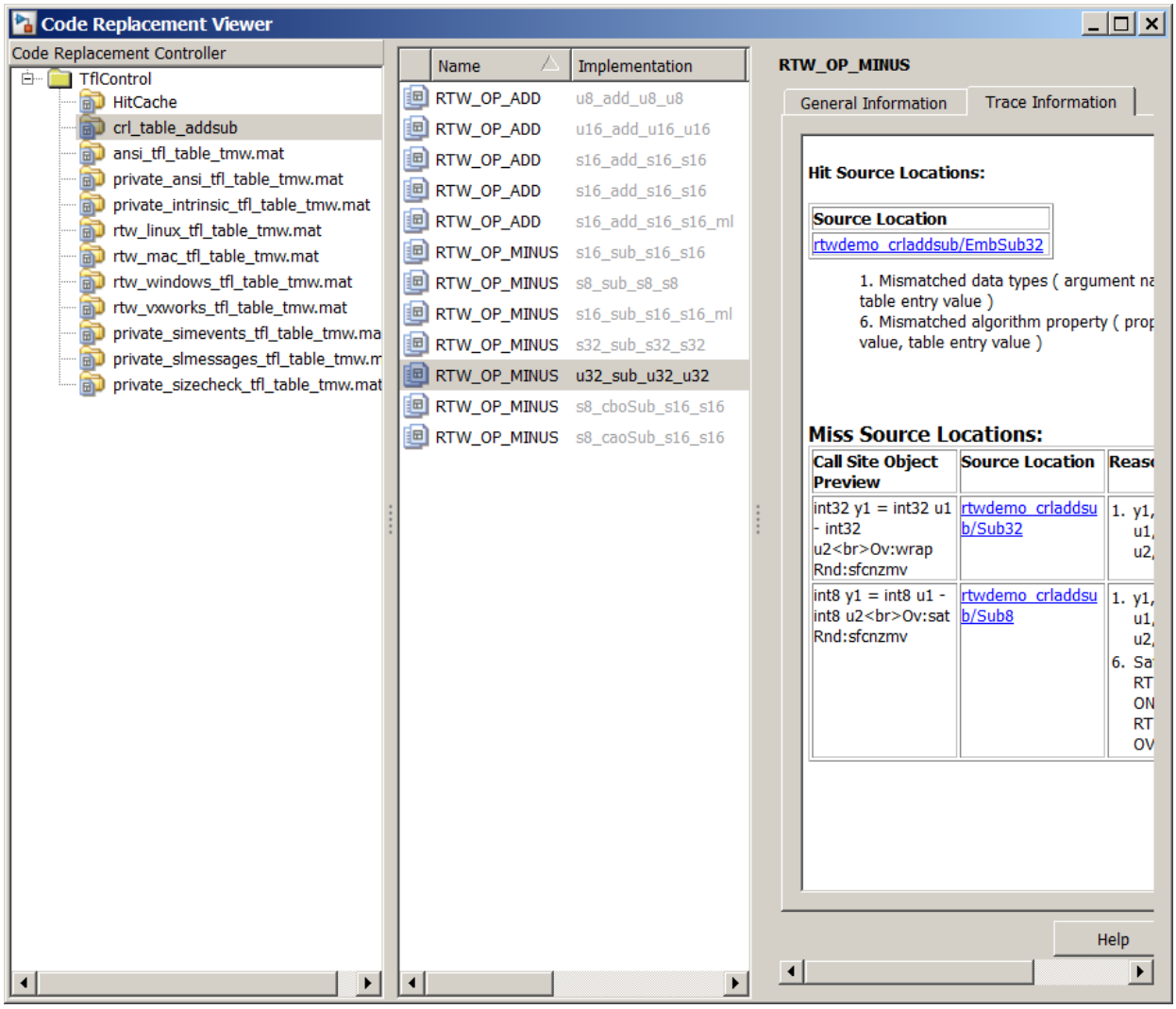
3. Click the link to open the Code Replacement Viewer.

4. In the viewer left pane, select your code replacement table. The following display shows entries for code replacement table `crl_table_addsub`.



5 In the middle pane, select table entry `RTW_OP_MINUS` with implementation function `u32_sub_u32_u32`.

6 In the right pane, select the **Trace Information** tab.



The **Trace Information** is a table that lists the following information for each miss:

- Call site object preview. The call site object is the conceptual representation of a subtraction operator. The code generator uses this object to query the code replacement library for a match.
- A link to the source location in the model for which the code generator considered replacing code.
- The reasons that the miss occurred. For the list of reasons that misses occur, see “Miss Reason Messages” on page 72-59.

For this example, the report shows misses for two blocks: Sub32 and Sub8.

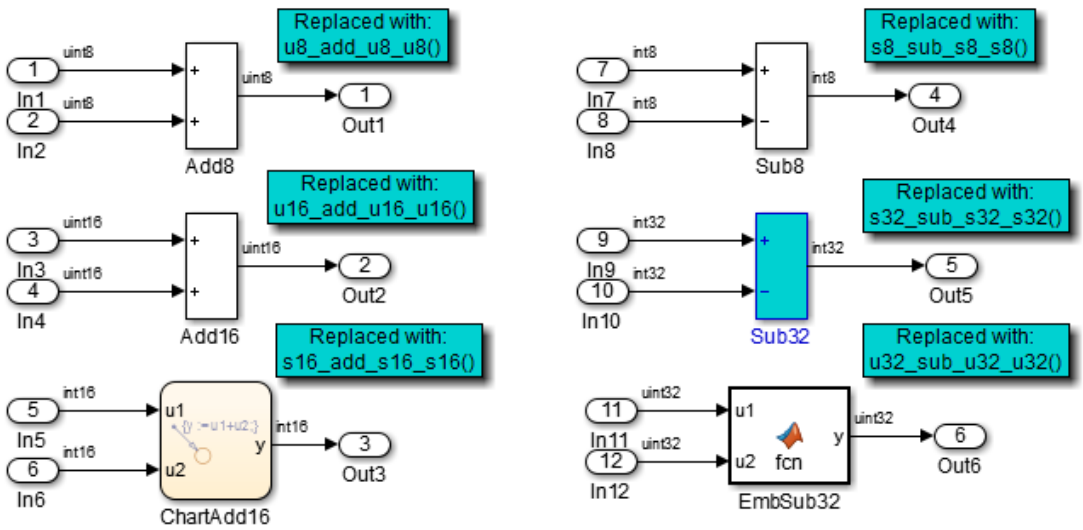
7 Find that source in the trace information. Depending on your situation and the reported miss reason, consider looking for a condition such as a typo in the code replacement table entry

definition or in a source parameter setting. “Miss Reason Messages” on page 72-59 lists conditions to consider.

For this example, determine why code for the Sub32 block was not replaced with code for an unsigned 32-bit subtraction operation. The miss reason for the Sub32 block indicates a data type mismatch. The data type in the call site object for the three arguments is a signed 32-bit integer. The code replacement entry specifies an unsigned 32-bit integer.

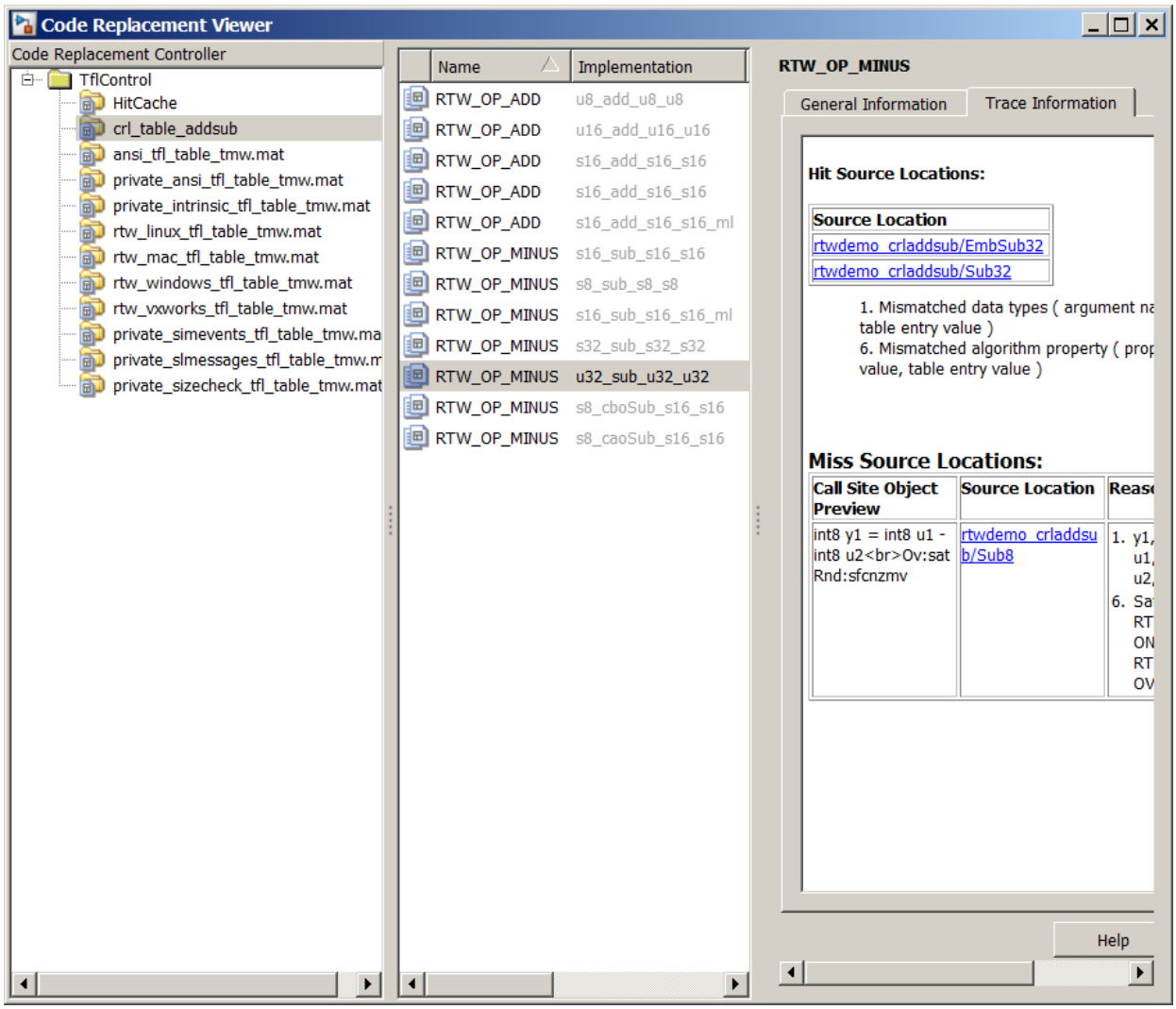
8 Correct the model or code replacement table entry. If the issue is in the model, use the source location link in the trace information to find the model element to correct. For this example, you expected an unsigned subtraction operation for the Sub32 block. Click the link in the trace report for the Sub32 block.

The model opens with the Sub32 block highlighted.



Change the data type setting for the two input signals and the output signal for the Sub32 block to `uint32`.

9 Regenerate code. Use the Code Replacement Viewer trace information to verify that your model or code replacement table entry corrects the code replacement issue. In the following display, the trace information shows a hit for block Sub32.



See Also

More About

- “Verify Code Replacement Library” on page 72-55

Math Function Code Replacement

This example shows how to develop a code replacement library to optimize the performance of math functions by providing information on how to define code replacement for a `sin` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_sin`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

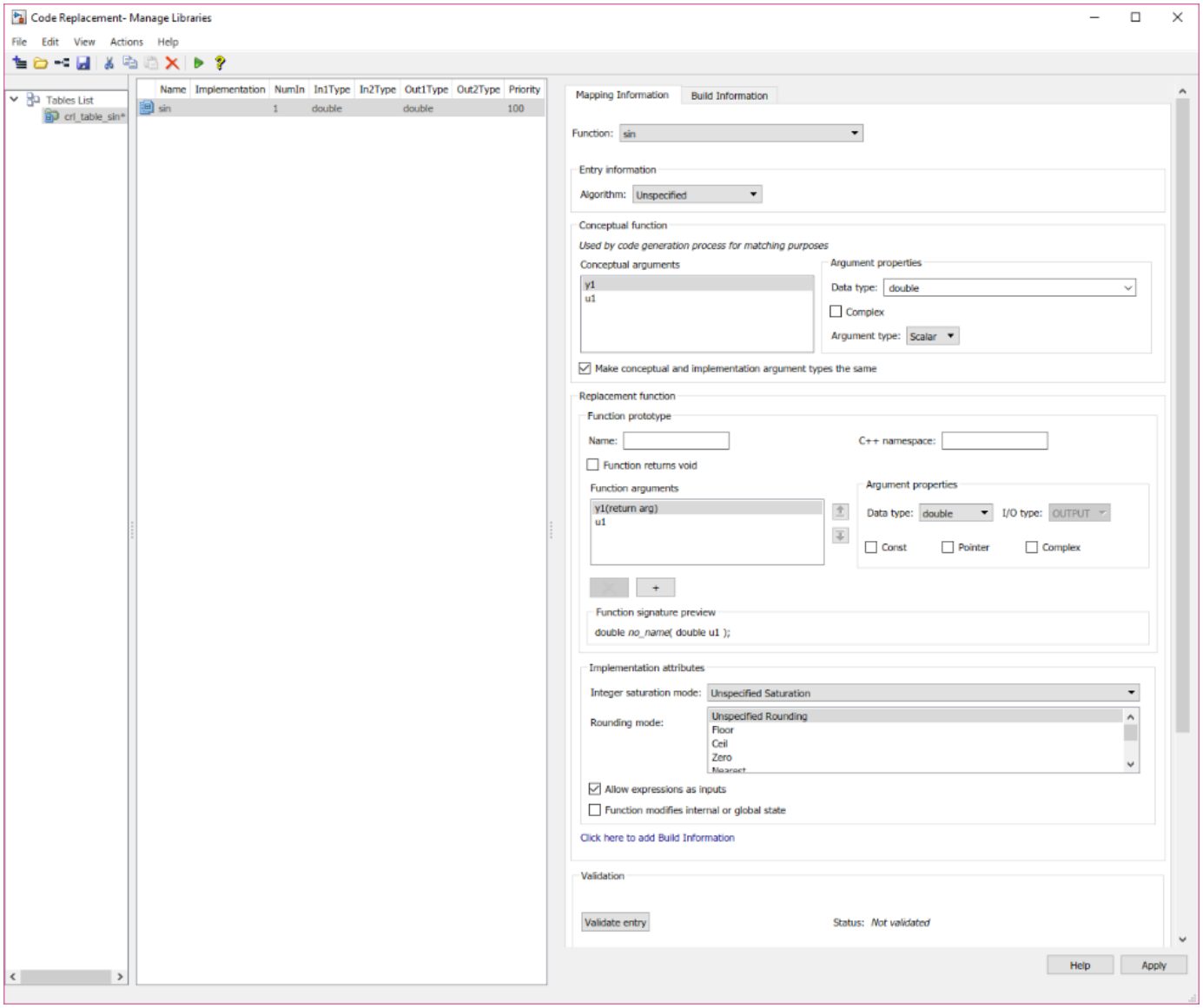
4. Create entry parameters. In the **Function** drop-down list, select `sin`. Algorithm information appears in the crtool. For this example, leave the parameter unspecified.

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input argument, u1, with the **Data Type** of double and the **Argument Type** of Scalar.

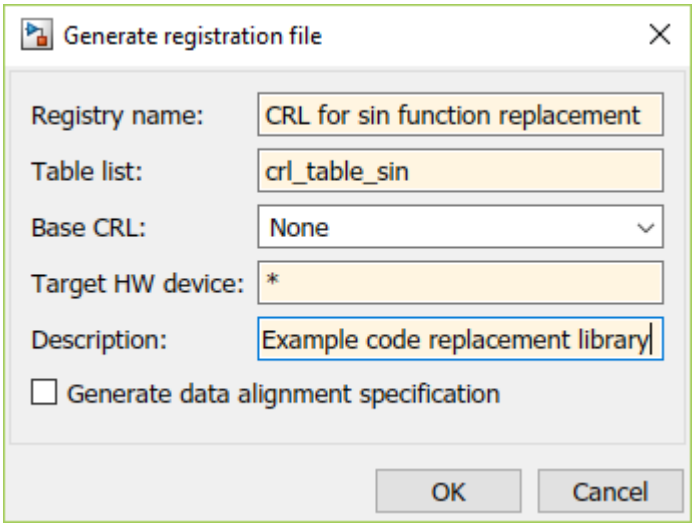
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_sin
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_sin
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');
```

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
        'DataTypeMode',           'double');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'double');
```

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
        'DataTypeMode',           'double');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'double');

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_sin
```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for sin function replacement’;
this(1).TableList = {'crl_table_sin.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';
```

```
end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Generate Code by Using Math Code Replacement Library

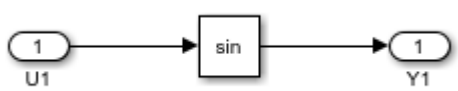
In this section, you generate code using the code replacement library that you created in the first two sections of this example. A math function `sin` is replaced with a call to a custom function in the generated code. This example does not provide an implementation function. Write your own implementation.

Example Model

Open the model `crl_mathfunction` for configuring the code replacement library.

```
open_system('crl_mathfunction');
```

```
copyfile mathFnRtwTargetInfo.txt rtwTargetInfo.m
```



Run the MATLAB® customization file to create a code replacement table and register the file. The customization file has already been executed. Run the sl_refresh_customizations function to register the library.

```
sl_refresh_customizations;
```

Enable Code Replacement Library

1 Open the Configuration Parameters dialog box.

2 On the **Interface** pane, set **Code Replacement Library** by clicking **Select** and adding `CRL for sin function replacement` to the **Selected code replacement libraries - prioritized** list pane. Alternatively, use the command-line API to enable the code replacement:

```
set_param('crl_mathfunction', 'CodeReplacementLibrary', 'CRL for sin function replacement');
```

Generate code from the model:

```
evalc('slbuild(''crl_mathfunction'')');
```

View the generated code. Here is a portion of `crl_mathfunction.c`.

```
cfile = fullfile('crl_mathfunction_ert_rtw','crl_mathfunction.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void crl_mathfunction_step(void)
{
  /* Outport: '<Root>/Y1' incorporates:
   *  Inport: '<Root>/U1'
   *  Trigonometry: '<Root>/Trigonometric Function'
   */
  crl_mathfunction_Y.Y1 = mySin(crl_mathfunction_U.U1);
}
```

The `sin` function is replaced with a custom `mysin` function.

Close the model and code generation report.

```
delete ./rtwTargetInfo.m
bdclose('crl_mathfunction');
rtwdemoclean;
```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Algorithm-Based Code Replacement” on page 72-88
- “Develop a Code Replacement Library” on page 72-20

Buffer Index Calculation Code Replacement

This example shows how to develop a code replacement library to optimize the performance of the buffer index calculation for a Delay block. Optimize the buffer index calculation by defining a code replacement for the circular index calculation function. To develop a code replacement library, use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool). At the MATLAB command line, enter:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_circularIndex`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `circularIndex`.

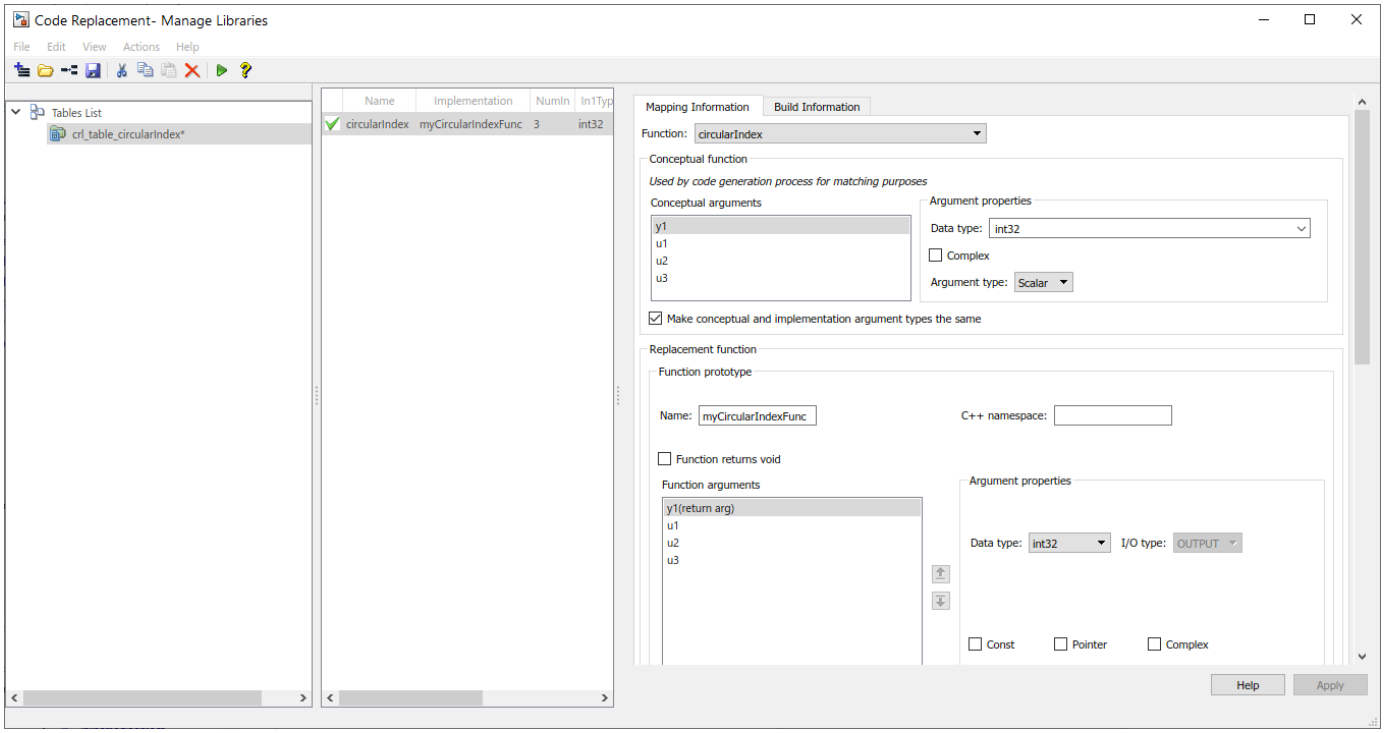
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input arguments, u1, u2, and u3, that use the **Data Type** of `int32` and the **Argument Type** of `Scalar`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

7. Specify the name of the replacement function. In the **Replacement Function** section, set **Name** to `myCircularIndexFunc`.

8. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

9. Validate and save the table. Click the **Mapping Information** tab and verify that the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



10 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:

- **Registry name** — CRL for circular buffer index replacement
- **Table list** — crl_table_circularIndex
- **Description** — Example code replacement library

To use your code replacement library, refresh your current MATLAB session. At the command line, enter:

```
>>sl_refresh_customizations
```

11 Verify the code replacement library. At the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_circularIndex()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_circularIndex()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4 Create entry parameters. Because this example replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_circularIndex()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'circularIndex', ...
        'Priority',                     30, ...
        'ImplementationName',           'myCircularIndexFunc');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_circularIndex()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'circularIndex', ...
        'Priority',                     30, ...
        'ImplementationName',           'myCircularIndexFunc');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
```

```
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u2', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u3', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'int32');
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_circularIndex()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                        'circularIndex', ...
        'Priority',                   30, ...
        'ImplementationName',         'myCircularIndexFunc');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u2', ...
        'IOType',                 'RTW_IO_INPUT',...
```

```
        'DataTypeMode',           'int32');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u3', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'int32');

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator requires for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. At the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_circularIndex
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for circular index calculation replacement';
   this(1).TableList = {'crl_table_circularIndex.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session. At the command line, enter:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. At the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

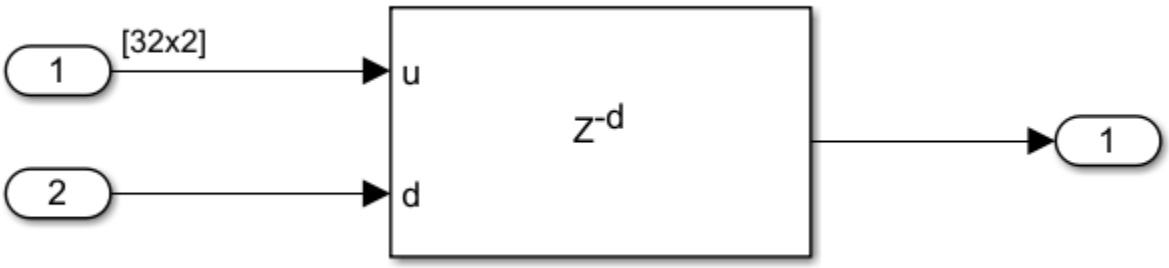
Generate Code by Using Circular Index Code Replacement Library

Generate code by using the code replacement library that you previously created in this example. The code that calculates the circular index for a delay buffer is replaced with a call to a custom function in the generated code. This example does not provide an implementation function. Write your own implementation or use the target specific implementation.

Example Model

The example model contains a Delay block with these properties:

- **Delay length** — Input port
- **Input processing** — Columns as channels (frame-based)
- **Use circular buffer for state** — On



The Delay block receives a 32 by 2 data input signal and a scalar delay length input signal. To replace the buffer index calculation, for the Delay block, you must enable **Use circular buffer for state** enabled and use frame-based input processing. Frame-based input processing requires DSP System Toolbox. For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

Enable Code Replacement Library

1. Open the Configuration Parameters dialog box.
2. On the **Interface** pane, set **Code Replacement Library** by clicking **Select**. Add `CRL for circular index calculation replacement` to the **Selected code replacement libraries - prioritized** list pane. Alternatively, use the command-line API to enable the code replacement.

   ```
   set_param('delayModel', 'CodeReplacementLibrary', 'CRL for circular index calculation');
   ```
3. Generate code from the model.

View the generated code. The portion of `delayModel.c` that calculates the buffer index uses the replacement function `myCircularIndexFunc`.

```
/* Update for Delay: '<Root>/Delay' incorporates:
 *  Inport: '<Root>/Input'
 */
currIdx = rtDW->CircBufIdx;
for (frameIdx = 0; frameIdx < 32; frameIdx++) {
  rtDW->Delay_DSTATE[currIdx] = rtInput[frameIdx];
  rtDW->Delay_DSTATE[currIdx + 200] = rtInput[frameIdx + 32];
  currIdx = myCircularIndexFunc(currIdx, 1, 200);
}

if (rtDW->CircBufIdx < 168) {
  rtDW->CircBufIdx += 32;
} else {
  rtDW->CircBufIdx -= 168;
}

/* End of Update for Delay: '<Root>/Delay' */
```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Algorithm-Based Code Replacement” on page 72-88
- “Develop a Code Replacement Library” on page 72-20

Memory Function Code Replacement

This example shows how to develop a code replacement library to optimize the performance of memory functions by providing information on how to define code replacement for a `memcpy` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_memcpy`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `memcpy`.

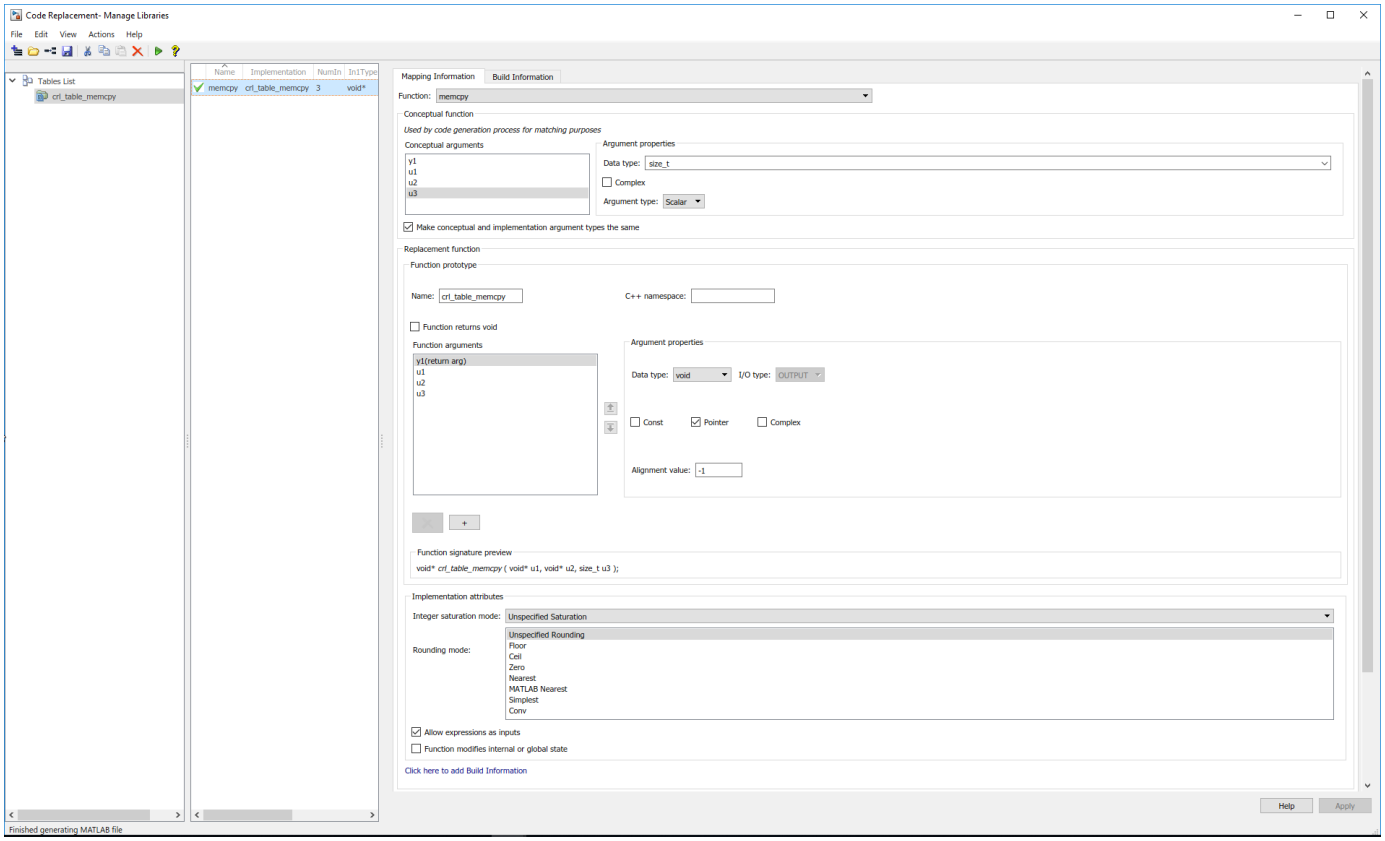
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, with the **Data Type** of void and the **Argument Type** of Pointer, the input arguments, `u1` and `u2` with the **Data Type** of void and the **Argument Type** of Pointer and `u3` with the **Data Type** of unsigned integral and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

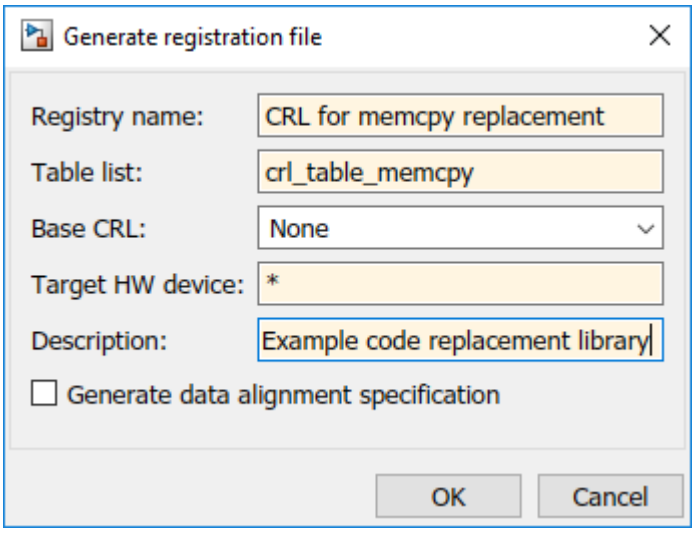
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_memcpy()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;


%% Create entry parameters
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
hEntry.setTflCFunctionEntryParameters(...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;


%% Create entry parameters
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
hEntry.setTflCFunctionEntryParameters(...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);

%% Create conceptual arguments
arg = getTflArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u2', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);


%% Create conceptual arguments
arg = getTflArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u2', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(hEntry, arg);
```

```
%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_memcpy
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for memory function replacement’;
   this(1).TableList = {'crl_table_memcpy.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Nonfinite Function Code Replacement

This example shows how to develop a code replacement library to optimize the performance of nonfinite functions by providing information on how to define code replacement for a `sqrt` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_nonfinite`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `getNaN` .

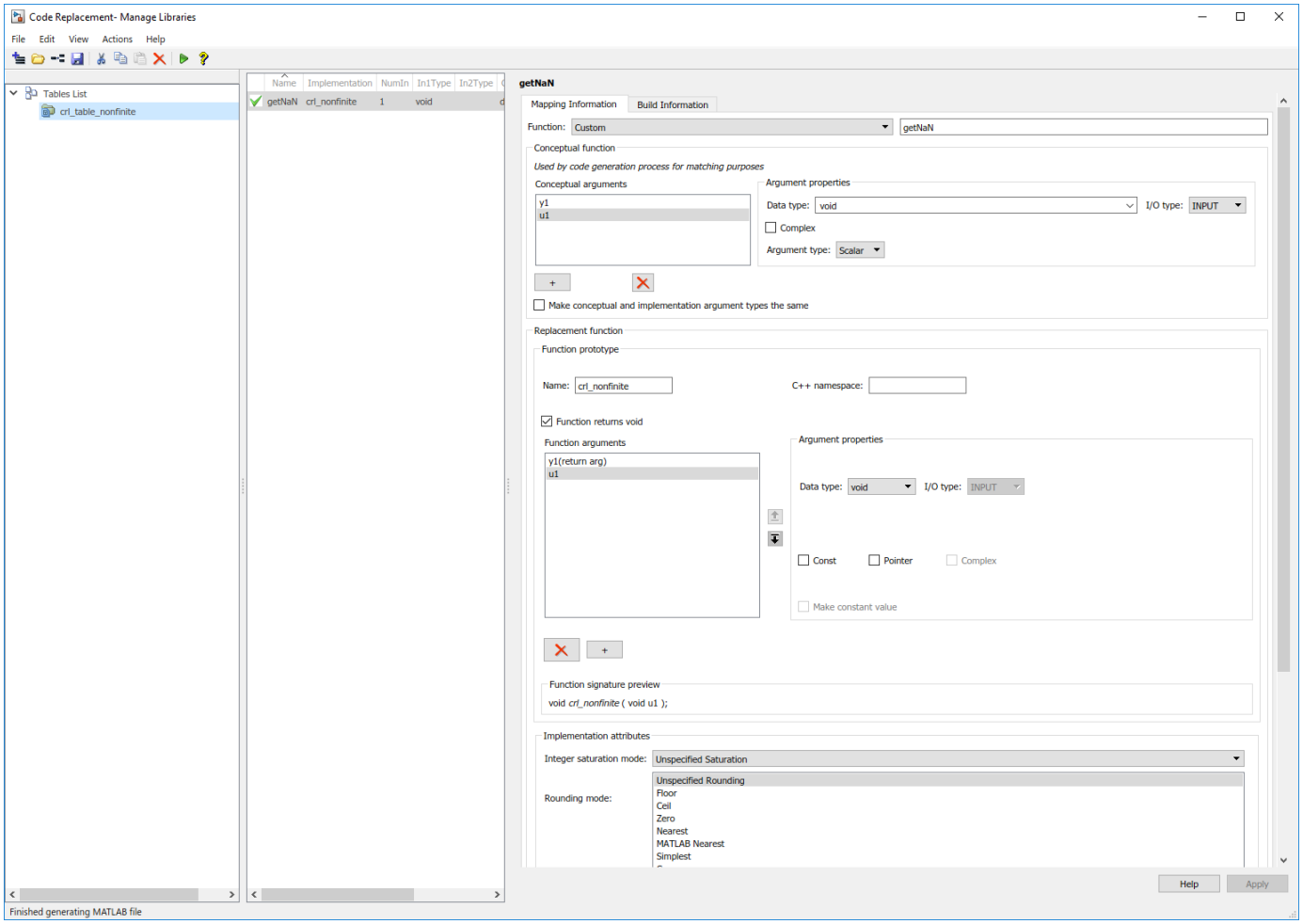
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, with the **Data Type** of double and the **Argument Type** of scalar and the input argument, `u1`, with the **Data Type** of void and the **Argument Type** of scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

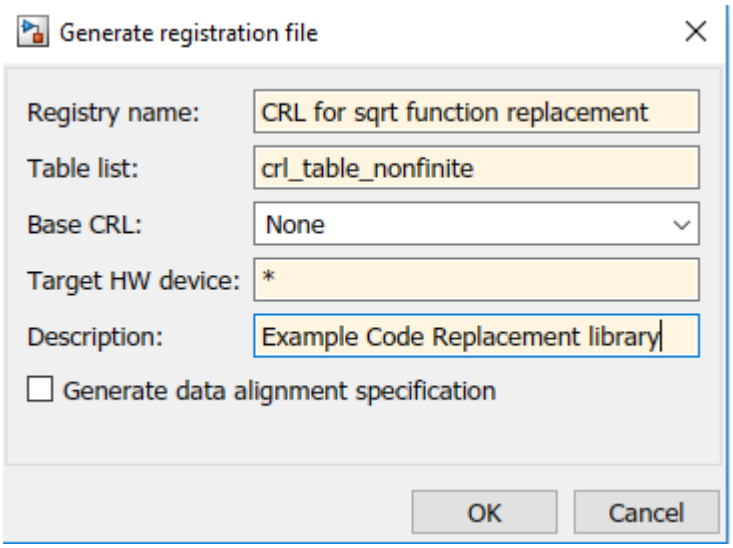
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, set the **Implementation header file** to `nonfin.h`.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_nonfinite()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_nonfinite()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;


%% Create an entry
hEntry  = rtw.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_nonfinite()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry  = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'getNaN', ...
        'Priority',                     30, ...
        'ImplementationName',           'mygetNaN', ...
        'ImplementationHeaderFile',     'getInf.h', ...
        'ImplementationSourceFile',     'getInf.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_nonfinite()
% Create a code replacement library table
```

```
%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry  = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'getNaN', ...
        'Priority',                     30, ...
        'ImplementationName',           'mygetNaN', ...
        'ImplementationHeaderFile',     'getInf.h', ...
        'ImplementationSourceFile',     'getInf.c');

%% Create the conceptual representation
  arg = getTflArgFromString(hTable, 'y1', 'double');
  arg.IOType = 'RTW_IO_OUTPUT';
  addConceptualArg(hEntry, arg);

  arg = getTflArgFromString(hTable, 'u1', 'void');
  addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_nonfinite()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry  = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'getNaN', ...
        'Priority',                     30, ...
        'ImplementationName',           'mygetNaN', ...
        'ImplementationHeaderFile',     'getInf.h', ...
        'ImplementationSourceFile',     'getInf.c');

%% Create the conceptual representation
  arg = getTflArgFromString(hTable, 'y1', 'double');
  arg.IOType = 'RTW_IO_OUTPUT';
  addConceptualArg(hEntry, arg);

  arg = getTflArgFromString(hTable, 'u1', 'void');
  addConceptualArg(hEntry, arg);

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
addEntry(hTable, hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_nonfinite
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
```

```
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for nonfinite function replacement';
this(1).TableList = {'crl_table_nonfinite.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Algorithm-Based Code Replacement

You can develop a code replacement library to optimize the performance of certain math functions based on the computation or approximation algorithm configured for a block.

For example, you can configure:

- The Reciprocal Sqrt block to use the `Newton-Raphson` or `Exact` computation method.
- The Trigonometric Function block, with **Function** set to `sin`, `cos`, `sincos`, or `atan2`, to use the approximation method `CORDIC`, `Lookup`, or `None`.

To develop a code replacement library, use either the interactive or programmatic approach. For more information, see "Develop a Code Replacement Library" on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

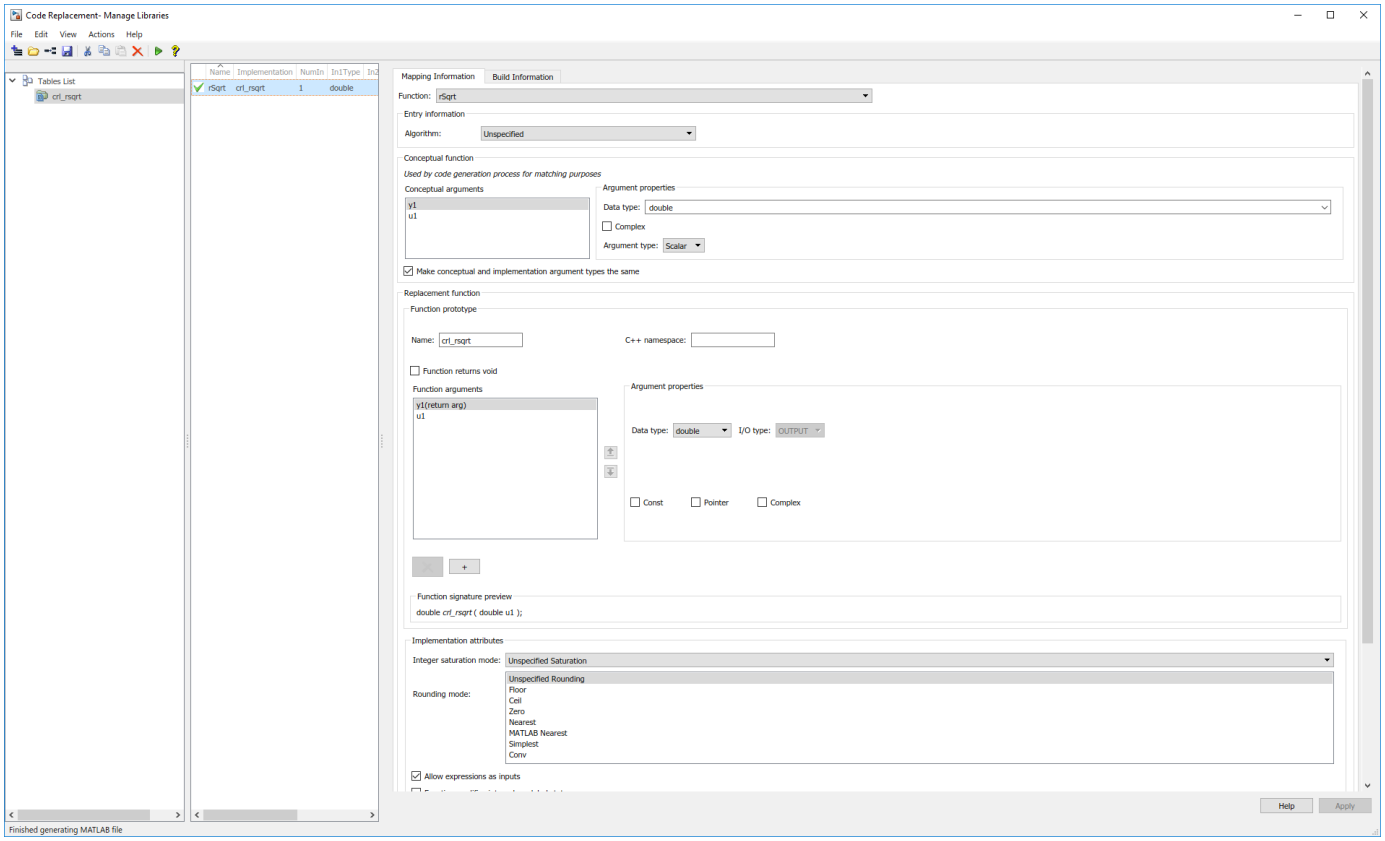
2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

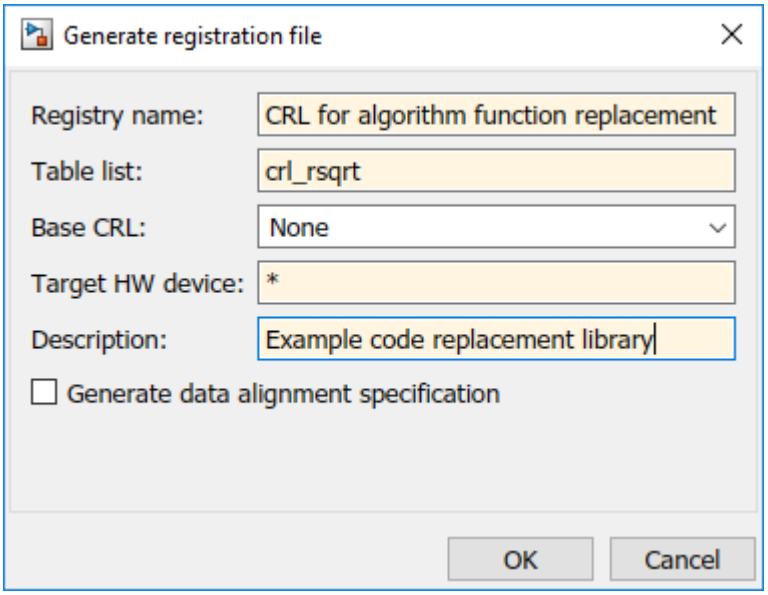
   b. In the right pane, name the table `crl_rsqrt`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.
4. Create entry parameters. In the **Function** drop-down list, select `rSqrt`. Algorithm information appears in the crtool. For this example, set the parameter to **Newton Raphson**.
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1`, by using the **Data Type** of double and the **Argument Type** of Scalar.

   Specify a **Name** for the replacement function under **Function prototype**.
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.
7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.
8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function cannot have arguments or return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_rsqrt()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_rsqrt()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

   To use the algorithm for a function in an entry definition, set the `EntryInfoAlgorithm` property in a call to the function `setTflCFunctionEntryParameters` as `RTW_Newton_Raphson`.

```
function hTable = crl_rsqrt()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
  'Key',                      'rSqrt', ...
  'Priority',                 80, ...
  'ImplementationName',       'rsqrt_newton', ...
  'ImplementationHeaderFile', 'rsqrt.h', ...
  'EntryInfoAlgorithm',       'RTW_NEWTON_RAPHSON');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_rsqrt()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
  'Key',                      'rSqrt', ...
  'Priority',                 80, ...
  'ImplementationName',       'rsqrt_newton', ...
  'ImplementationHeaderFile', 'rsqrt.h', ...
  'EntryInfoAlgorithm',       'RTW_NEWTON_RAPHSON');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
  'Name',         'y1', ...
  'IOType',       'RTW_IO_OUTPUT', ...
  'DataTypeMode', 'double');

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
  'Name',         'u1', ...
  'DataTypeMode', 'double');
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_rsqrt()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
  'Key',                      'rSqrt', ...
  'Priority',                 80, ...
  'ImplementationName',       'rsqrt_newton', ...
  'ImplementationHeaderFile', 'rsqrt.h', ...
  'EntryInfoAlgorithm',       'RTW_NEWTON_RAPHSON');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
  'Name',         'y1', ...
  'IOType',       'RTW_IO_OUTPUT', ...
```

```
  'DataTypeMode', 'double');

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
  'Name',         'u1', ...
  'DataTypeMode', 'double');

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_rsqrt
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)
   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl
   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for algorithm function replacement';
   this(1).TableList = {'crl_rsqrt.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';
   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20

- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Lookup Table Function Code Replacement

You can configure the algorithm for table lookup operations and index searches and generate code. If the code generated for available algorithm options does not meet requirements for your application, create custom code replacement table entries to replace the generated algorithm code. You can develop a code replacement library to optimize the performance of algorithms in a lookup table block. For more information, see “Develop a Code Replacement Library” on page 72-20. For more information about using lookup table blocks, see “Nonlinearity”.

Lookup Table Function Signatures

To create code replacement table entries for a function corresponding to a lookup table algorithm, you must have:

- Information about the conceptual function signature.
- Relevant algorithm parameters.

This table provides the conceptual function signature information.

Conceptual Function Signature	Argument Summary
`y1 = interp1D(u1, u2, u3, u4)`	`y1` – output `u1` – index `u2` – fraction `u3` – table data `u4` – table dimension length
`y1 = interp2D(u1, u2, u3, u4, u5, u6, u7)`	`y1` – output `u1, u3` – index `u2, u4` – fraction `u5` – table data `u6, u7` – table dimension lengths
`y1 = interp3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)`	`y1` – output `u1, u3, u5` – index `u2, u4, u6` – fraction `u7` – table data `u8, u9, u10` – table dimension lengths
`y1 = interp4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u3, u5, u7` – index `u2, u4, u6, u8` – fraction `u9` – table data `u10, u11, u12, u13` – table dimension lengths
`y1 = interp5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)`	`y1` – output `u1, u3, u5, u7, u9` – index `u2, u4, u6, u8, u10` – fraction `u11` – table data `u12, u13, u14, u15, u16` – table dimension lengths

Conceptual Function Signature	Argument Summary
`y1 = interpND({ui, uf,}... ut, un...)`	`y1` – output `ui, uf` is an index and fraction pair per dimension `ut` – table data `un` – table dimension lengths
Explicit values `y1 = lookup1D(u1, u2, u3, u4)`	`y1` – output `u1` – input `u2` – breakpoint data `u3` – table data `u4` – table dimension length
Even spacing `y1 = lookup1D(u1, u2, u3, u4, u5)`	`y1` – output `u1` – input `u2` – first point of breakpoint data `u3` – spacing of breakpoints `u4` – table data `u5` – table dimension length
Explicit values `y1 = lookup2D(u1, u2, u3, u4, u5, u6, u7)`	`y1` – output `u1, u2` – input `u3, u4` – breakpoint data `u5` – table data `u6, u7` – table dimension lengths
Even spacing `y1 = lookup2D(u1, u2, u3, u4, u5, u6, u7, u8, u9)`	`y1` – output `u1, u2` – input `u3, u5` – first point of breakpoint data `u4, u6` – spacing of breakpoints `u7` – table data `u8, u9` – table dimension lengths
Explicit spacing `y1 = lookup3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)`	`y1` – output `u1, u2, u3` – input `u4, u5, u6` – breakpoint data `u7` – table data `u8, u9, u10` – table dimension lengths
Even spacing `y1 = lookup3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u2, u3` – input `u4, u6, u8` – first point of breakpoint data `u5, u7, u9` – spacing of breakpoints `u10` – table data `u11, u12, u13` – table dimension lengths
Explicit values `y1 = lookup4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u2, u3, u4` – input `u5, u6, u7, u8` – breakpoint data `u9` – table data `u10, u11, u12, u13` – table dimension lengths

Conceptual Function Signature	Argument Summary
Even spacing `y1 = lookup4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16, u17)`	`y1` – output `u1, u2, u3, u4` – input `u5, u7, u9, u11` – first point of breakpoint data `u6, u8, u10, u12` – spacing of breakpoints `u13` – table data `u14, u15, u16, u17` – table dimension lengths
Explicit values `y1 = lookup5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)`	`y1` – output `u1, u2, u3, u4, u5` – input `u6, u7, u8, u9, u10` – breakpoint data `u11` – table data `u12, u13, u14, u15, u16` – table dimension lengths
Even spacing `y1 = lookup5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16, u17, u18, u19, u20, u21)`	`y1` – output `u1, u2, u3, u4, u5` – input `u6, u8, u10, u12, u14` – first point of breakpoint data `u7, u9, u11, u13, u15` – spacing of breakpoints `u16` – table data `u17, u18, u19, u20, u21` – table dimension lengths
Explicit values `y1 = lookupND(un,..., ub,..., ut, un...)`	`y1` – output `un`, input per dimension `ub`, breakpoint per dimension `ut` – table data `un` – table dimension lengths
Even spacing `y1 = lookupND(un,..., {ufn, usn,}... ut, un...)`	`y1` – output `un` – input per dimension `ufn` – first point of breakpoint data per dimension `usn` – spacing of breakpoint per dimension `ut` – table data `un` – table dimension lengths
`y1 = lookupND_Direct(u1, u2,...ui, ui+1)`	`y1` – output `u1...ui` – input `ui+1` – table data
Explicit values `y1, y2 = prelookup(u1, u2, u3)`	`y1` – index `y2` – fraction `u1` – input `u2` – breakpoint data `u3` – number of breakpoints

Conceptual Function Signature	Argument Summary
Evenly spaced `y1, y2 = prelookup(u1, u2, u3, u4)`	`y1` – index `y2` – fraction `u1` – input `u2` – first point of breakpoint data `u3` – spacing of breakpoints `u4` – number of breakpoints

For the table and breakpoint data, the conceptual function uses matrix arguments by default. If the lookup table object has the parameter **Allow multiple instances of this type to have different table and breakpoint sizes** enabled, the conceptual function uses pointer arguments for the table and breakpoint data. To replace a lookup table function that uses pointer arguments, you must create the entry programmatically.

When defining a table entry programmatically, you might also need to change the values of required (primary) and optional algorithm parameters.

- Set values for required parameters to achieve code replacement.
- If you do not set a value for an optional parameter, the algorithm parameter software applies `don't care`. The code replacement software ignores the parameter while searching for matches.

Interactively Develop a Code Replacement Library

This example shows how to specify a code replacement table entry for a lookup table algorithm `interp1D` by using the Code Replacement Tool.

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File > New Table**.

   b. In the right pane, name the table `crl_LUT`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File > New entry > Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `lookup1D` . Set these algorithm parameters as shown in the crtool snapshot below:

   - **Extrapolation method** — Linear
   - **Interpolation method** — Linear point-slope
   - **Integer rounding mode** — Round
   - **Saturate on overflow** — off
   - **Index search method** — Linear search
   - **Remove out-of-range input protection** — off
   - **Support tunable table size** — off
   - **Use last table value for inputs at or above last breakpoint** — off
   - **Begin index search using previous index result** — off

- **Use algorithms optimized for row-major array layout** — off

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument `u1` with the **Data Type** of `double` and `u4` , with the **Data Type** of `uint32` and the **Argument Type** of `Scalar`.

Specify the **Argument Type** of input arguments `u2` and `u3` as `Matrix` and **Data Type** as `double`. Set the **Algorithm parameters** as shown in the figure below.

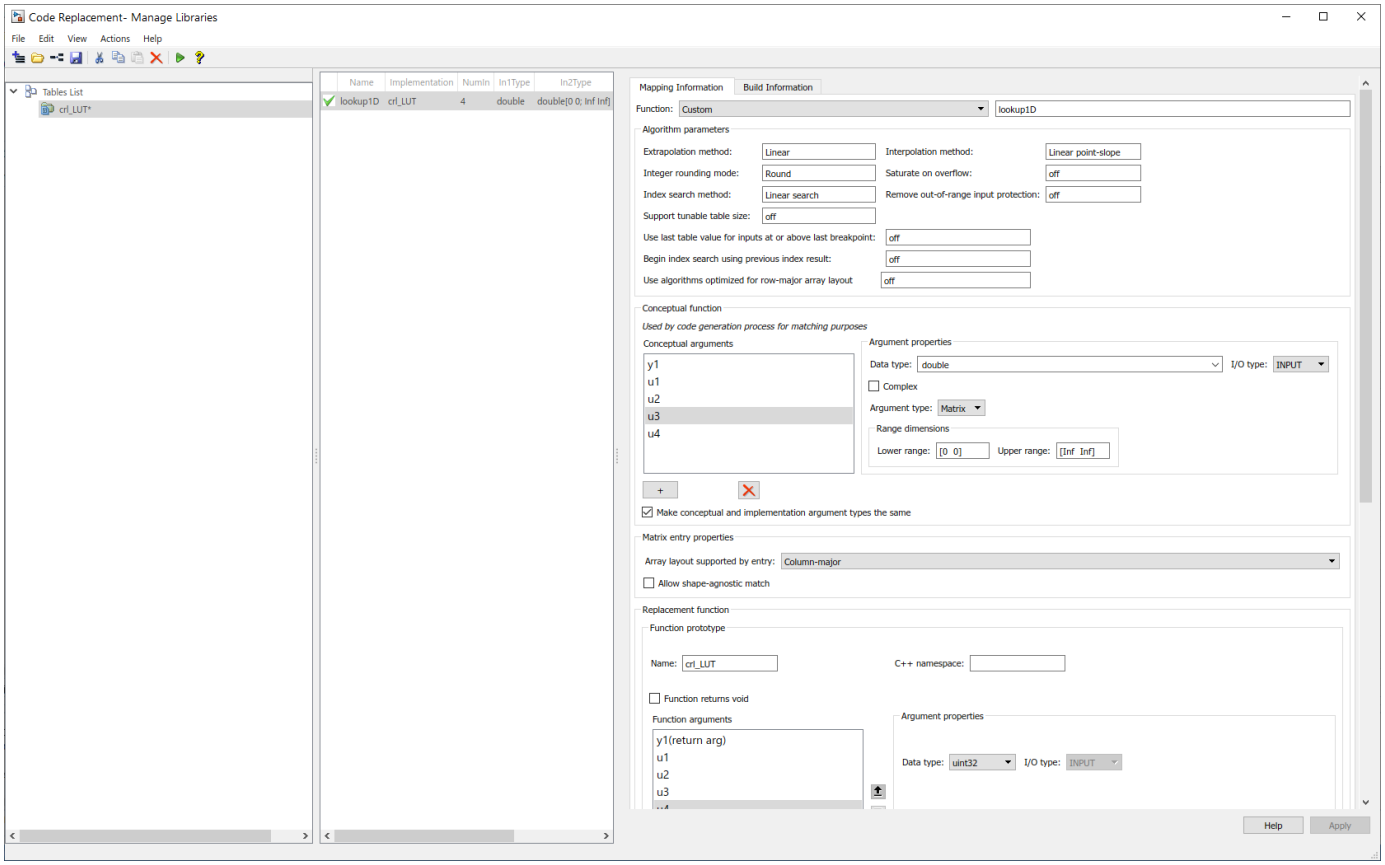
6 Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

Add the return argument `y1` and the arguments `u1, u2, u3` and `u4`.

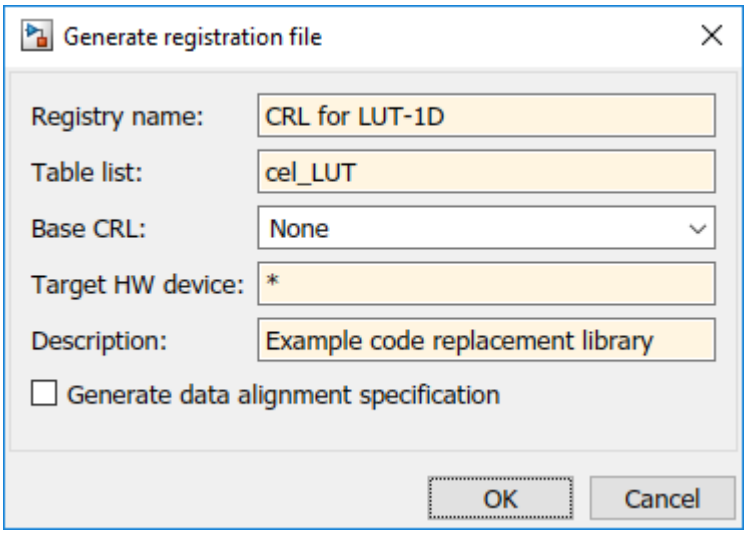
Specify a **Name** for the replacement function under **Function prototype**.

7 Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8 Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File > Save table > Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programatically develop a code replacement library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = my_lookup_replacement_table
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = my_lookup_replacement_table
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = my_lookup_replacement_table()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
          'Key',                      'lookup1D', ...
          'Priority',                 100, ...
          'ImplementationName',       'my_Lookup1D_Repl', ...
          'ImplementationHeaderFile', 'myLookup1D.h', ...
          'ImplementationSourceFile', 'myLookup1D.c', ...
          'GenCallback',              'RTW.copyFileToBuildDir');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = my_lookup_replacement_table()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
          'Key',                      'lookup1D', ...
          'Priority',                 100, ...
          'ImplementationName',       'my_Lookup1D_Repl', ...
          'ImplementationHeaderFile', 'myLookup1D.h', ...
          'ImplementationSourceFile', 'myLookup1D.c', ...
          'GenCallback',              'RTW.copyFileToBuildDir');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

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

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

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

```
arg = hEntry.getTflArgFromString('u4','uint32');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable = my_lookup_replacement_table()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
          'Key',                      'lookup1D', ...
          'Priority',                 100, ...
          'ImplementationName',       'my_Lookup1D_Repl', ...
          'ImplementationHeaderFile', 'myLookup1D.h', ...
          'ImplementationSourceFile', 'myLookup1D.c', ...
          'GenCallback',              'RTW.copyFileToBuildDir');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

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

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

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

arg = hEntry.getTflArgFromString('u4','uint32');
addConceptualArg(hEntry, arg);

algParams = getAlgorithmParameters(hEntry);
algParams.InterpMethod = 'Linear';
algParams.ExtrapMethod = 'Clip';
algParams.UseRowMajorAlgorithm = 'off';
algParams.RndMeth = 'Round';
algParams.IndexSearchMethod = 'Linear search';
algParams.UseLastTableValue = 'off';
algParams.RemoveProtectionInput = 'off';
algParams.SaturateOnIntegerOverflow = 'off';
algParams.SupportTunableTableSize = 'off';
algParams.BPPower2Spacing = 'off';
algParams.BeginIndexSearchUsingPreviousIndexResult = 'off';
setAlgorithmParameters(hEntry, algParams);


%% Create the Implementation Representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

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

arg = hEntry.getTflArgFromString('u2','double*');
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u3','double*');
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u4','uint32');
hEntry.Implementation.addArgument(arg);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = my_lookup_replacement_table
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for LUT 1D function replacement';
this(1).TableList = {'my_lookup_replacement_table.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Generate Code by Using Lookup Table Code Replacement Library

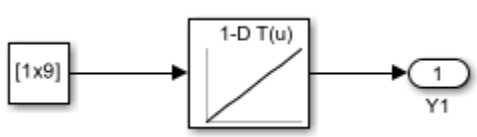
In this section, you generate code using the code replacement library that you created in the first two sections of this example. A Lookup table function is replaced with a call to a custom function in the generated code. This example does not provide an implementation function. Write your own implementation.

Example Model

Open the model `crl_LUTfunction` for configuring the code replacement library.

```
open_system('crl_LUTfunction');
```

```
copyfile LutFnRtwTargetInfo.txt rtwTargetInfo.m
```



Run the MATLAB customization file to create a code replacement table and register the file. The customization file has already been executed. Run the sl_refresh_customizations function to load the library.

```
sl_refresh_customizations;
```

Enable Code Replacement Library

1. Open the Configuration Parameters dialog box.
2. On the **Interface** pane, set **Code Replacement Library** by clicking **Select** and adding `CRL for LUT function replacement` to the **Selected code replacement libraries - prioritized** list pane.

```
%
% Alternatively, use the command-line API to enable the code replacement:
set_param('crl_LUTfunction', 'CodeReplacementLibrary', 'CRL for LUT function replacement');
```

Generate code from the model:

```
evalc('slbuild(bdroot)');
```

View the generated code. Here is a portion of `crl_LUTfunction.c`.

```
cfile = fullfile('crl_LUTfunction_ert_rtw','crl_LUTfunction.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void crl_LUTfunction_step(void)
{
  int32_T iU;

  /* Lookup_n-D: '<Root>/1-D Lookup Table2' incorporates:
   *  Constant: '<Root>/Constant2'
   */
  for (iU = 0; iU < 9; iU++) {
    crl_LUTfunction_Y.Y1[iU] = my_Lookup1D_Repl
      (crl_LUTfunction_ConstP.Constant2_Value[iU],
       &crl_LUTfunction_ConstP.uDLookupTable2_bp01Data[0],
       &crl_LUTfunction_ConstP.uDLookupTable2_tableData[0], 10U);
  }

  /* End of Lookup_n-D: '<Root>/1-D Lookup Table2' */
}
```

The Lookup Table function is replaced with a custom `my_Lookup1D_Repl` function.

Close the model and code generation report.

```
delete ./rtwTargetInfo.m
bdclose('crl_LUTfunction');
rtwdemoclean;
```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Define Code Replacement Library Optimizations” on page 72-36
- “Data Alignment for Code Replacement” on page 72-105
- “Reserved Identifiers and Code Replacement” on page 72-131
- “Customize Match and Replacement Process” on page 72-132
- “Develop a Code Replacement Library” on page 72-20

Data Alignment for Code Replacement

Code replacement libraries can align data objects passed into a replacement function to a specified boundary.

Code Replacement Data Alignment

You can take advantage of function implementations that require aligned data to optimize application performance. To configure data alignment for a function implementation:

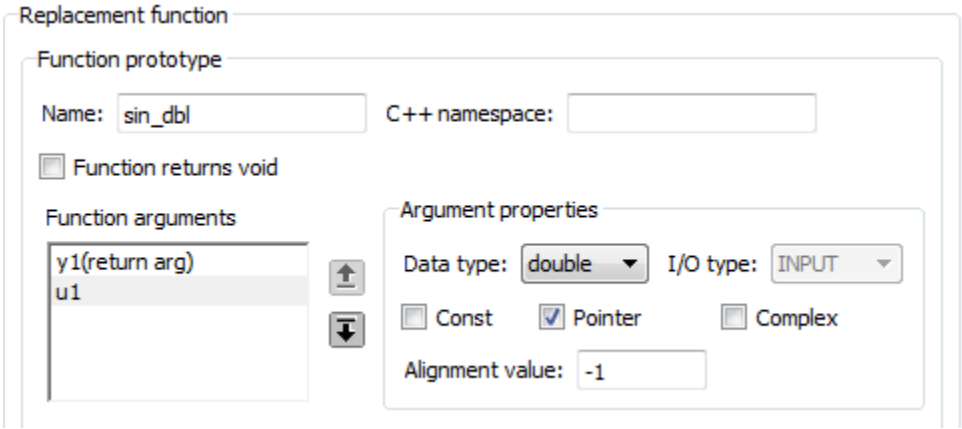
1. Specify the data alignment requirements in a code replacement entry. Specify alignment separately for each implementation function argument or collectively for all function arguments. See “Specify Data Alignment Requirements for Function Arguments” on page 72-105.
2. Specify the data alignment capabilities and syntax for one or more compilers. Include the alignment specifications in a library registration entry in the `rtwTargetInfo.m` file. See “Provide Data Alignment Specifications for Compilers” on page 72-106.
3. Register the library containing the table entry and alignment specification object.
4. Configure the code generator to use the code replacement library and generate code. Observe the results.

For examples, see “Basic Example of Code Replacement Data Alignment” on page 72-109 and the “Data Alignment for Function Implementations” section of the “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31 example page.

Specify Data Alignment Requirements for Function Arguments

To specify the data alignment requirement for an argument in a code replacement entry:

- If you are defining a replacement function in a code replacement table registration file, create an argument descriptor object (`RTW.ArgumentDescriptor`). Use its `AlignmentBoundary` property to specify the required alignment boundary and assign the object to the argument `Descriptor` property.
- If you are defining a replacement function using the **Code Replacement Tool**, on the **Mapping Information** tab, in the **Argument properties** section for the replacement function, enter a value for the **Alignment value** parameter.



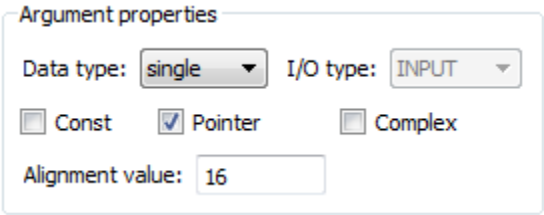
The `AlignmentBoundary` property (or **Alignment value** parameter) specifies the alignment boundary for data passed to a function argument, in number of bytes. The `AlignmentBoundary` property is valid only for addressable objects, including matrix and pointer arguments. It is not applicable for value arguments. Valid values are:

- `-1` (default) — If the data is a `Simulink.Bus`, `Simulink.Signal`, or `Simulink.Parameter` object, specifies that the code generator determines an optimal alignment based on usage. Otherwise, specifies that there is not an alignment requirement for this argument.
- Positive integer that is a power of 2 — Specifies number of bytes in the boundary. The starting memory address for the data allocated for the function argument is a multiple of the specified value. If you specify an alignment boundary that is less than the natural alignment of the argument data type, the alignment directive is emitted in the generated code. However, the target compiler ignores the directive.

The following code specifies the `AlignmentBoundary` for an argument as 16 bytes.

```
hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
arg = getTflArgFromString(hLib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);
```

The equivalent alignment boundary specification in the Code Replacement Tool dialog box is in this figure.



Note If your model imports `Simulink.Bus`, `Simulink.Parameter`, or `Simulink.Signal` objects, specify an alignment boundary in the object properties, using the **Alignment** property. For more information, see `Simulink.Bus`, `Simulink.Parameter`, and `Simulink.Signal`.

Provide Data Alignment Specifications for Compilers

To support data alignment in generated code, describe the data alignment capabilities and syntax for your compilers in the code replacement library registration. Provide one or more alignment specifications for each compiler in a library registry entry.

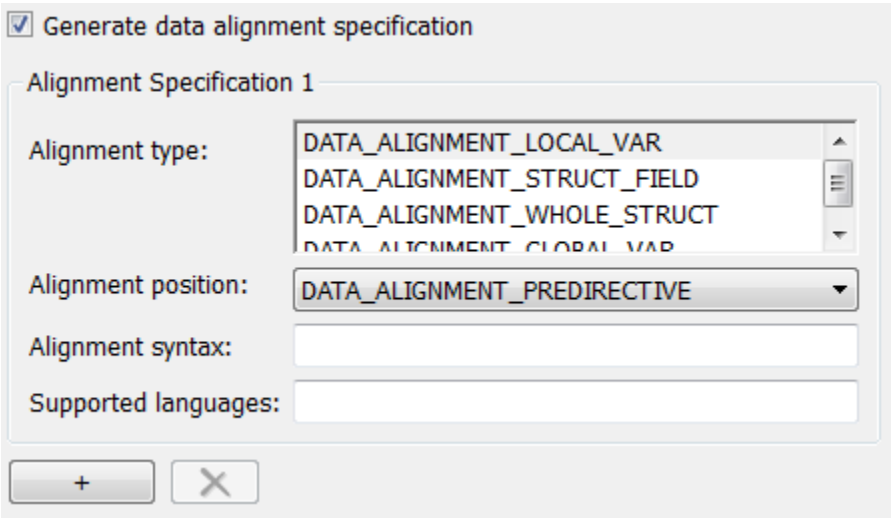
To describe the data alignment capabilities and syntax for a compiler:

- If you are defining a code replacement library registration entry in a `rtwTargetInfo.m` customization file, add one or more `AlignmentSpecification` objects to an `RTW.DataAlignment` object. Attach the `RTW.DataAlignment` object to the `TargetCharacteristics` object of the registry entry.

The `RTW.DataAlignment` object also has the property `DefaultMallocAlignment`, which specifies the default alignment boundary, in bytes, that the compiler uses for dynamically allocated memory. If the code generator uses dynamic memory allocation for a data object involved in a code replacement, this value determines if the memory satisfies the alignment requirement of the replacement. If not, the code generator does not use the replacement. The default value for `DefaultMallocAlignment` is `-1`, indicating that the default alignment boundary used for dynamically allocated memory is unknown. In this case, the code generator uses the natural alignment of the data type to determine whether to allow a replacement.

Additionally, you can specify the alignment boundary for complex types by using the `addComplexTypeAlignment` function.

- If you are generating a customization file function using the Code Replacement Tool, fill out the following fields for each compiler.



Click the plus (+) symbol to add additional compiler specifications.

For each data alignment specification, provide the following information.

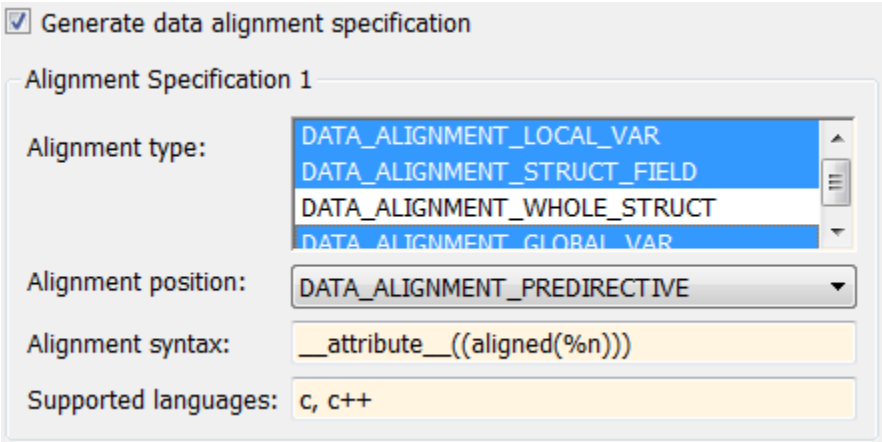
Alignment-Specification Property	Dialog Box Parameter	Description
`AlignmentType`	**Alignment type**	Cell array of predefined enumerated strings, specifying which types of alignment this specification supports. • `DATA_ALIGNMENT_LOCAL_VAR` — Local variables. • `DATA_ALIGNMENT_GLOBAL_VAR` — Global variables. • `DATA_ALIGNMENT_STRUCT_FIELD` — Individual structure fields. • `DATA_ALIGNMENT_WHOLE_STRUCT` — Whole structure, with padding (individual structure field alignment, if specified, is favored and takes precedence over whole structure alignment). Each alignment specification must specify at least `DATA_ALIGNMENT_GLOBAL_VAR` and `DATA_ALIGNMENT_STRUCT_FIELD`.
`AlignmentPosition`	**Alignment position**	Predefined enumerated string specifying the position in which you must place the compiler alignment directive for alignment type `DATA_ALIGNMENT_WHOLE_STRUCT`: • `DATA_ALIGNMENT_PREDIRECTIVE` — The alignment directive is emitted before `struct st_tag{…}`, as part of the type definition statement (for example, MSVC). • `DATA_ALIGNMENT_POSTDIRECTIVE` — The alignment directive is emitted after `struct st_tag{…}`, as part of the type definition statement (for example, gcc). • `DATA_ALIGNMENT_PRECEDING_STATEMENT` — The alignment directive is emitted as a standalone statement immediately preceding the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. • `DATA_ALIGNMENT_FOLLOWING_STATEMENT` — The alignment directive is emitted as a standalone statement immediately following the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. For alignment types other than `DATA_ALIGNMENT_WHOLE_STRUCT`, code generation uses alignment position `DATA_ALIGNMENT_PREDIRECTIVE`.

Alignment-Specification Property	Dialog Box Parameter	Description
`AlignmentSyntax-Template`	**Alignment syntax**	Specifies the alignment directive string that the compiler supports. The string is registered as a syntax template that has placeholders in it. These placeholders are supported: • `%n` — Replaced by the alignment boundary for the replacement function argument. • `%s` — Replaced by the aligned symbol, usually the identifier of a variable. For example, for the gcc compiler, you can specify `__attribute__((aligned(%n)))`, or for the MSVC compiler, `__declspec(align(%n))`.
`SupportedLanguages`	**Supported languages**	Cell array specifying the languages to which this alignment specification applies, among `c` and `c++`. Sometimes alignment syntax and position differ between languages for a compiler.

Here is a data alignment specification for the GCC compiler:

```
da = RTW.DataAlignment;

as = RTW.AlignmentSpecification;
as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                    'DATA_ALIGNMENT_STRUCT_FIELD', ...
                    'DATA_ALIGNMENT_GLOBAL_VAR'};
as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
as.AlignmentPosition = 'DATA_ALIGNMENT_PREDIRECTIVE';
as.SupportedLanguages = {'c', 'c++'};
da.addAlignmentSpecification(as);

tc = RTW.TargetCharacteristics;
tc.DataAlignment = da;
```

Here is the corresponding specification in the **Generate customization** dialog box of the Code Replacement Tool.



Basic Example of Code Replacement Data Alignment

A simple example of the complete workflow for data alignment specified for code replacement is:

1. Create and save the following code replacement table definition file, `crl_table_mmul_4x4_single_align.m`. This table defines a replacement entry for the * (multiplication) operator, the `single` data type, and input dimensions `[4,4]`. The entry also specifies a data alignment boundary of 16 bytes for each replacement function argument. The entry expresses the requirement that the starting memory address for the data allocated for the function arguments during code generation is a multiple of 16.

```
function hLib = crl_table_mmul_4x4_single_align
%CRL_TABLE_MMUL_4x4_SINGLE_ALIGN - Describe matrix operator entry with data alignment

hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(entry, ...
    'Key',                   'RTW_OP_MUL', ...
    'Priority',              90, ...
    'ImplementationName',    'matrix_mul_4x4_s');

% conceptual arguments
createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',         'y1', ...
                                 'IOType',       'RTW_IO_OUTPUT', ...
                                 'BaseType',     'single', ...
                                 'DimRange',     [4 4]);

createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',         'u1', ...
                                 'BaseType',     'single', ...
                                 'DimRange',     [4 4]);

createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',         'u2', ...
                                 'BaseType',     'single', ...
                                 'DimRange',     [4 4]);

% implementation arguments
arg = getTflArgFromString(hLib, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hLib, 'y1','single*');
arg.IOType = 'RTW_IO_OUTPUT';
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hLib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hLib, 'u2','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

hLib.addEntry(entry);
```

2. Create and save the following registration file, `rtwTargetInfo.m`. If you want to compile the code generated in this example, first modify the `AlignmentSyntaxTemplate` property for the compiler that you use. For example, for the MSVC compiler, replace the gcc template specification `__attribute__((aligned(%n)))` with `__declspec(align(%n))`.

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

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

end % End of RTWTARGETINFO

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

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

  % add the alignment specification object
  da = RTW.DataAlignment;
  da.addAlignmentSpecification(as);

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

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

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

end % End of LOCCRLREGFCN
```

3. To register your library with code generator without having to restart MATLAB, enter this command:

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

4. Configure the code generator to use your code replacement library.
5. Generate code and a code generation report.
6. Review the code replacements. For example, check whether a multiplication operation is replaced with a `matrix_mul_4x4_s` function call. In `mmalign.h`, check whether the gcc alignment directive `__attribute__((aligned(16)))` is generated to align the function variables.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Replace MATLAB Functions with Custom Code by Using the coder.replace Function

You can develop a code replacement library to optimize the performance of MATLAB functions specified within a MATLAB function block containing a `coder.replace` function. To develop a code replacement library, use either the interactive or programmatic approach. For more information, see "Develop a Code Replacement Library" on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_matlab_fcn_coder_replace`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `calculate`.

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input argument, u1, with the **Data Type** of double and the **Argument Type** of Scalar.

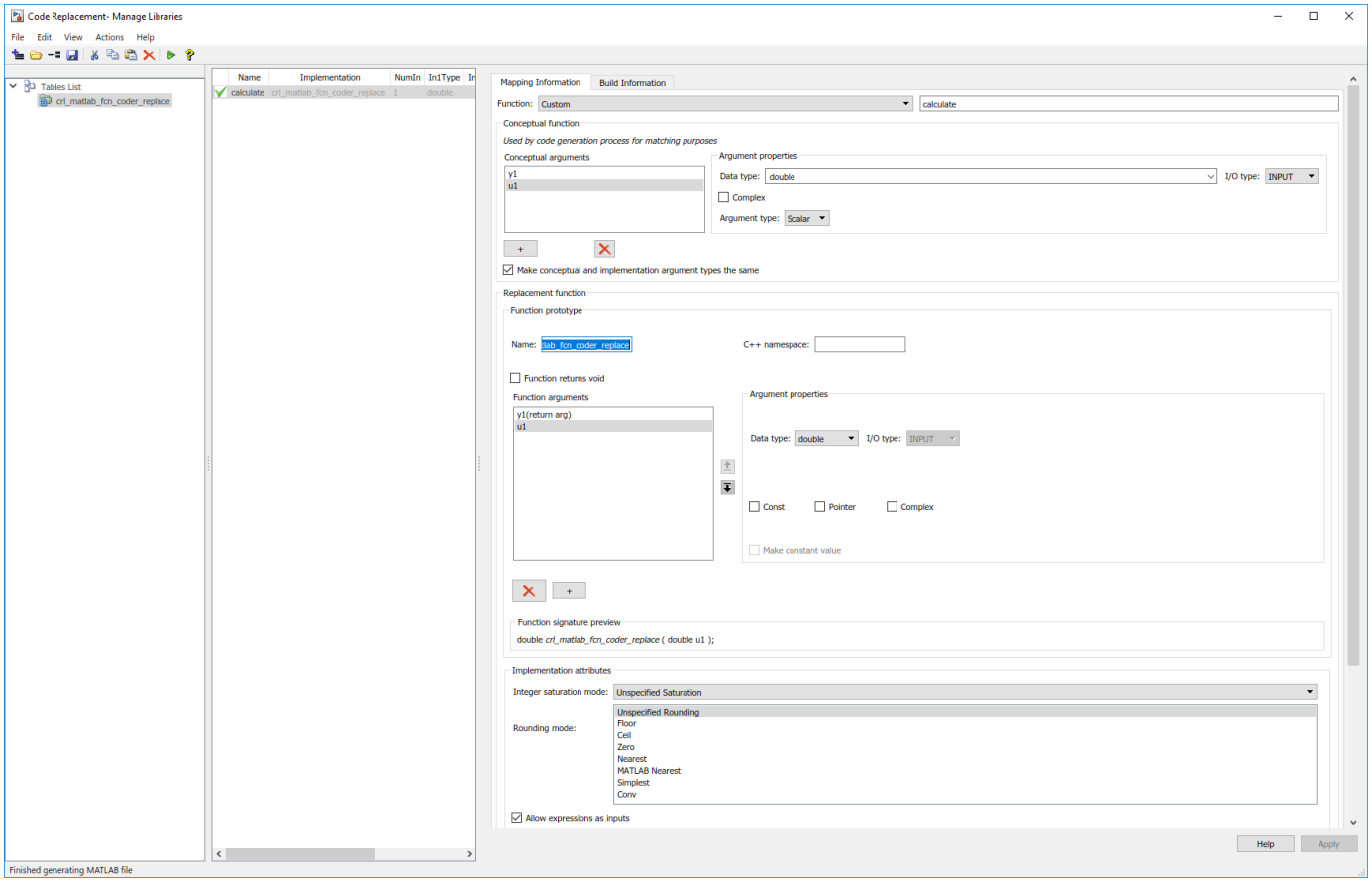
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

   Add the return argument `y1` and the arguments `u1`.

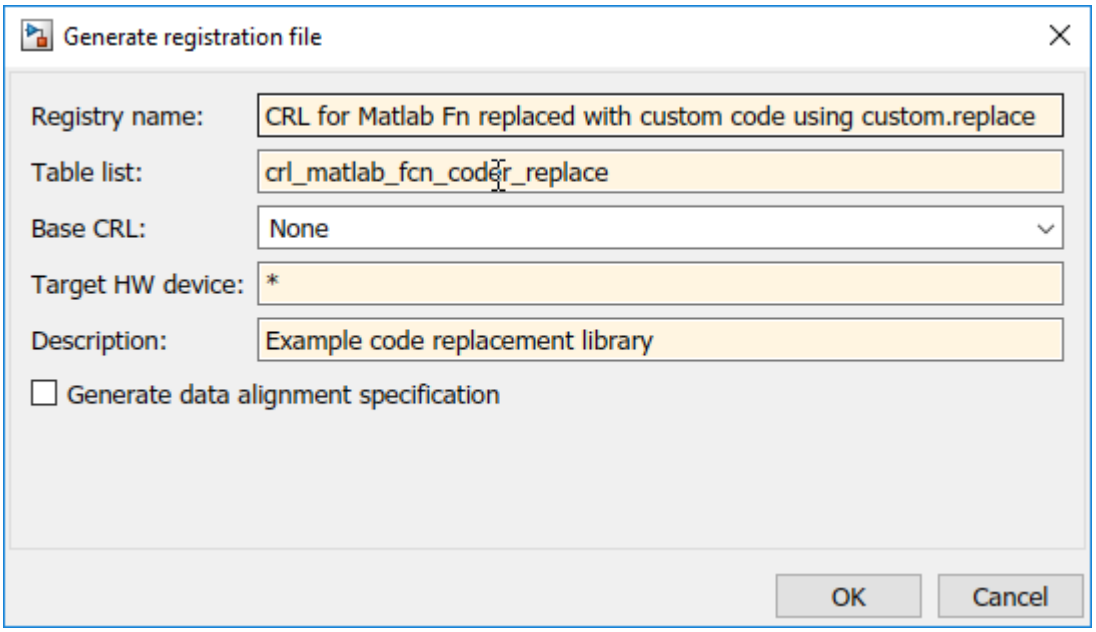
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
```

```
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hEntry, 'u1','double');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
```

```
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

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

%% Create the Conceptual Representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

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

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_matlab_fcn_coder_replace
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for matlab function with custom code using coder.replace’;
this(1).TableList = {'crl_matlab_fcn_coder_replace.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Replace coder.ceval Calls to External Functions

The `coder.ceval` function calls external C/C++ functions from code generated from MATLAB code. The code replacement software supports replacement of the function that you specify in a call to `coder.ceval`. An application of this code replacement scenario is to write generic MATLAB code that you can customize for different platforms with code replacements. A code replacement library can define hardware-specific code replacements for the function call. Use `coder.ceval` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Example Files

For the examples in “Interactively Develop a Code Replacement Library” on page 73-86 and “Programmatically Develop a Code Replacement Library” on page 72-121 you must have set up the following:

- Custom C function `my_add.c`.

```
/* my_add.c */

#include "my_add.h"

double my_add(double in1, double in2)
{
  return in1 + in2;
}
```

- Custom C header file `my_add.h`.

```
/* my_add.h */

double my_add(double in1, double in2);
```

- MATLAB function `call_my_add.m`, which uses `coder.ceval` to invoke `my_add.c`.

```
function y = call_my_add(in1, in2)  %#codegen

y=0.0;

if ~coder.target('Rtw')
% Executing in MATLAB, call MATLAB equivalent of C function my_add
  y= in1+in2;
else
% Executing in generated code, call C function my_add
  y = coder.ceval('my_add', in1, in2);
end
```

- MATLAB test function `call_my_add_test.m`, which calls `call_my_add.m`.

```
in1=10;
in2=20;
```

```
y = call_my_add(in1, in2);

disp('Output')
disp('y =')
disp(y);
```

- Replacement C function `my_add_replacement.c`.

```
/* my_add_replacement.c */

#include "my_add_replacement.h"

double my_add_replacement(double in1, double in2)
{
  return in1 + in2;
}
```

- Replacement C header file `my_add_replacement.h`.

```
/* my_add_replacement.h */

double my_add_replacement(double in1, double in2);
```

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_coder_cevals_calls`. Click **Apply**.

3. Create an entry. From the crtool menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `my_add` .

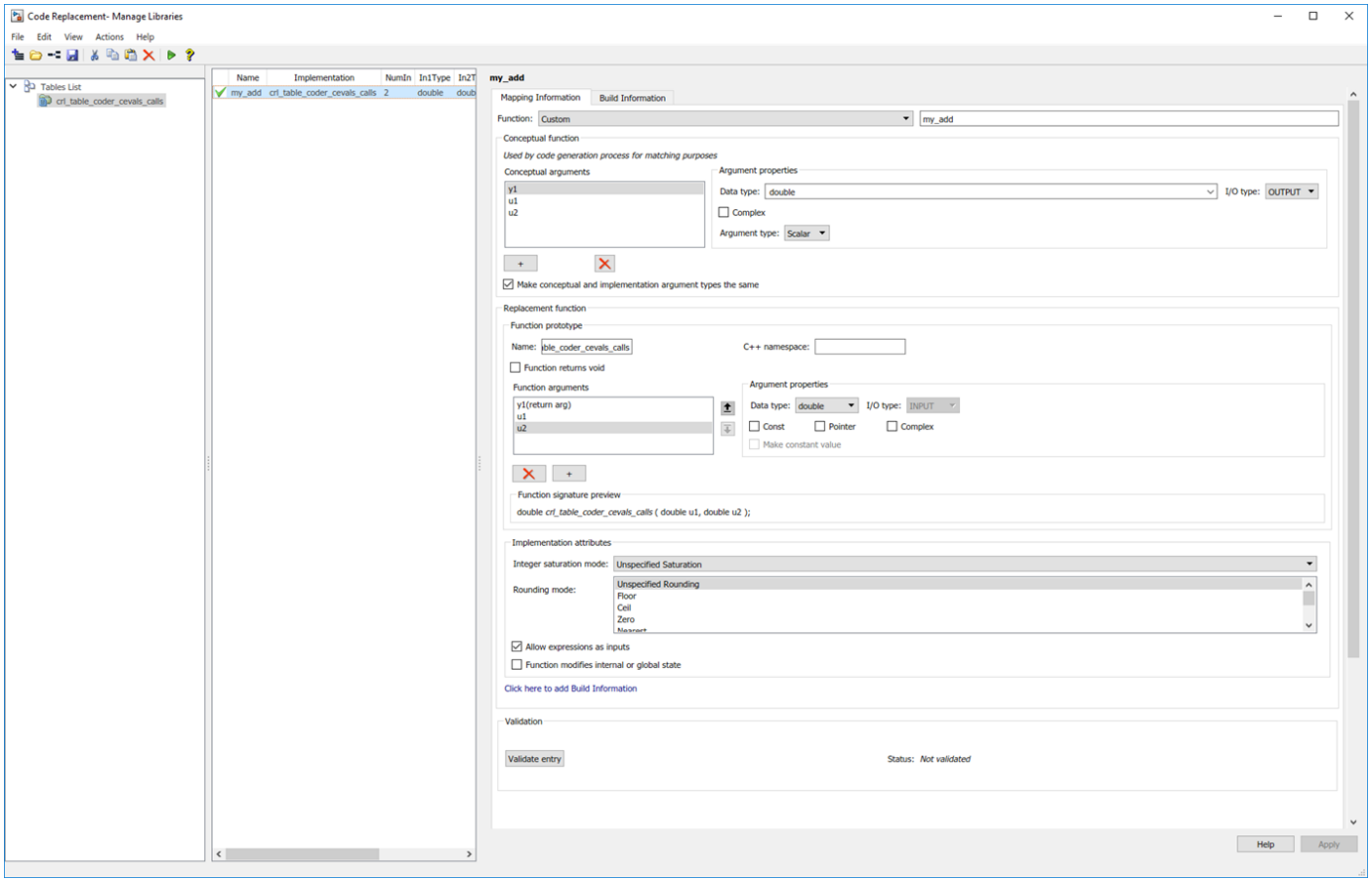
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1` and `u2` with the **Data Type** of double and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

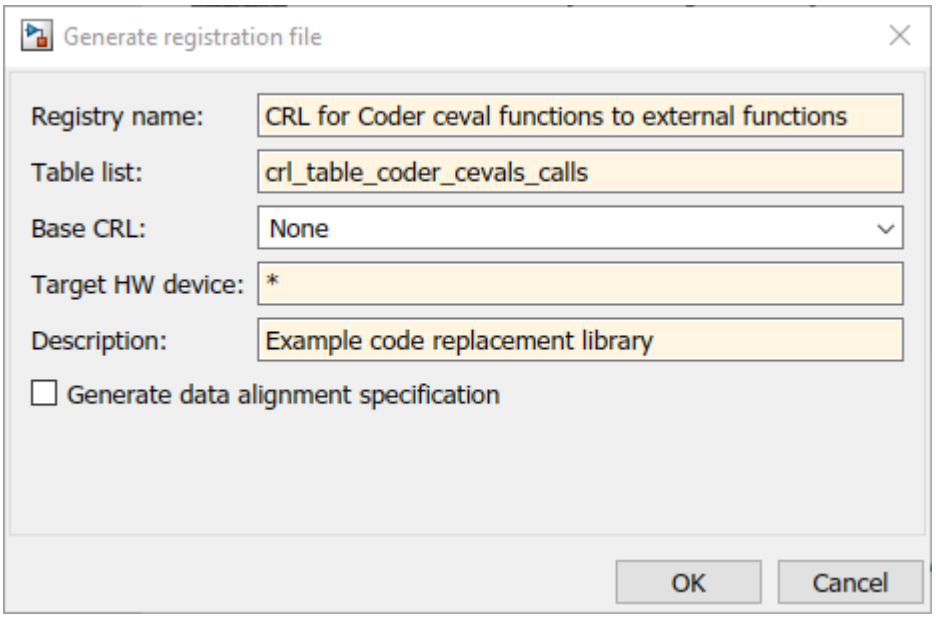
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, set the **Implementation header file** to `my_add_replacement.h`.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.addConceptualArg(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);
```

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.addConceptualArg(arg);


%% Create the implementation representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable =  crl_table_coder_cevals_calls
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for coder.ceval calls to external functions code replacement’;
   this(1).TableList = {' crl_table_coder_cevals_calls.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Math Function Code Replacement” on page 72-64
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Custom Function Code Replacement

This example shows how to replace a custom function from your model with a different function in the generated code. Use this replacement when your model uses a MATLAB Function block to call a function for simulation and you want the generated code to use a different version of the function that is optimized for your code generation target. You can replace a function in these scenarios:

- The MATLAB function uses the `coder.ceval` function to call a custom C function.
- The MATLAB function uses the `coder.replace` function to indicate that you want to replace a call to a custom MATLAB function.

To replace a custom function, define a custom entry in a code replacement library.

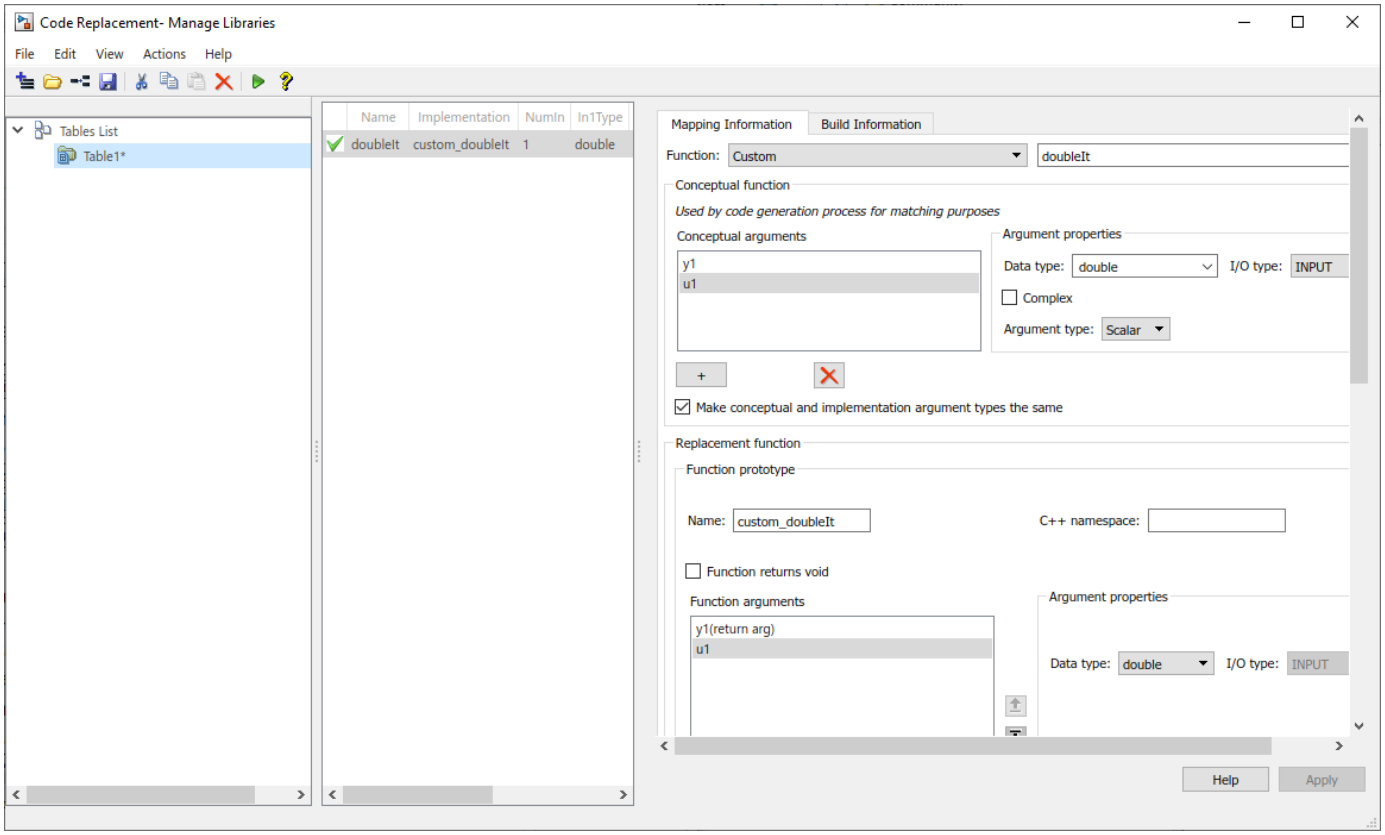
Interactively Develop a Code Replacement Library

Specify a code replacement entry for a custom C function by using the Code Replacement Tool.

1. At the MATLAB command prompt, open the Code Replacement Tool by using the `crtool` command.

   ```
   crtool
   ```

2. Create a table. From the crtool context menu, select **File** > **New Table**.
3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.
4. Create entry parameters. In the **Function** drop-down list, select `Custom`. Next to the drop-down list, specify the name of the function that you want to replace. For this example, set the function name to `doubleIt`.
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection, add the return argument y1 and the input argument u1. For both arguments, specify the **Data Type** of double and the **Argument Type** of scalar. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.
6. Create the implementation representation. In the **Replacement function** > **Function prototype** section, specify the name of the replacement function. For this example, set the name to `custom_doubleIt`. In the **Function arguments** section, add the return argument y1 and the input argument u1.
7. Optionally, specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, specify the source file `custom_doubleIt.c` and the header file `custom_doubleIt.h`.
8. Validate and save the table. Click the **Mapping Information** tab and verify that the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save** and name the table `customCRL.m`.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, specify the registry name `Custom Function CRL` and the table list `customCRL`. To use your code replacement library, refresh your current MATLAB session by using the command `sl_refresh_customizations`.

```
sl_refresh_customizations
```

10 Verify the code replacement library. At the MATLAB command prompt, open the library by using the Code Replacement Viewer and verify that the table and entry are correctly specified.

Programmatically Develop a Code Replacement Library

Programmatically specify a code replacement entry for a custom C function.

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function that has the name of your code replacement library table, which does not have arguments and returns a table object. Use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_custom()
% Create a function to call the code replacement library table
```

```
%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_custom()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this example replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_custom()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'doubleIt', ...
          'ImplementationName', 'custom_doubleIt', ...
          'ImplementationHeaderFile', 'custom_doubleIt.h', ...
          'ImplementationSourceFile', 'custom_doubleIt.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_custom()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'doubleIt', ...
          'ImplementationName', 'custom_doubleIt', ...
          'ImplementationHeaderFile', 'custom_doubleIt.h', ...
          'ImplementationSourceFile', 'custom_doubleIt.c');

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'double');
```

```
arg.IOType = 'RTW_IO_INPUT';
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_custom()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'doubleIt', ...
          'ImplementationName', 'custom_doubleIt', ...
          'ImplementationHeaderFile', 'custom_doubleIt.h', ...
          'ImplementationSourceFile', 'custom_doubleIt.c');

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'double');
arg.IOType = 'RTW_IO_INPUT';
addConceptualArg(hEntry, arg);

%% Create the implementation representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
end
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator requires for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. At the command prompt, validate the code replacement library table by calling it:

```
>> hTable = crl_table_custom
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl
```

```
this(1) = RTW.TflRegistry;
this(1).Name = 'Custom Function CRL';
this(1).TableList = {'crl_table_custom.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. At the command prompt, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Generate Code by Using Custom Function Code Replacement

Generate code using the code replacement library that you previously created. A custom C function that is called in a MATLAB Function block is replaced with a call to a different custom function in the generated code.

Example Model

Open the model `myDoubleModel` for configuring the code replacement library.

```
open_system('myDoubleModel')
```

The MATLAB Function block in the model uses the `coder.ceval` function to call the custom C function `doubleIt`.

```
function y = fcn(u)

y = 0.0;

y = coder.ceval('doubleIt',u);
```

The code replacement library replaces this function with a call to the custom function `custom_doubleIt`. For this example, the code replacement table is saved in the file `customCRL.m` and the library is registered in the registration file `rtwTargetInfo.m`.

To use the code replacement library, refresh the current MATLAB session.

```
copyfile targetInfoFile/rtwTargetInfo.m
sl_refresh_customizations
```

Enable Code Replacement Library

Configure the model to use the code replacement library and generate code.

1 Open the Configuration Parameters dialog box. On the **Modeling** tab, click **Settings**.

2 On the **Code Generation** > **Interface** pane, for the **Code replacement libraries** parameter, click **Select**. Select the code replacement library `Custom Function CRL` and click **OK**.

```
% Alternatively, use the command-line API to enable the code replacement:
set_param('myDoubleModel', 'CodeReplacementLibrary', 'Custom Function CRL');
```

Generate code for the model. Press **Ctrl+B** or open the Embedded Coder app and click **Generate Code**.

```
evalc('slbuild(bdroot)');
```

In the Code view, confirm that the step function calls the custom C function `custom_doubleIt`.

```
cfile = fullfile('myDoubleModel_ert_rtw','myDoubleModel.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void myDoubleModel_step(void)
{
  /* Outport: '<Root>/y' incorporates:
   *  Inport: '<Root>/u'
   *  MATLAB Function: '<Root>/MATLAB Function'
   */
  myDoubleModel_Y.y = custom_doubleIt(myDoubleModel_U.u);
}
```

The function `doubleIt` is replaced with the custom function `custom_doubleIt`.

See Also

Related Examples

- “Customize Match and Replacement Process” on page 72-132

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`. For example:

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

In a code replacement table, the code generator registers each function implementation name defined by a table entry as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

To register additional code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Customize Match and Replacement Process” on page 72-132
- “Define Code Replacement Library Optimizations” on page 72-36
- “Develop a Code Replacement Library” on page 72-20

Customize Match and Replacement Process

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations replace default implementations.
- Preset replacement function signatures.

It is possible that preset match criteria and preset replacement function signatures do not completely meet your function and operator replacement needs. For example:

- You want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

To add extra logic into the code replacement match and replacement process, create custom code replacement table entries. With custom entries, you can specify additional match criteria and modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

1. Create a custom code replacement entry class, derived from `RTW.TflCFunctionEntryML (for function replacement) or RTW.TflCOperationEntryML` (for operator replacement).
2. In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations that instantiate the class:
   - Add match criteria that the base class does not provide. The base class provides a match based on:
     - Argument number
     - Argument name
     - Signedness
     - Word size
     - Slope (if not specified with wildcards)
     - Bias (if not specified with wildcards)
     - Math modes, such as saturation and rounding
     - Operator or function key
   - Modify the implementation signature by adding additional arguments or setting constant input argument values. You can inject a constant value, such as an input scaling value, as an additional argument to the replacement function.
3. Create code replacement entries that instantiate the custom entry class.
4. Register a library containing the code replacement table that includes your entries.

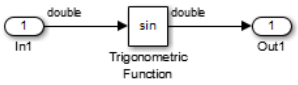
During code generation, the code replacement match process tries to match function or operator call sites with the base class of your derived entry class. If the process finds a match, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Customize Code Match and Replacement for Functions

This example shows how to use custom code replacement table entries to refine the match and replacement logic for functions. The example shows how to:

- Modify a sine function replacement only if the integer size on the current target platform is 32 bits.
- Change the replacement such that the implementation function passes in a degrees-versus-radians flag as an input argument.

1. To exercise the table entries that you create in this example, create an ERT-based model with a sine function block. For example:



   In the Inport block parameters, set the signal **Data type** to `double`. If the value selected for **Configuration Parameters** > **Hardware Implementation** > **Device type** supports an integer size other than 32, do one of the following:

   - Select a temporary target platform with a 32-bit integer size.
   - Modify the code to match the integer size of your target platform.

2. Create a class, for example `TflCustomFunctionEntry`, that is derived from the base class `RTW.TflCFunctionEntryML`. The derived class defines a `do_match` method with the signature:

```
function ent = do_match(hThis, ...
        hCSO, ...
        targetBitPerChar, ...
        targetBitPerShort, ...
        targetBitPerInt, ...
        targetBitPerLong, ...
        targetBitPerLongLong)
```

   In the `do_match` signature:

   - `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `TflCFunctionEntry` handle.
   - `hThis` is a handle to the class instance.
   - `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
   - Remaining arguments are the number of bits for various data types of the current target.

   The `do_match` method:

   - Adds required additional match criteria that the base class does not provide.
   - Makes required modifications to the implementation signature.

   In this case, the `do_match` method must match only `targetBitPerInt`, representing the number of bits in the C `int` data type for the current target, to the value 32. If the code generator finds a match, the method sets the return handle and creates and adds an input argument. The input argument represents whether units are expressed as degrees or radians, to the replacement function signature.

Alternatively, create and add the additional implementation function argument for passing a units flag in each code replacement table definition file that instantiates this class. In that case, this class definition code does not create the argument. That code sets only the argument value. For an example of creating and adding additional implementation function arguments in a table definition file, see “Customize Code Match and Replacement for Scalar Operations” on page 72-139.

```
classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok
      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This will check additional data and perhaps modify
      % the implementation function.

      ent = []; % default the return to empty, indicating the match failed.

      % Match sine function only if the target int size is 32 bits
      if targetBitPerInt == 32
        % Need to modify the default implementation, starting from a copy
        % of the standard TflCFunctionEntry.
        ent = RTW.TflCFunctionEntry(hThis);

        % If the target int size is 32 bits, the implementation function
        % takes an additional input flag argument indicating degress vs.
        % radians. The additional argument can be created and added either
        % in the CRL table definition file that instantiates this class, or
        % here in the class definition, as follows:
        createAndAddImplementationArg(ent, 'RTW.TflArgNumericConstant', ...
                                      'Name',           'u2', ...
                                      'IsSigned',       true, ...
                                      'WordLength',     32, ...
                                      'FractionLength', 0, ...
                                      'Value',          1);
      end
    end
  end
end
```

Exit the class folder and return to the previous working folder.

3 Create and save the following code replacement table definition file, `crl_table_custom_sinfcn_double.m`. This file defines a code replacement table that contains a function table entry for sine with `double` input and output. This entry instantiates the derived class from the previous step, `TflCustomFunctionEntry`.

```
function hTable = crl_table_custom_sinfcn_double

hTable = RTW.TflTable;

%% Add TflCustomFunctionEntry
fcn_entry = TflCustomFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
    'Key',                      'sin', ...
    'Priority',                 30, ...
    'ImplementationName',       'mySin', ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
     'Name',         'y1', ...
     'IOType',       'RTW_IO_OUTPUT', ...
     'DataTypeMode', 'double');
```

```
createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
     'Name',         'u1', ...
     'IOType',       'RTW_IO_INPUT', ...
     'DataTypeMode', 'double');

% TflCustomFunctionEntry class do_match method will create and add
% an implementation function argument during code generation if
% the supported integer size on the current target is 32 bits.
copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);
```

4 Check the validity of the code replacement table entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_sinfcn_double
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_sinfcn_double)
  ```

Customize Code Match and Replacement for Nonscalar Operations

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a nonscalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

This example restricts the match criteria for an element-wise multiplication replacement to entries with a specific dimension range. When a match occurs, the custom `do_match` method modifies the replacement signature to pass the number of elements into the function.

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1 Create the replacement function source and header files. For this example, create a directory named `src` and create these files in the directory:

- `myMulImplLib.c` which contains this code:

  ```
  #include "myMulImplLib.h"

  void myElemMul_s32(int32_T* u1, int32_T* u2, int32_T* y1, uint32_T numElements)
  {
      int idx;
      for(idx = 0; idx<numElements; ++idx)
      {
          y1[idx] = u1[idx] * u2[idx];
      }
  }
  ```

- `myMulImplLib.h` which contains this code:

  ```
  #ifndef __myMulImplLib_h
  #define __myMulImplLib_h

  #include "rtwtypes.h"

  void myElemMul_s32(int32_T* u1, int32_T* u2, int32_T* y1, uint32_T numElements);

  #endif /*__myMulImplLib_h*/
  ```

2 Create a class, for example `MyElemMultEntry`, which is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

```
function ent = do_match(hThis, ...
    hCSO, ...
    targetBitPerChar, ...
    targetBitPerShort, ...
    targetBitPerInt, ...
    targetBitPerLong, ...
    targetBitPerLongLong)
```

In the `do_match` signature:

- `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
- `hThis` is the handle to the derived instance.
- `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
- Remaining arguments are the number of bits for various data types of the current target.

The `do_match` method:

- Adds match criteria that the base class does not provide.
- Makes changes to the implementation signature.

The `do_match` method relies on the base class for checking data types and dimension ranges. If the code generator finds a match, `do_match`:

- Sets the return handle.
- Uses the conceptual arguments to compute the number of elements in the array. In the replacement entry returned, sets the value of the constant implementation argument as the number of elements of the array.
- Updates the code replacement entry such that it matches CSOs that have the same argument dimensions.

```
classdef MyElemMultEntry < RTW.TflCOperationEntryML
    methods
        function obj = MyElemMultEntry(varargin)
            mlock;
            obj@RTW.TflCOperationEntryML(varargin{:});
        end

        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong, ... %#ok
            targetBitPerLongLong ) %#ok

            %  Fourth implementation arg represents number of elements for producing matches.
            assert(strcmp(hThis.Implementation.Arguments(4).Name,'numElements'));

            ent = RTW.TflCOperationEntry(hThis);

            % Calculate number of elements and set value of injected constant.
            ent.Implementation.Arguments(4).Value = prod(hCSO.ConceptualArgs(1).DimRange(1,:));
```

```
            % Since implementation has been modified for specific DimRange, update
            % returned entry to match similar CSOs only.
            for idx =1:3
                ent.ConceptualArgs(idx).DimRange = hCSO.ConceptualArgs(idx).DimRange;
            end
        end
    end
end
```

3 Create and save the following code replacement table definition file, `myElemMultCrlTable.m`. This file defines a code replacement table that contains an operator entry generator for element-wise multiplication. The table entry:

- Instantiates the derived class `myElemMultEntry` from the previous step.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Creates conceptual arguments `y1`, `u1`, and `u2`. The argument class `RTW.TflArgMatrix` specifies matrix arguments to match. The three arguments are set up to match 2-dimensional matrices with at least two elements in each dimension.
- Calls the `getTflArgFromString` function to create a return value and four implementation arguments. Arguments `u1` and `u2` are the operands, `y1` is the product, and the fourth argument is the number of elements.

  Alternatively, the `do_match` method of the derived class `myElemMultEntry` can create and add the implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.
- Calls `addEntry` to add the entry to a code replacement table.

```
function hLib = myElemMultCrlTable

libPath = fullfile(fileparts(which(mfilename)),'src');

hLib = RTW.TflTable;
%---------- entry: RTW_OP_ELEM_MUL -----------
hEnt = MyElemMultEntry;
hEnt.setTflCOperationEntryParameters( ...
          'Key', 'RTW_OP_ELEM_MUL', ...
          'Priority', 100, ...
          'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
          'ImplementationName', 'myElemMul_s32', ...
          'ImplementationSourceFile', 'myMulImplLib.c', ...
          'ImplementationSourcePath', libPath, ...
          'ImplementationHeaderFile', 'myMulImplLib.h', ...
          'ImplementationHeaderPath', libPath, ...
          'SideEffects', true, ...
          'GenCallback','RTW.copyFileToBuildDir');

% Conceptual Args

arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);
```

```
arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

% Implementation Args

arg = hEnt.getTflArgFromString('unused','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','int32*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','int32*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('y1','int32*');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('numElements','uint32',0);
hEnt.Implementation.addArgument(arg);

hLib.addEntry( hEnt );
```

4 Check the validity of the code replacement table entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = myElemMultCrlTable
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(myElemMultCrlTable)
  ```

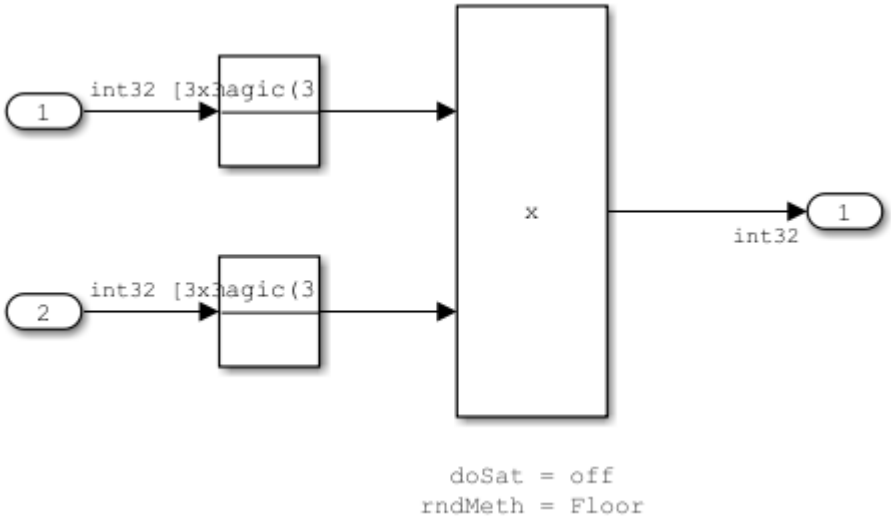
5 Register the code replacement library. Create this file named `rtwTargetInfo.m`.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'My Element-Wise Multiplication CRL';
this(1).TableList = {'myElemMultCrlTable'};
this(1).BaseTfl = '';
this(1).TargetHWDeviceType = {'*'};
this(1).Description = '';
```

6 Refresh your current MATLAB session by using the command `sl_refresh_customizations`.

```
sl_refresh_customizations
```

7 Create the model `myElemMul.slx`, which will use the code replacement library.



The model contains:

- Two Input ports with **Data type** `Int32` and **Port dimensions** set to `[3 3]`
- Two Initial condition blocks with **Initial condition** set to `magic(3)`
- A Product block with **Integer rounding mode** set to `Floor` and **Saturate on integer overflow** de-selected
- One Output port from the output of the Product block

Set these configuration parameters:

- **System target file** — `ert.tlc`
- **Code replacement libraries** — `My CRL Lib`

When you generate code from the model, the generated code calls the replacement source code.

Customize Code Match and Replacement for Scalar Operations

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a scalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

For example:

- When fraction lengths are within a specific range, replace an operator with a fixed-point implementation function.
- When a match occurs, modify the replacement function signature based on compile-time information, such as passing fraction-length values into the function.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1. Create a class, for example `TflCustomOperationEntry`, that is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

```
function ent = do_match(hThis, ...
        hCSO, ...
        targetBitPerChar, ...
        targetBitPerShort, ...
        targetBitPerInt, ...
        targetBitPerLong, ...
        targetBitPerLongLong)
```

In the `do_match` signature:

- `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
- `hThis` is the handle to the class instance.
- `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
- Remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds match criteria that the base class does not provide. The method makes modifications to the implementation signature. In this case, the `do_match` method relies on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to value 0. If the code generator finds a match, `do_match`:

- Sets the return handle.
- Removes slope and bias wild cards from the conceptual arguments (the match is for specific slope and bias values).
- Writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed
```

```
      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = 0;
        end

        % Set the fraction-length values in the implementation function.
        ent.Implementation.Arguments(3).Value = ...
           -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
        ent.Implementation.Arguments(4).Value = ...
           -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
        ent.Implementation.Arguments(5).Value = ...
           -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
      end
    end
  end
end
```

Exit the class folder and return to the previous working folder.

2 Create and save the following code replacement table definition file, `crl_table_custom_add_ufix32.m`. This file defines a code replacement table that contains a single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. The table entry:

- Instantiates the derived class `TflCustomOperationEntry` from the previous step. If you want to replace word sizes and signedness attributes, you can use the same derived class, but not the same entry, because you cannot use a wild card with the `WordLength` and `IsSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, add five other distinct entries. To use different implementation functions for saturation and rounding modes other than overflow and round to floor, add entries for those match permutations.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Calls the `createAndAddConceptualArg` function to create conceptual arguments `y1`, `u1`, and `u2`.
- Calls `createAndSetCImplementationReturn` and `createAndAddImplementationArg` to define the signature for the replacement function. Three of the calls to `createAndAddImplementationArg` create implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead, the `do_match` method of the derived class `TflCustomOperationEntry`

can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.

- Calls `addEntry` to add the entry to a code replacement table.

```
function hTable = crl_table_custom_add_ufix32

hTable = RTW.TflTable;

% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
```

```
    'FractionLength', 0);

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

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_out', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

addEntry(hTable, op_entry);
```

3 Check the validity of the operator entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_add_ufix32
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_add_ufix32)
  ```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6

- “Define Code Replacement Library Optimizations” on page 72-36
- “Develop a Code Replacement Library” on page 72-20

Scalar Operator Code Replacement

This example shows how to develop a code replacement library to optimize the performance of scalar operators by providing information on how to define code replacement for a `multiply` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_scalar`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Multiply`.

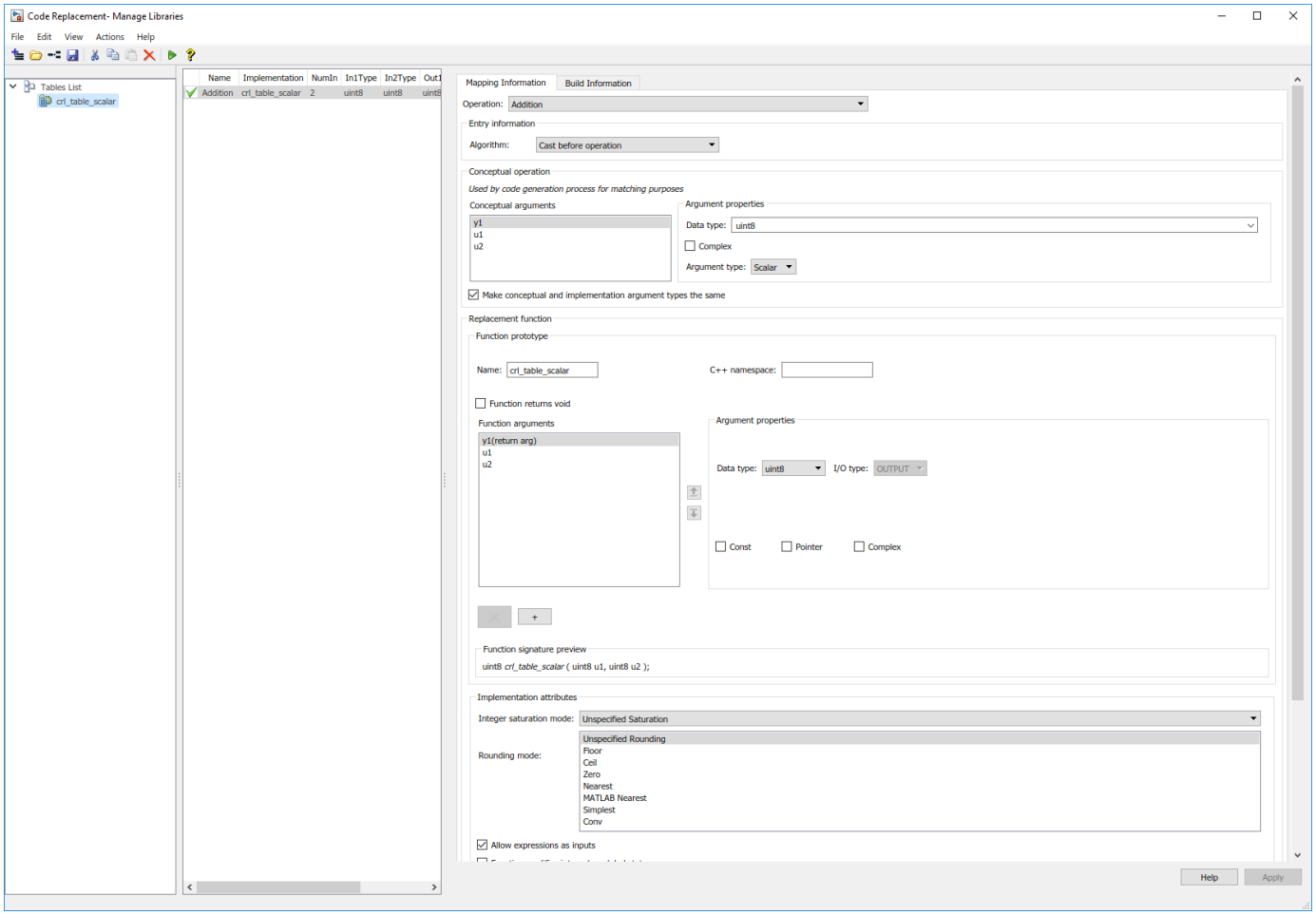
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1` and `u2` with the **Data Type** of `uint8` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

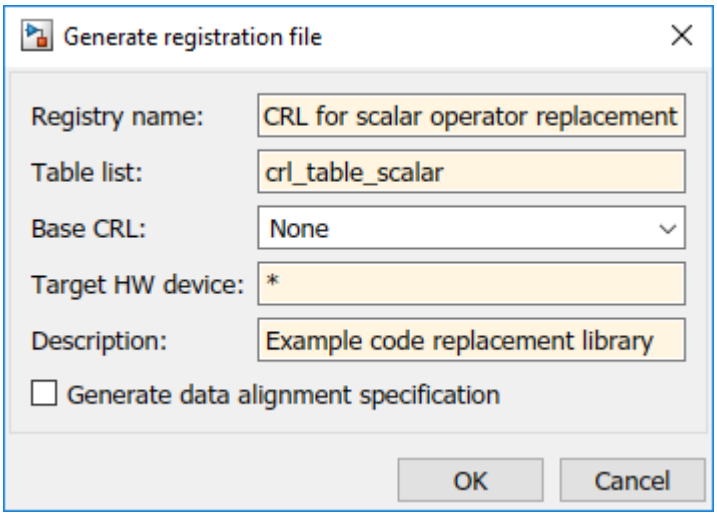
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_scalar()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function .

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(hEntry, arg );
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(hEntry, arg );

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_scalar
```

9 Create a registration file for the code replacement library with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for scalar operator replacement';
this(1).TableList = {'crl_table_scalar.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Shift Left Operations and Code Replacement” on page 72-224

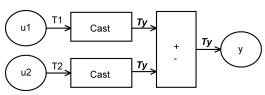
Addition and Subtraction Algorithm Types for Code Replacement

When you replace an addition or subtraction operator, your library function algorithm must match the algorithm type of the operator that you want to replace. Addition and subtraction operations are either cast-before-operation or cast-after-operation algorithms. First, determine which type of algorithm your library function implements. Then, to replace an operator in the generated code, determine how the code generator classifies the operator algorithm. Match your library function algorithm to the classification. For an example, see “Replace Addition and Subtraction Operator Code” on page 72-153.

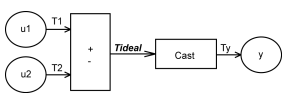
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- Cast-before-operation (CBO), default — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.



- Cast-after-operation (CAO) — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operand to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.



Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`double`	`double`	`double`	CBO, CAO
`double`	`double`	`double`	`single`	—
`double`	`double`	`single`	`double`	—

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`double`	`single`	`single`	CBO
`double`	`single`	`double`	`double`	CBO, CAO
`double`	`single`	`double`	`single`	—
`double`	`single`	`single`	`double`	—
`double`	`single`	`single`	`single`	CBO
`single`	`single`	`single`	`single`	CBO, CAO
`single`	`single`	`single`	`double`	—
`single`	`single`	`double`	`single`	—
`single`	`single`	`double`	`double`	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: • Operation type is `double`. • Operation type is `single` and input types are `single` or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: • Accumulator type equals output type (`Tacc == Tout`). • Output type is a superset of input types (`Tacc >= {Tin1, Tin2}`) and accumulator type is a superset of output type (`Tacc >= Tout`). • Operation does not incur range or precision loss.
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see “Addition” or “Subtraction” in “Fixed-Point Operator Code Replacement” on page 73-134 (for MATLAB code) or “Fixed-Point Operator Code Replacement” on page 72-188 (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 –> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Limitations

- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.

- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A)(cast B)(u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (`+`) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

See Also

More About

- “Replace Addition and Subtraction Operator Code” on page 72-153
- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Shift Left Operations and Code Replacement” on page 72-224

Replace Addition and Subtraction Operator Code

This example shows how to develop a code replacement library to optimize the performance of addition and subtraction operators by providing information on how to define code replacement for a `addition` operator. To develop a code replacement library use either the interactive or programmatic approach. When you replace an addition or subtraction operator, your library function algorithm must match the algorithm type of the operator that you want to replace. For more information, see "Addition and Subtraction Algorithm Types for Code Replacement" on page 72-150.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_addition`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Function** drop-down list, select `Addition`. Algorithm information appears in the crtool. For this example, select the parameter to `Cast before operation`.

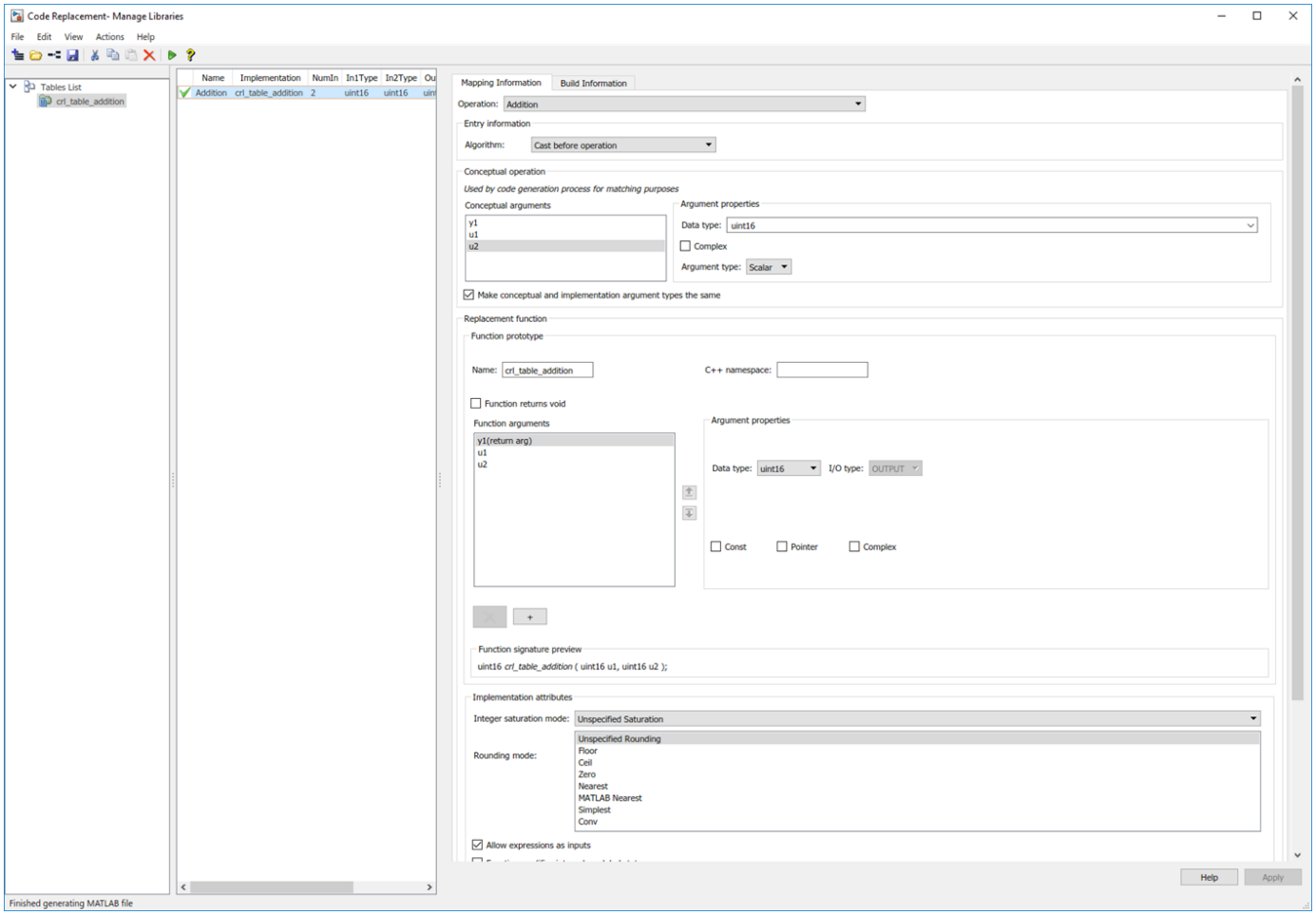
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, with the **Data Type** of `uint8` and the **Argument Type** of Scalar, the input arguments, u1 and u2 with the **Data Type** of `uint16` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

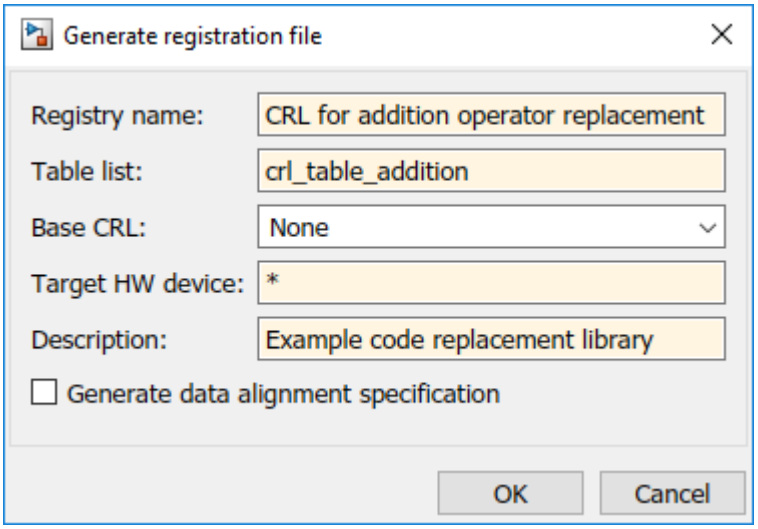
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                   'Key',                      'RTW_OP_ADD', ...
                   'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                   'ImplementationName',       'u8_add_cao_u16_u16');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table
```

```
%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_ADD', ...
                    'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                    'ImplementationName',       'u8_add_cao_u16_u16');

%% Create the conceptual representation
arg = getTflArgFromString(hTflTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTflTable, 'u1', 'uint16');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTflTable, 'u2', 'uint16');
addConceptualArg(hEntry, arg );
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTflTable = make_cbo_add_crl_table()
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_ADD', ...
                    'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                    'ImplementationName',       'u8_add_cao_u16_u16');

%% Create the conceptual representation
arg = getTflArgFromString(hTflTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTflTable, 'u1', 'uint16');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTflTable, 'u2', 'uint16');
addConceptualArg(hEntry, arg );

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
addEntry(hTflTable, hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = make_cbo_add_crl_table
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for addition subtraction operator replacement';
this(1).TableList = {'make_cbo_add_crl_table.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Generate Code using Addition and Subtraction Operator Code Replacement Library

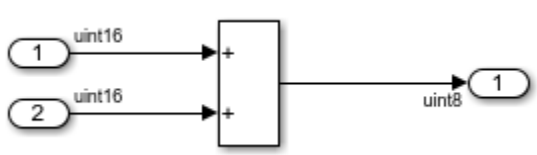
In this section, you generate code using the code replacement library that you created in the first two sections of this example. The generated code contains the implementation function signature that modifies an addition operator. For this example, the addition operator contains the `Cast before operation` parameter. This example does not provide an implementation function. You should write the implementation function that contains the optimization that you want to see in the generated code.

Example Model

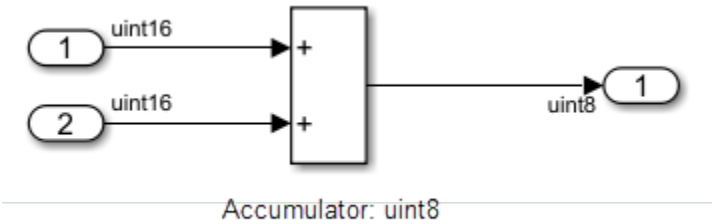
Open the model `crl_AddSubfunction` for configuring the code replacement library.

```
crl_AddSubfunction
```

```
copyfile AddSubFnRtwTargetInfo.txt rtwTargetInfo.m
```



The input ports are of data type `uint16` and the output port is of data type `uint8`. For the Add block, the **Accumulator data type** is set to `uint8`. Based on these data types, the addition is a cast-before-operation algorithm.



To create a code replacement table and register the file, run the MATLAB customization file. For this example, the customization file has already been executed. To register the library, run the sl_refresh_customizations function.

```
sl_refresh_customizations;
```

Enable Code Replacement Library

1. Open the Configuration Parameters dialog box.
2. On the **Interface** pane, set **Code Replacement Library**. Click **Select** and add `CRL for Addition and Subtraction operator replacement` to the **Selected code replacement libraries - prioritized** list pane. Alternatively, use the command-line API to enable the code replacement:

```
set_param('crl_AddSubfunction', 'CodeReplacementLibrary', 'CRL for Addition and Subtraction opera
```

Generate code from the model:

```
evalc('slbuild(''crl_AddSubfunction'')');
```

View the generated code. Here is a portion of `crl_AddSubfunction.c`.

```
cfile = fullfile('crl_AddSubfunction_ert_rtw','crl_AddSubfunction.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void crl_AddSubfunction_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<Root>/Add8'
   */
  crl_AddSubfunction_Y.Out1 = u8_add_cbo_u16_u16(crl_AddSubfunction_U.In1,
    crl_AddSubfunction_U.In2);
}
```

The + operator can be replaced with your custom implementation.

Close the model.

```
delete ./rtwTargetInfo.m
bdclose('crl_AddSubfunction');
rtwdemoclean;
```

See Also

More About

- “Addition and Subtraction Algorithm Types for Code Replacement” on page 72-150
- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Shift Left Operations and Code Replacement” on page 72-224

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

You can develop a code replacement library for floating-point matrix/matrix and matrix/vector multiplication operations with the multiplication functions `dgemm` and `dgemv` defined in the MathWorks BLAS library. If you use a third-party BLAS library for replacement, you must change the build requirements in this example to point to your library. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_blas`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Blas Operation (Fortran)**.

4. Create entry parameters. In the **Function** drop-down list, select `Multiply`.

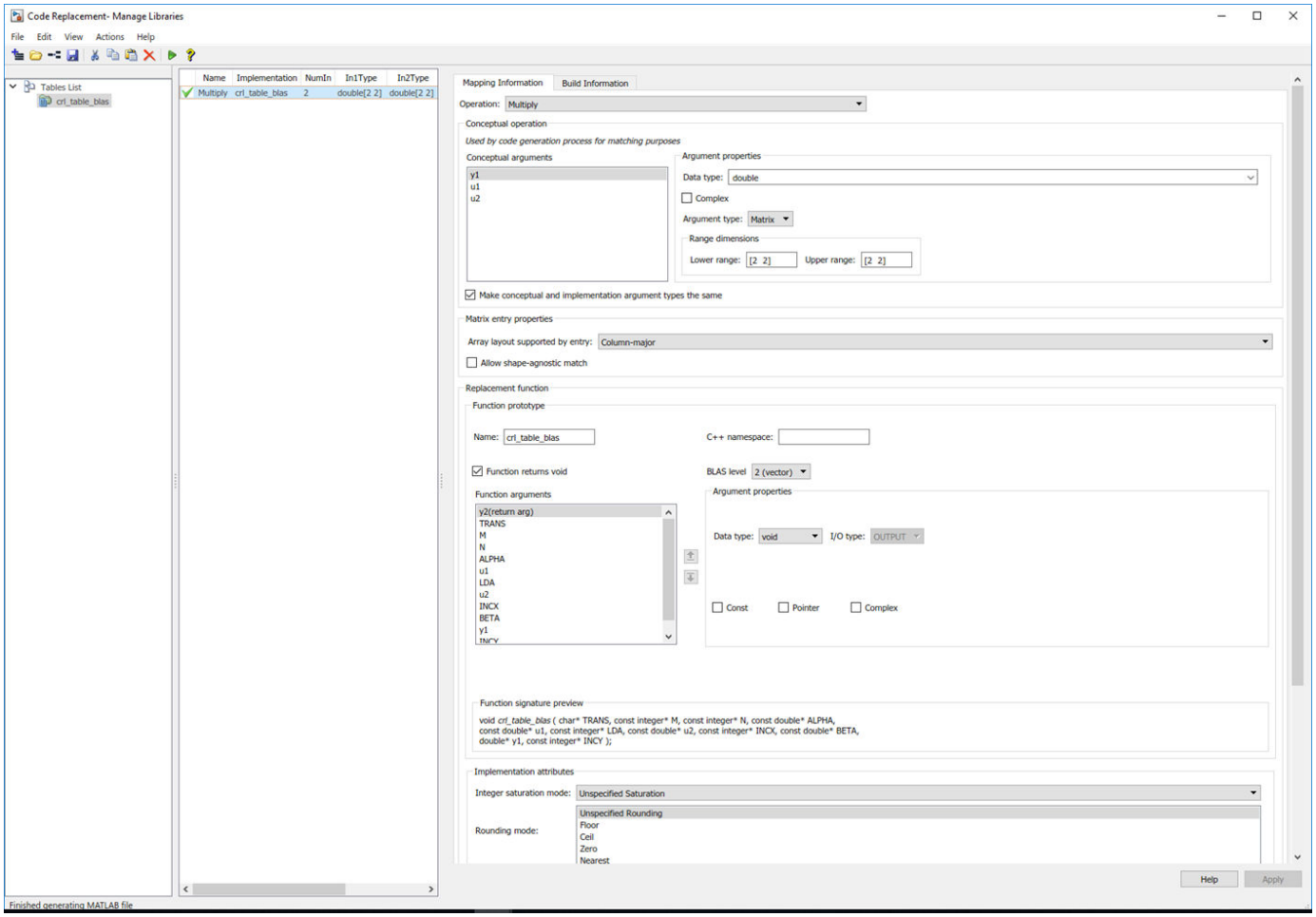
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of double and the **Argument Type** of Matrix.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Set the **BLAS level** to `2(Vector)`.

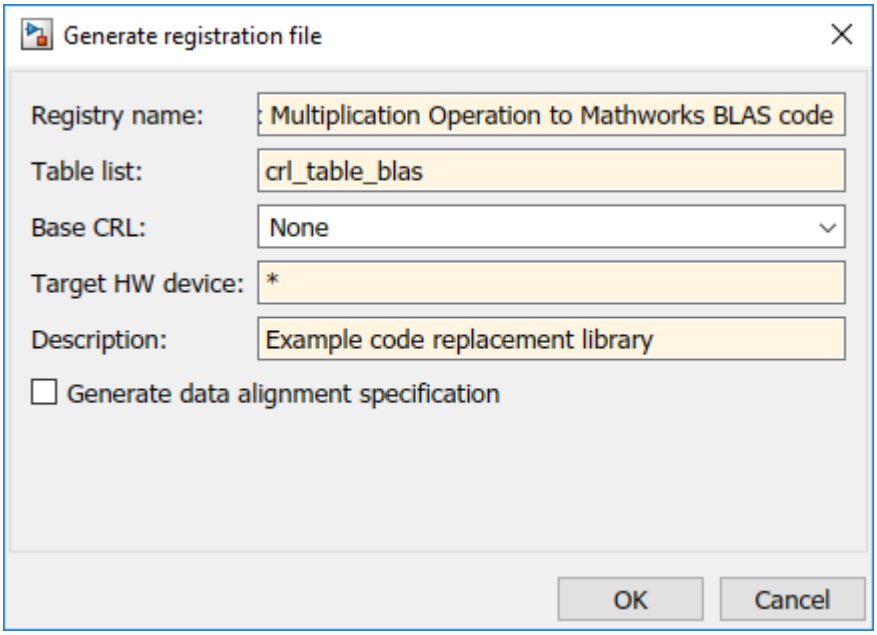
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_blas
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflBlasEntryGenerator`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
```

```
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, [2 2; inf inf] means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the dgemm32 entry for matrix/matrix multiplication replacement specifies dimensions [2 2; inf inf], while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions [2 1; inf 1].

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
```

```
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'u1', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'u2', ...
                            'BaseType',     'double', ...
                            'DimRange',     [1 1; inf inf]);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'y1', ...
                            'IOType',       'RTW_IO_OUTPUT', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
```

```
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf inf]);

%% Create the Implementation Representation
% Using RTW.TflBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANSA');
% Possible values for PassByType property are
%  RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
%  RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = RTW.TflArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_blas
   ```
9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for matrix multiplication for Mathworks Blas code';
   this(1).TableList = {'crl_table_blas.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```
10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

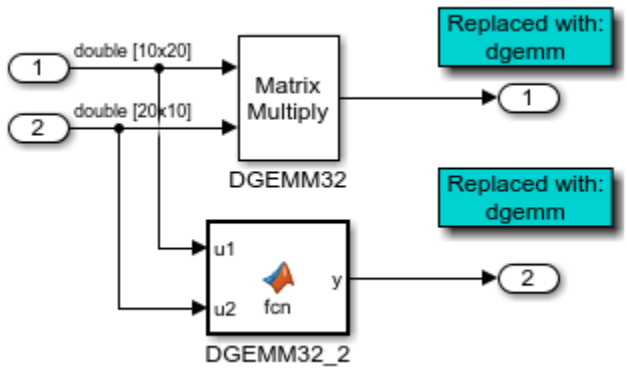
Convert a Matrix Multiplication Operation to MathWorks BLAS Through Code Replacement

In this section, you generate code using the code replacement library that you created in the first two sections of this example. A floating-point matrix/matrix multiplication operation is replaced with a custom multiplication function `dgemm` in the generated code. This example does not provide an implementation function. Write your own implementation.

Example Model

Open the model `crl_blas` for configuring the code replacement library.

```
open_system('crl_blas');

copyfile BlasTargetInfo.txt rtwTargetInfo.m
```



Run the MATLAB customization file to create a code replacement table and register the file. The customization file has already been executed. Run the sl_refresh_customizations function to register the library.

```
sl_refresh_customizations;
```

Enable Code Replacement Library

1. Open the Configuration Parameters dialog box.
2. On the **Interface** pane, set **Code Replacement Library** by clicking **Select** and adding `CRL for matrix multiplication for Mathworks Blas code` to the **Selected code replacement libraries - prioritized** list pane. Alternatively, use the command-line API to enable the code replacement:

```
set_param('crl_blas', 'CodeReplacementLibrary', 'CRL for matrix multiplication for Mathworks Blas
```

Generate code from the model:

```
evalc('slbuild(''crl_blas'')');
```

View the generated code. Here is a portion of `crl_blas.c`.

```
cfile = fullfile('crl_blas_ert_rtw','crl_blas.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void crl_blas_step(void)
{
  real_T ALPHA;
  real_T BETA;
  int32_T K;
  int32_T LDA;
  int32_T LDB;
  int32_T LDC;
  int32_T M;
  int32_T N;
  char_T TRANSA;
```

```
  char_T TRANSB;

  /* Product: '<Root>/DGEMM32' */
  TRANSA = 'N';
  TRANSB = 'N';
  M = 10;
  N = 10;
  K = 20;
  ALPHA = 1.0;
  LDA = 10;
  LDB = 20;
  BETA = 0.0;
  LDC = 10;

  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/DGEMM32'
   */
  dgemm32(&TRANSA, &TRANSB, &M, &N, &K, &ALPHA, &rtU.In1[0], &LDA, &rtU.In2[0],
          &LDB, &BETA, &rtY.Out1[0], &LDC);

  /* MATLAB Function: '<Root>/DGEMM32_2' */
  /* MATLAB Function 'DGEMM32_2': '<S1>:1' */
  /* '<S1>:1:5' */
  TRANSA = 'N';
  TRANSB = 'N';

  /* Outport: '<Root>/Out7' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  MATLAB Function: '<Root>/DGEMM32_2'
   */
  dgemm32(&TRANSA, &TRANSB, &M, &N, &K, &ALPHA, &rtU.In1[0], &LDA, &rtU.In2[0],
          &LDB, &BETA, &rtY.Out7[0], &LDC);
}
```

The matrix/matrix multiplication operation is repalced with the custom `dgemm32` function.

Close the model and code generation report.

```
delete ./rtwTargetInfo.m
bdclose('crl_blas');
rtwdemoclean;
```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 72-170

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

You can develop a code replacement library for floating-point matrix/matrix and matrix/vector multiplication operations with the multiplication functions `sgemm` defined in the MathWorks C BLAS library. Use a third-party C BLAS library for replacement and change the build requirements in this example to point to your library. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_cblas`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **CBlas Operation (Atlas)**.

4. Create entry parameters. In the **Function** drop-down list, select `Multiply`.

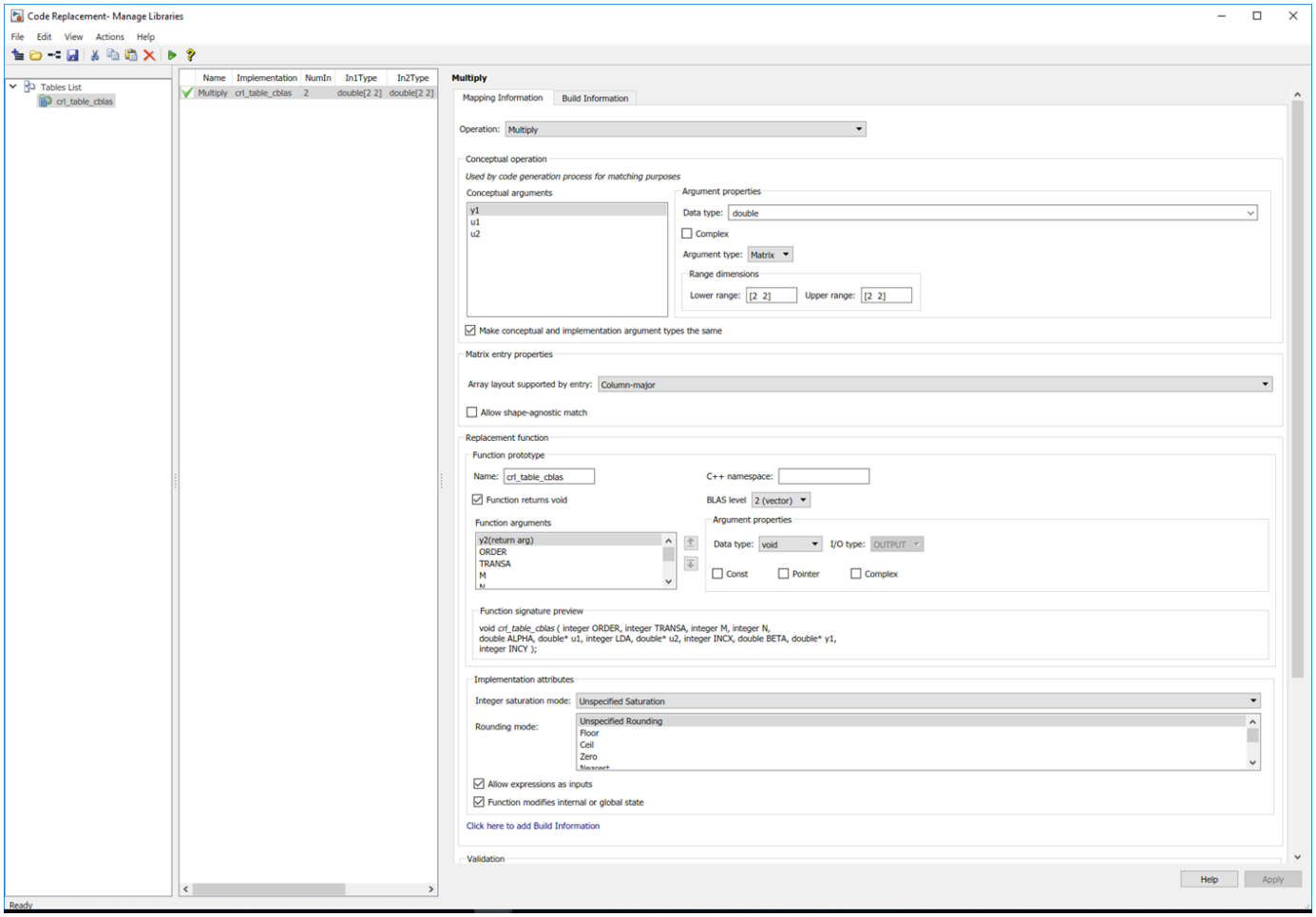
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of single and the **Argument Type** of Matrix.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Set the **BLAS level** to `3(Vector)`.

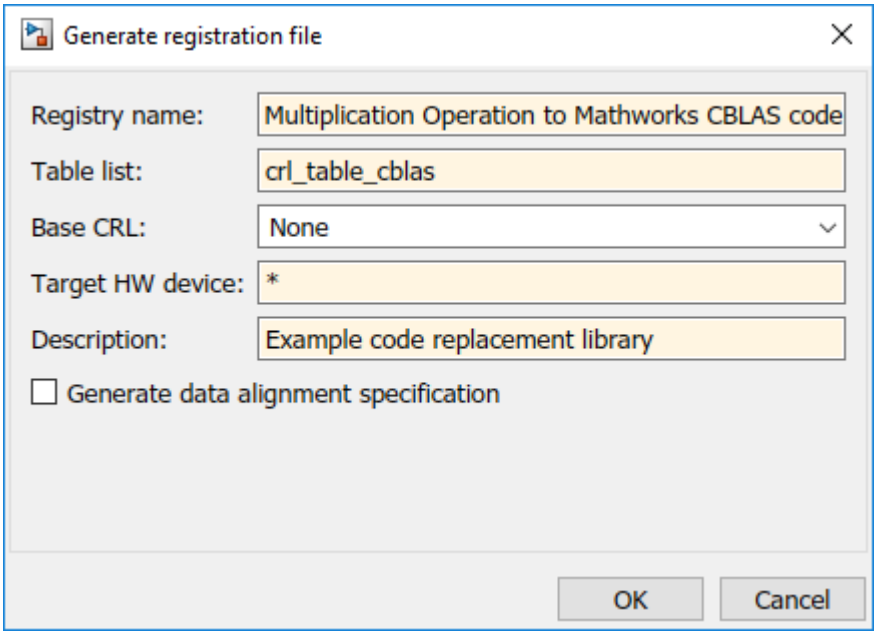
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_cblas
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCBlasEntryGenerator`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
```

```
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',        {'-lgfortran'},...
    'SideEffects', true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, [2 2; inf inf] means a two-dimensional matrix of size 2x2 or larger.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
```

```
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',      {'-lgfortran'},...
    'SideEffects', true);

%% Create the conceptual representation
arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'single');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'single');
arg.DimRange = [2 1; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'single');
arg.DimRange = [1 2; Inf Inf];
hEnt.addConceptualArg(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',      {'-lgfortran'},...
    'SideEffects', true);

%% Create the conceptual representation
```

```
arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'single');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'single');
arg.DimRange = [2 1; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'single');
arg.DimRange = [1 2; Inf Inf];
hEnt.addConceptualArg(arg);

%% Create the Implementation Representation
% Using RTW.TflCBlasEntryGenerator for sgemm requires the following
% implementation signature:
%
% void no_name( const integer ORDER, const integer TRANSA, const integer TRANSB,
% const integer M, const integer N, const integer K, const double ALPHA,
% const double* u1, const integer LDA, const double* u2, const integer LDB,
% const double BETA, double* y1, const integer LDC );
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature
arg = hEntry.getTflArgFromString('y2','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = hEntry.getTflArgFromString('ORDER','integer',102.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('TRANSA','integer',111.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('TRANSB','integer',111.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('M','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('N','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('K','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('ALPHA','double',1.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u1','single*');
arg.Type.BaseType.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('LDA','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u2','single*');
arg.Type.BaseType.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('LDB','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('BETA','double',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('y1','single*');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.addArgument(arg);
```

```
arg = hEntry.getTflArgFromString('LDC','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_cblas
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for matrix multiplication for Mathworks CBlas code';
this(1).TableList = {'crl_table_cblas.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 72-160

Small Matrix Operation to Processor Code Replacement

This example shows how to develop a code replacement library to optimize the performance of non-scalar small matrix operations with processor-specific intrinsic functions by providing information on how to define code replacement for a `addition` operator. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_small_matrix`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Function** drop-down list, select `Addition`. Algorithm information appears in the crtool. For this example, select the parameter to `Cast before operation`.

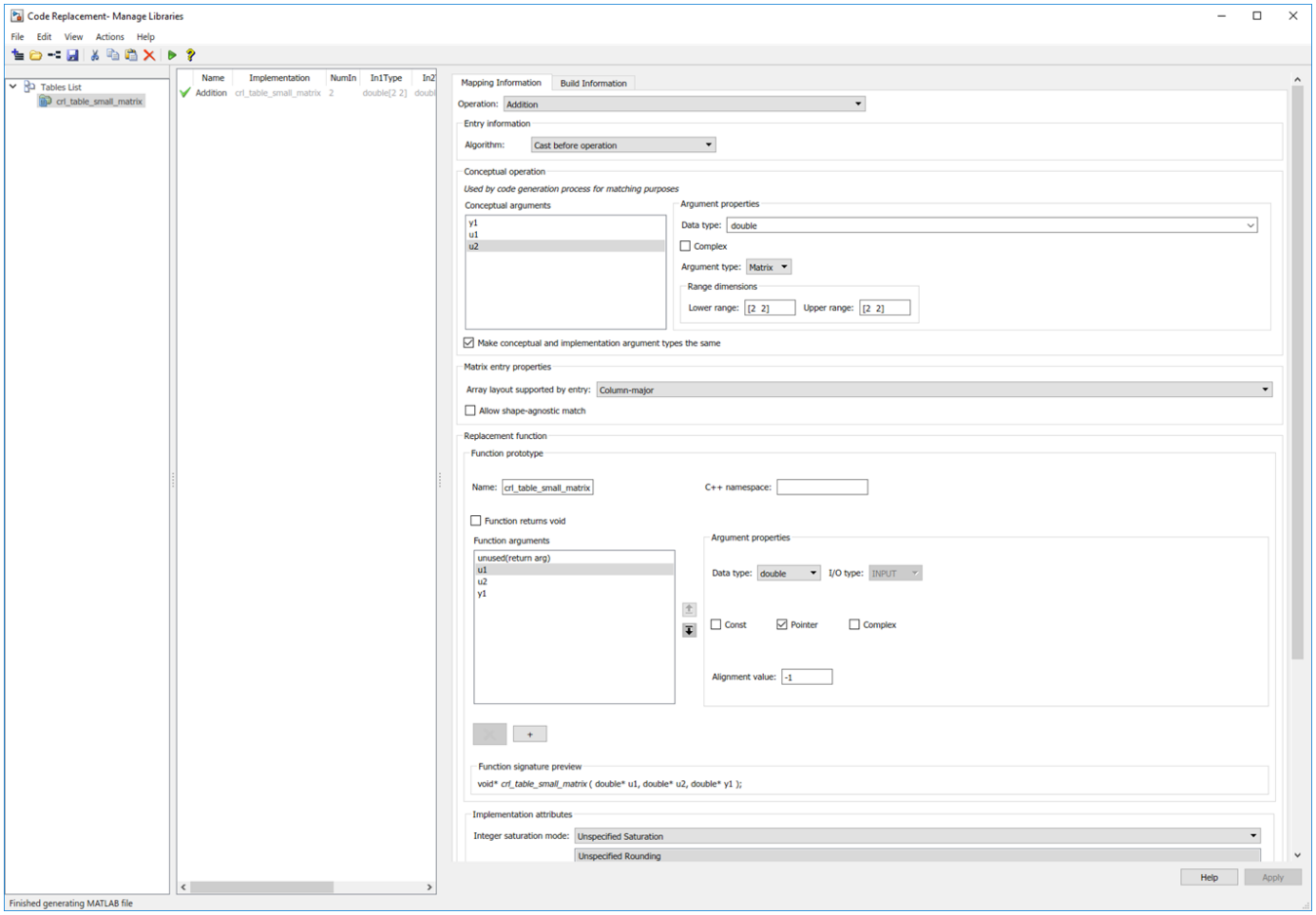
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, with the **Data Type** of `double` and the **Argument Type** of `Matrix`, the input argument, `u1` with the **Data Type** of `double` and the **Argument Type** of `Matrix` and the input argument, `u2` with the **Data Type** of `double` and the **Argument Type** of `Matrix`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Specify the return argument `y1, u1` and `u2`, with the **Data Type** of `double` and select the **Pointer** dialog box.

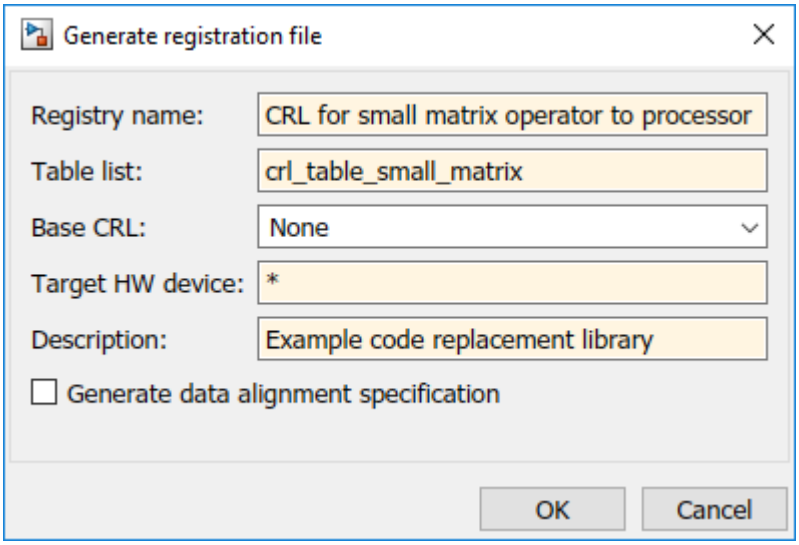
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
```

```
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);


%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

```
%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);


%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);


%% Create the Implementation Representation
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);

arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_small_matrix
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for small matrix operator to processor';
this(1).TableList = {'crl_table_small_matrix.m'};
% table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Generate Code to Optimize Performance of a Small Matrix Operation for Processor-Specific Intrinsic Function

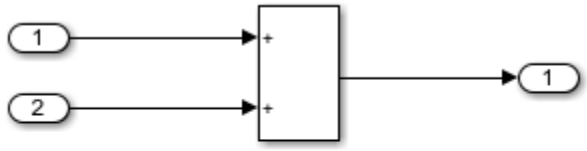
In this section, you generate code using the code replacement library that you created in the first two sections of this example. A nonscalar small matrix multiplication operation is replaced with a call to a custom function in the generated code. This example does not provide an implementation of the custom function. Write your own implementation.

Example Model

Open the model `crl_SmallMatrixOperation` for configuring the code replacement library.

```
open_system('crl_SmallMatrixOperation');
```

```
copyfile SmallMatrixOperatorTargetInfo.txt rtwTargetInfo.m
```



Copyright 2007-2020 The MathWorks, Inc.

Run the MATLAB® customization file to create a code replacement table and register the file. The customization file has already been executed. Run the sl_refresh_customizations function to register the library.

```
sl_refresh_customizations;
```

Enable Code Replacement Library

1. Open the Configuration Parameters dialog box.
2. On the **Interface** pane, set **Code Replacement Library** by clicking **Select** and adding `CRL for small matrix operator` to the **Selected code replacement libraries - prioritized** list pane. Alternatively, use the command-line API to enable the code replacement:

```
set_param('crl_SmallMatrixOperation', 'CodeReplacementLibrary', 'CRL for small matrix operator')
```

Generate code from the model:

```
evalc('slbuild(''crl_SmallMatrixOperation'')');
```

View the generated code. Here is a portion of `crl_SmallMatrixOperation.c`.

```
cfile = fullfile('crl_SmallMatrixOperation_ert_rtw','crl_SmallMatrixOperation.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void crl_SmallMatrixOperation_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<Root>/Add'
   */
  matrix_sum_2x2_double(&crl_SmallMatrixOperation_U.In1[0],
                        &crl_SmallMatrixOperation_U.In2[0],
                        &crl_SmallMatrixOperation_Y.Out1[0]);
}
```

The matrix multiplication operation on `2X2` matrices are replaced with the `matrix_sum_2x2_double` function.

Close the model and code generation report.

```
delete ./rtwTargetInfo.m
bdclose('crl_SmallMatrixOperation');
rtwdemoclean;
```

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

- “Scalar Operator Code Replacement” on page 72-145

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
                       'SideEffects',              true );
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

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

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

   arg = getTflArgFromString(hTable, 'u2', 'uint8');
   addConceptualArg(op_entry, arg );
   ```

6. Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the

arguments. When defining the implementation function return argument, create a new `void` output argument, for example, `y2`. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Create new void output y2
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);
```

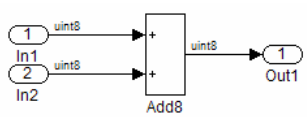
7. Add the entry to a code replacement table with a call to the `addEntry` function.

   ```
   addEntry(hTable, op_entry);
   ```

8. Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1. Register the code replacement mapping.
2. Create a model that includes an Add block.



3. Configure the model with the following settings:
   - On the **Solver** pane, select a fixed-step solver.
   - On the **Code Generation** pane, select an ERT-based system target file.
   - On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.
   - Set the **Optimize global data access** parameter to `Use global to hold temporary results` to reduce data copies in the generated code.
4. Generate code and a code generation report.
5. Review the code replacements.

See Also

More About

- "Code You Can Replace From Simulink Models" on page 72-6
- "Define Code Replacement Library Optimizations" on page 72-36

- “Develop a Code Replacement Library” on page 72-20

Fixed-Point Operator Code Replacement

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

Common Ways to Match Fixed-Point Operator Entries

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point binary-point-only scaling. • `FractionLength`: Specify a fraction length (for example, 3).
A specific slope bias scaling combination on the operator inputs and output.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/-FixedExponent` equivalent): Specify a slope value (for example, 15). • `Bias`: Specify a bias value (for example, 2).

Match	Create entry	Minimally specify parameters
Net slope between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntry-Generator_NetSlope`	`setTflCOperationEntryParameters` function: • `NetSlopeAdjustmentFactor`: Specify the slope adjustment factor (`F`) part of the net slope, $F2^E$ (for example, `1.0`). • `NetFixedExponent`: Specify the fixed exponent (`E`) part of the net slope, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.
Relative scaling between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `RelativeScalingFactorF`: Specify the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$ (for example, `1.0`). • `RelativeScalingFactorE`: Specify the fixed exponent (`E`) part of the relative scaling factor, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.
Equal slope and zero net bias across operator inputs and output (addition and subtraction).	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes $\tilde{V}$, referred to as the quantized integer.
- S is a coefficient of Q, referred to as the slope.
- B is an additive correction, referred to as the bias.

The general equation for an operation between fixed-point operands is:

$$(S_O Q_O + B_O) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

The objective of fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 + \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. To match for replacement, the slopes must be the same for all addition conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 - \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters

`SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. To match for replacement, the slopes must be the same for all subtraction conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few known slope and bias combinations. Use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0Q_0 = (S_1Q_1)(S_2Q_2)$$

$$Q_0 = \left(\frac{S_1S_2}{S_0}\right)Q_1Q_2$$

$$Q_0 = S_nQ_1Q_2$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0. Also, set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all multiplication conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few known slope and bias combinations. Use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry (see “Customize Match and Replacement Process” on page 72-132).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0Q_0 = \left(\frac{S_1Q_1}{S_2Q_2}\right)$$

$$Q_0 = S_n\left(\frac{Q_1}{Q_2}\right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. For the `s16_netslope0p5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0. Also, set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all division conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope. Set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all cast conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n}\right)$$

$$Q_0 = \left(\frac{S_1}{S_0}\right)\left(\frac{Q_1}{2^n}\right)$$

$$Q_0 = S_n\left(\frac{Q_1}{2^n}\right)$$

where S_n is the net slope. Set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all shift conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Define Code Replacement Library Optimizations” on page 72-36
- “Binary-Point-Only Scaling Code Replacement” on page 72-194
- “Slope Bias Scaling Code Replacement” on page 72-199
- “Net Slope Scaling Code Replacement” on page 72-205
- “Equal Slope and Zero Net Bias Code Replacement” on page 72-212
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 72-218
- “Shift Left Operations and Code Replacement” on page 72-224
- “Data Alignment for Code Replacement” on page 72-105
- “Remap Operator Output to Function Input” on page 72-185
- “Customize Match and Replacement Process” on page 72-132
- “Develop a Code Replacement Library” on page 72-20

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination of operator inputs and output. This example shows how to develop a code replacement library to optimize the performance of fixed point data type operations by providing information that maps a binary point scale entry to a `addition` operator. To develop a code replacement library use only the programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process see “Customize Code Match and Replacement for Scalar Operations” on page 72-139. You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntryBinaryPoint < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = 0;
        end

        % Set the fraction-length values in the implementation function.
        ent.Implementation.Arguments(3).Value = ...
```

```
            -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
         ent.Implementation.Arguments(4).Value = ...
            -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
         ent.Implementation.Arguments(5).Value = ...
            -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
      end
    end
  end
end
```

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_fixed_binptscale()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the custom entry function .

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `createAndSetCImplementationReturn` and `createAndAddImplementationArg`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
```

```
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);

%% Create the implementation Representation
% Specify replacement function signature
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_out', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);
```

```
%% Add the entry to the table
hTable.addEntry(op_entry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_fixed_binptscale
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for binary point scaling CRL’;
   this(1).TableList = {'crl_table_fixed_binptscale.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Fixed-Point Operator Code Replacement” on page 72-188
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. This example shows how to develop a code replacement library to optimize the performance of fixed point data type operations by providing information on how to define code replacement for a `division` operator. To develop a code replacement library use only the programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process, see “Customize Code Match and Replacement for Scalar Operations” on page 72-139. You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntrySlopeBias < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = hCSO.ConceptualArgs(idx).Type.Bias;
        end

        % Set the Slope and Bias values in the implementation function.
        ent.Implementation.Arguments(3).Value = hCSO.ConceptualArgs(2).Type.Slope;
        ent.Implementation.Arguments(4).Value = hCSO.ConceptualArgs(2).Type.Bias;
        ent.Implementation.Arguments(5).Value = hCSO.ConceptualArgs(3).Type.Slope;
        ent.Implementation.Arguments(6).Value = hCSO.ConceptualArgs(3).Type.Bias;
```

```
        ent.Implementation.Arguments(7).Value = hCSO.ConceptualArgs(1).Type.Slope;
        ent.Implementation.Arguments(8).Value = hCSO.ConceptualArgs(1).Type.Bias;
      end
    end
  end
end
```

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the custom entry function .

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'SlopeBias', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'SlopeBias', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'SlopeBias', ...
       'IsSigned',   false, ...
       'WordLength', 32);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function

`createAndSetCImplementationReturn` and `createAndAddImplementationArg`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'SlopeBias', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'SlopeBias', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'SlopeBias', ...
       'IsSigned',   false, ...
       'WordLength', 32);

%% Create the implementation Representation
% Specify replacement function signature
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
```

```
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 6 args for slopes and biases.
% Actual values are set during code generation.
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_out1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_out1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_custom_add_ufix32_slopebias
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for slope bias scaling function replacement';
this(1).TableList = {'crl_table_custom_add_ufix32_slopebias.m'};
% table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Fixed-Point Operator Code Replacement” on page 72-188
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Net Slope Scaling Code Replacement

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example shows how to develop a code replacement library to optimize the performance of fixed point net slope operations by providing information on how to define code replacement for a `divide` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_netslopeScaling`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Divide`.

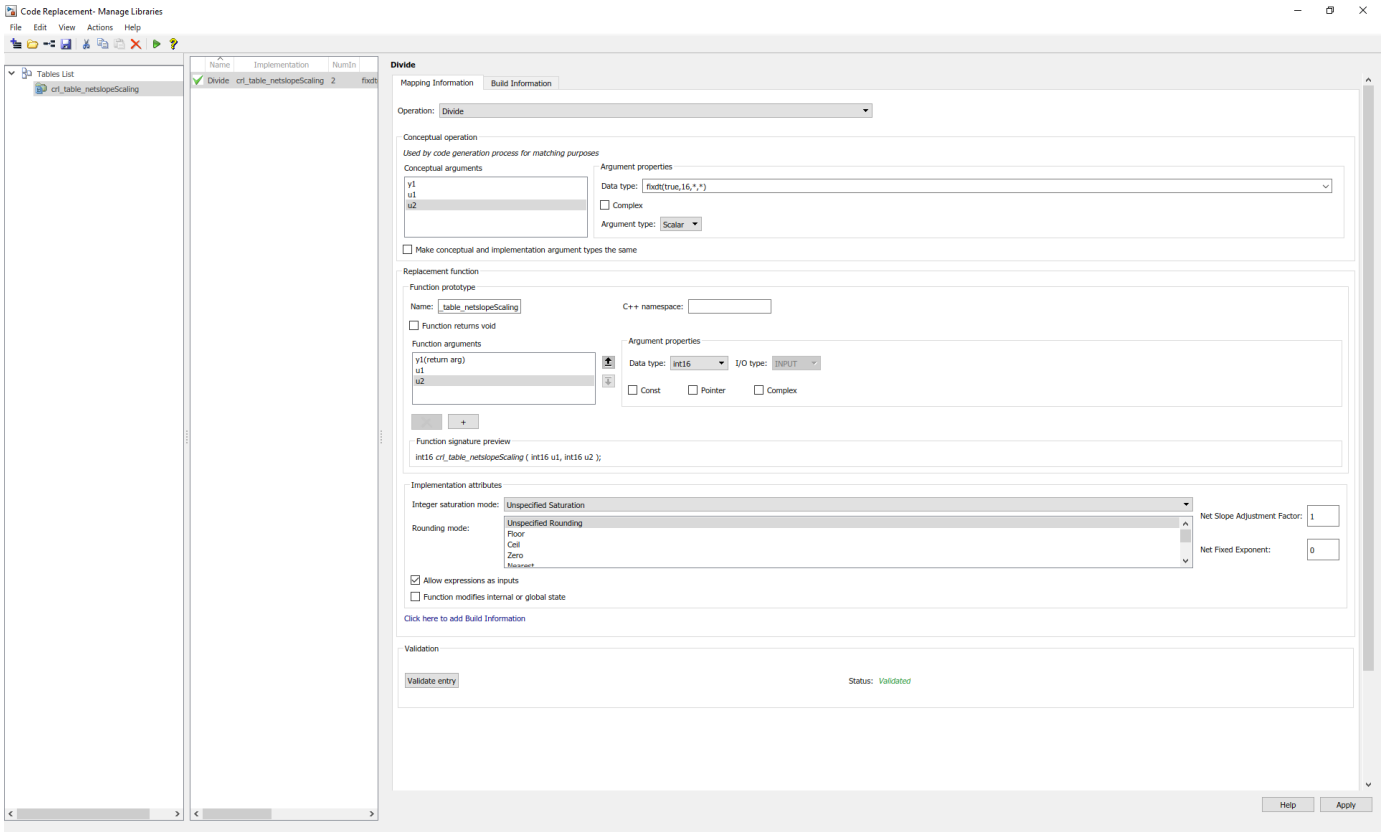
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `fixdt(true,16,*,*)` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `int 16`. Also set the **Net slope Adjustment Factor** as 1 and **Net Fixed Exponent** as 0.

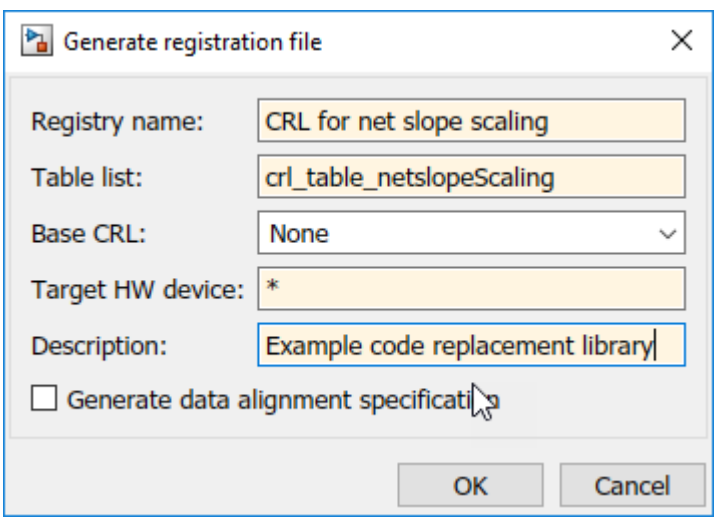
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For

more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntryGenerator_NetSlope`.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
```

```
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'y1', ...
   'IOType',         'RTW_IO_OUTPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'y1', ...
   'IOType',         'RTW_IO_OUTPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

%% Create the Implementation Representation
```

```
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
                                  'Name',           'y1', ...
                                  'IOType',         'RTW_IO_OUTPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
                                  'Name',           'u1', ...
                                  'IOType',         'RTW_IO_INPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
                                  'Name',           'u2', ...
                                  'IOType',         'RTW_IO_INPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_netslopeScaling
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for net slope scaling’;
   this(1).TableList = {'crl_table_netslopeScaling.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Limitations

- For the conceptual arguments of a net slope operation replacement entry, you must set these parameters:
  - `CheckSlope` — `false`
  - `CheckBias` — `false`
- For net slope operation replacement entries, you must set `MustHaveZeroNetBias` to `true`.
- You must set `SlopesMustBeTheSame` depending on the operation type:
  - Add/Minus — `true`
  - Cast/Shift — `true`
  - Mul/Div/MulDiv — `false`
- For non-trivial net slope or bias cases, use `RTW.TflCOperationEntryML` instead.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Fixed-Point Operator Code Replacement” on page 72-188
- “Equal Slope and Zero Net Bias Code Replacement” on page 72-212
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match the relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values when mapping relative slope and bias values to a replacement function for addition or subtraction.

This example shows how to develop a code replacement library to optimize the performance of fixed point net slope operations by providing information on how to define code replacement for an `addition` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_equal_slope_netbias_add`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Addition`.

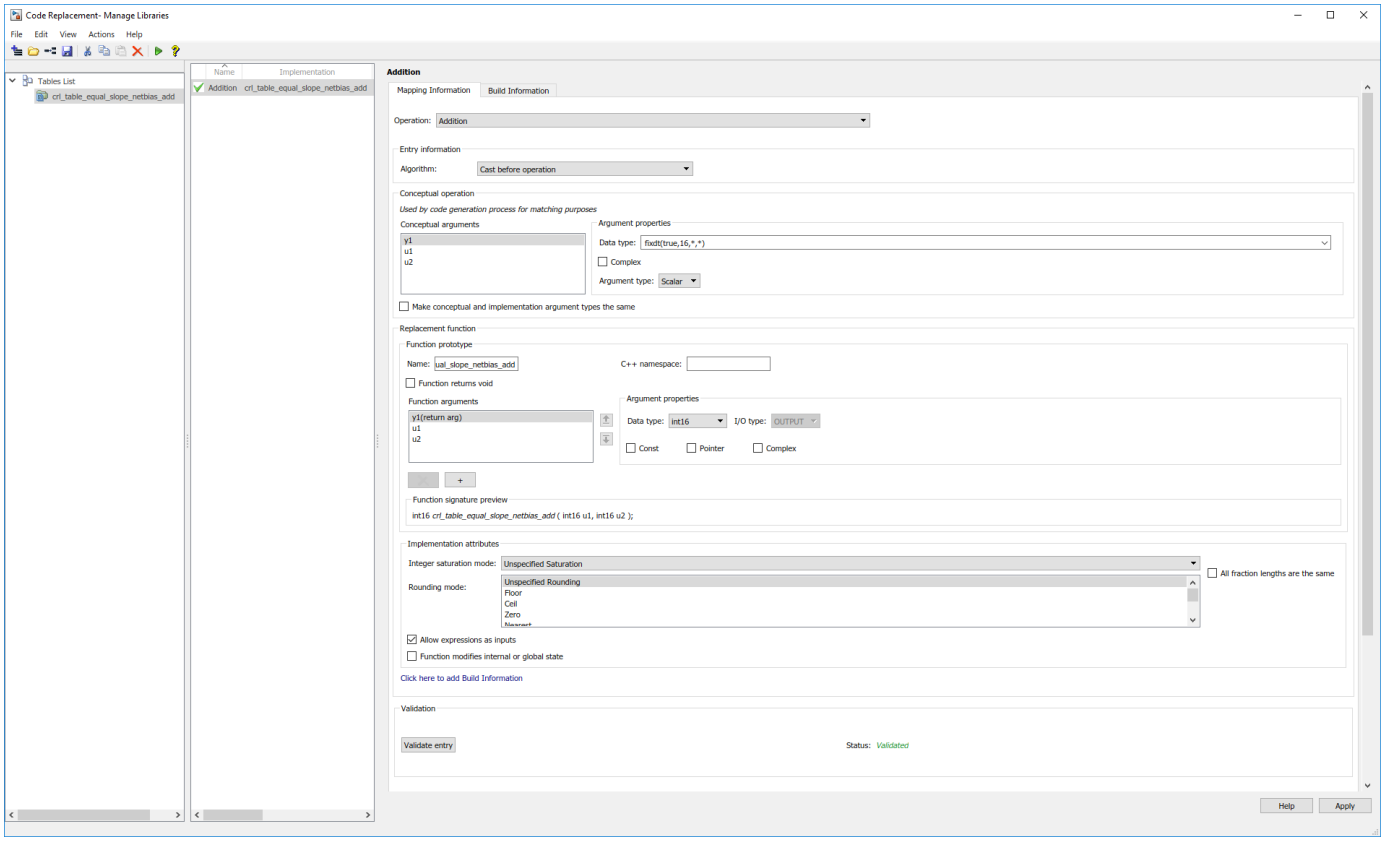
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `fixdt(true,16,*,*)` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `int 16`.

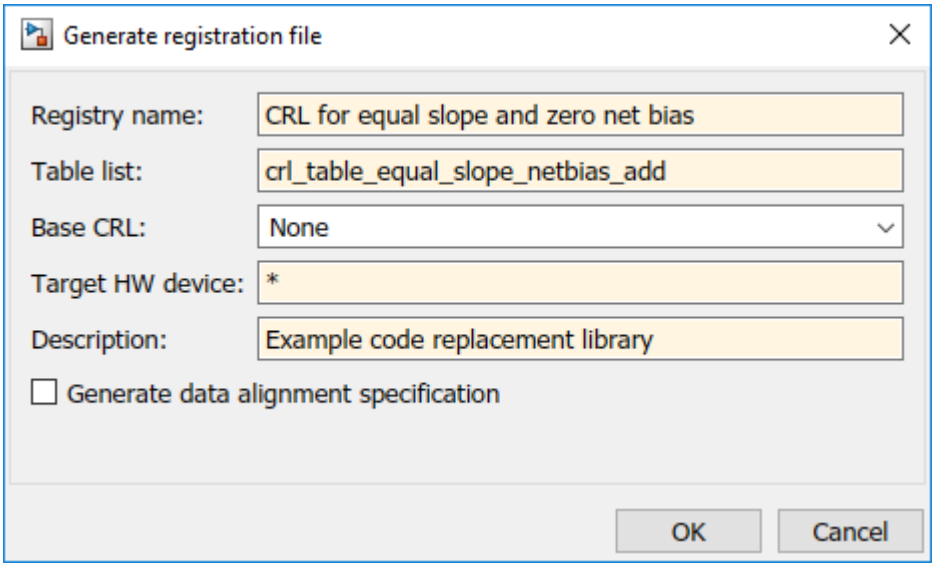
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntryGenerator_NetSlope`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. This example uses calls to the `createAndSetCImplementationReturn` function to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
```

```
    'ImplementationName',       'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);


%% Create the Implementation Representation
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_equal_slope_netbias_add
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for equal slope and zero net bias code replacement’;
   ```

```
this(1).TableList = {'crl_table_equal_slope_netbias_add.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Fixed-Point Operator Code Replacement” on page 72-188
- “Net Slope Scaling Code Replacement” on page 72-205
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Data Type Conversions (Casts) and Operator Code Replacement

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), at the MATLAB command line, enter:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_cast_int32_to_int16`. Click **Apply**.

3. Create entry parameters. In the **Operation** drop-down list, select `Cast`.

4. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

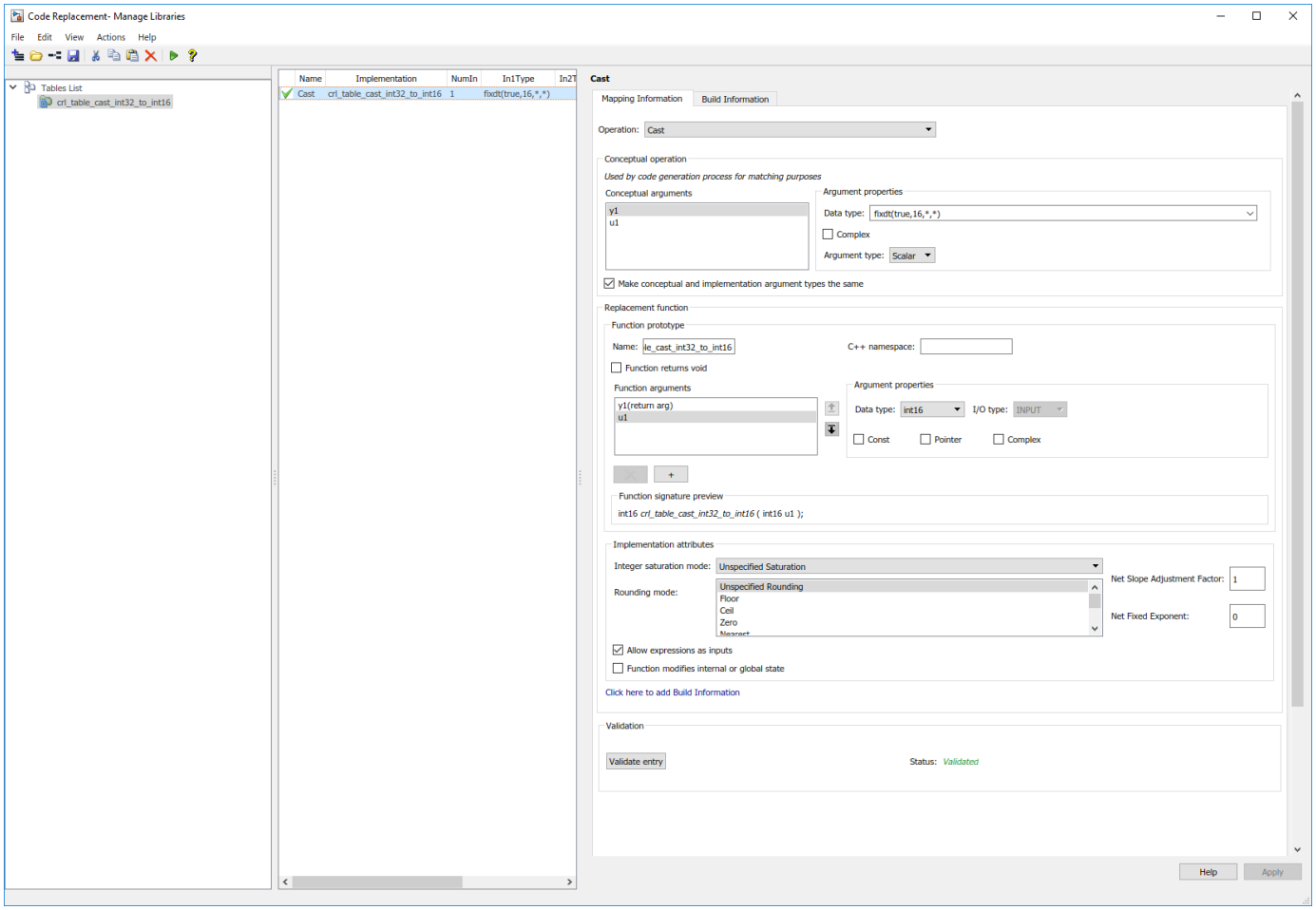
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1` with the **Data Type** of `int16` , and the input argument, `u1` with the **Data Type** of `int32` and the **Argument Type** of `Scalar`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` with the **Data Type** of `int16` and `int32`.

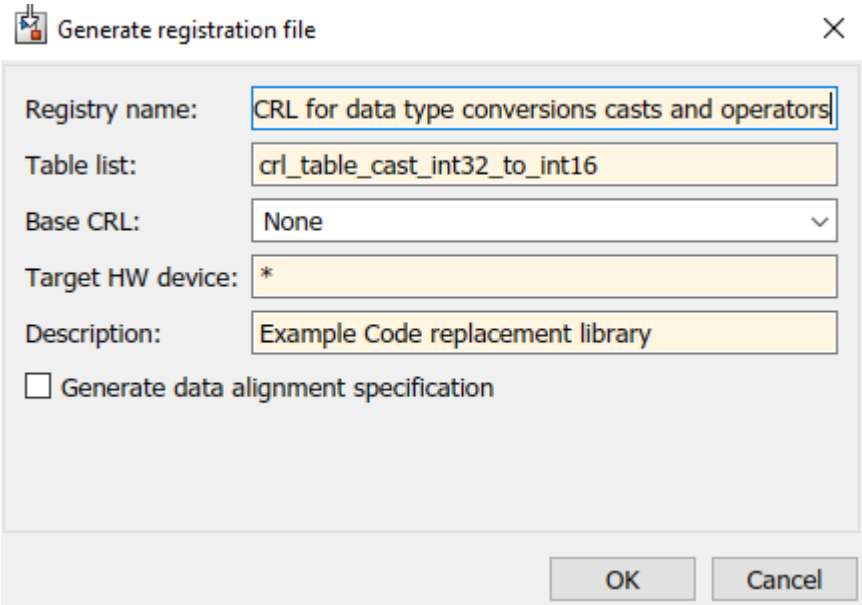
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

```
%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.The parameters specify the type of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual representation

arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the

same order and properties as the conceptual arguments, call the function `getTflArgFromString`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual and implementation representation

arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable,hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_cast_int32_to_int16
```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for data type conversions(casts) abd operator code replacement’;
this(1).TableList = {'crl_table_cast_int32_to_int16.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Fixed-Point Operator Code Replacement” on page 72-188
- “Net Slope Scaling Code Replacement” on page 72-205
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Shift Left Operations and Code Replacement

This example shows how to develop a code replacement library to optimize the performance of shift left (<<) operation by providing information on how to define code replacement for a `shift left` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 72-20.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_shift_left_int16`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Operator** drop-down list, select `Shift Left`.

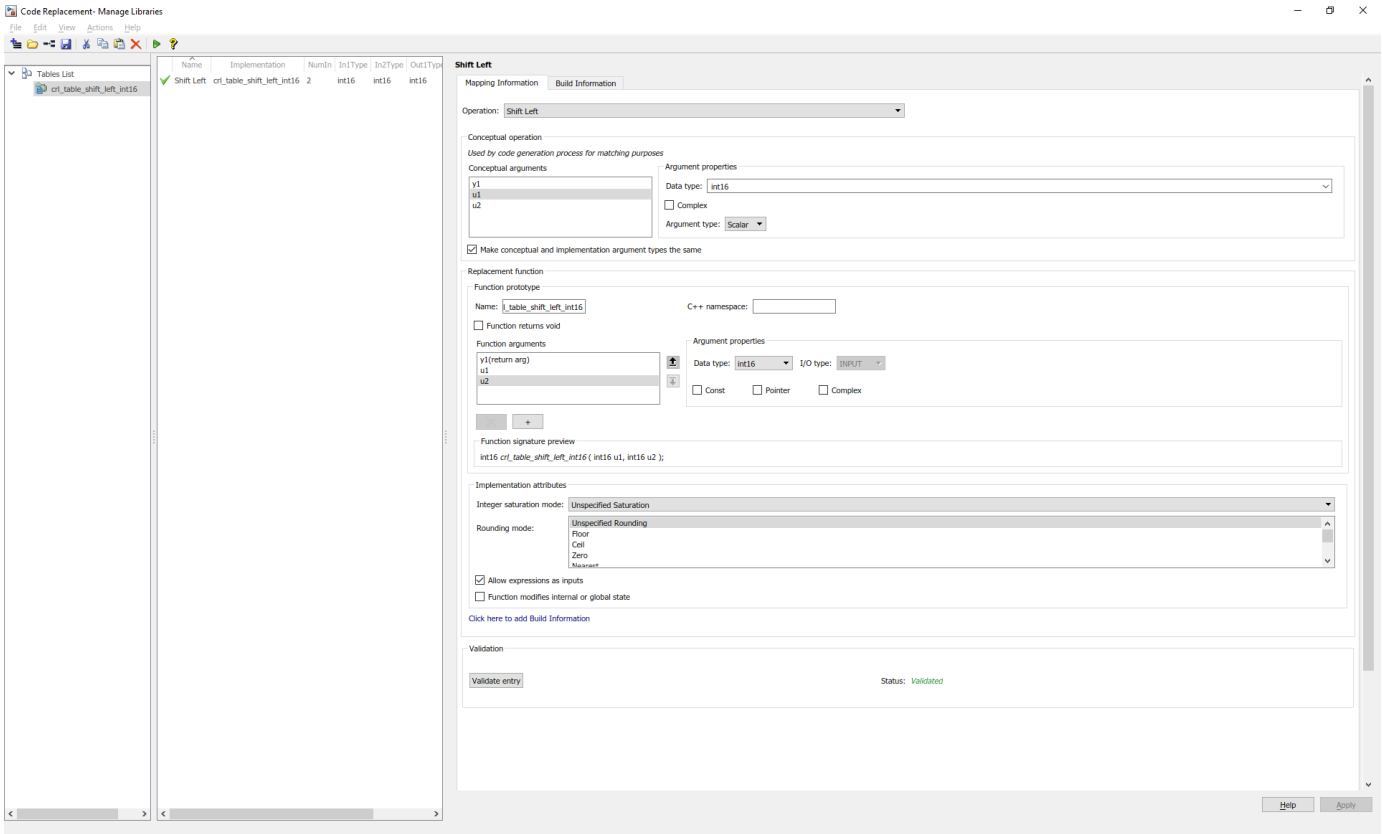
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1`, and `u2`, with **Data Type** of `int16` and the **Argument Type** of `scalar`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

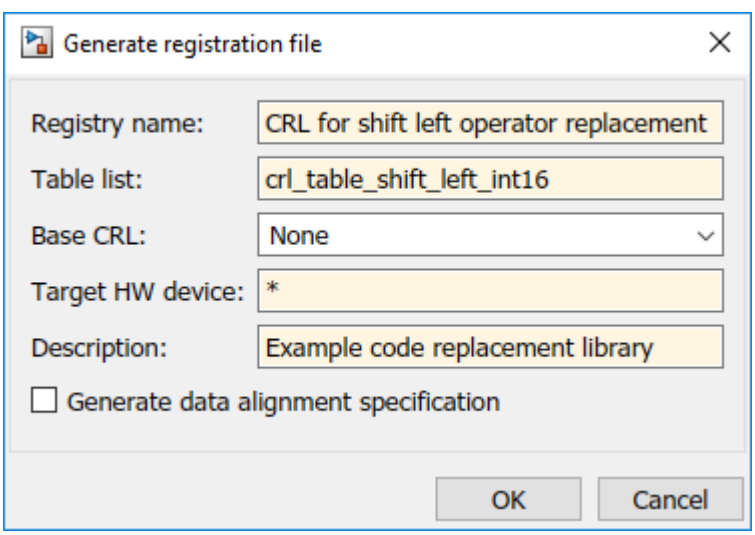
   Under **Function prototype**specify a **Name** for the replacement function.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_shift_left_int16
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this example replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString` and `addConceptualArg` .

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', 'int16');
arg.CheckType = false;
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual and implementation representation
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', 'int16');
arg.CheckType = false;
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_shift_left_int16
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) by using these specifications.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for shift left operation’;
this(1).TableList = {'crl_table_shift_left_int16.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Replace Addition and Subtraction Operator Code” on page 72-153

Optimize Generated Code by Using Multiple Code Replacement Libraries

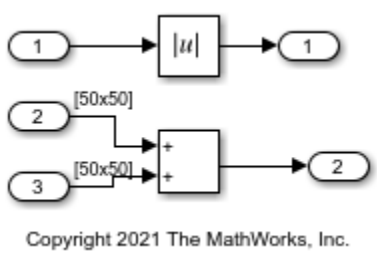
You can generate code that has code replacements enabled by multiple code replacement libraries. You can select multiple Code Replacement libraries in a unified workflow so that the generated code contains optimizations from varied code replacement libraries such as `AUTOSAR 4.0` and `GNU C99 extensions`. You can also select your own custom code replacement libraries along with libraries included with Embedded Coder® to further optimize the generated code. You can select and use a combination of libraries and achieve the requisite functionalities, and need not author separate libraries. In instances where there are multiple custom code replacement libraries and you need functionalities from all of them, individually select the libraries.

Example Model

This example shows how to replace the generated code for a model containing an Add block and an Abs block by using multiple code replacement libraries. You replace the Abs block using a code replacement customization that replaces the regular `fabs` function with the custom implementation `CRL_abs_d` from the `ScalarFcn` library. You replace the Add block with the custom implementation `CRL_d_add_d_d_cao` from the `ScalarOp` library.

Open the model `mMultiLibrary_abs` that has an Add block and an Abs block.

```
model = 'mMultiLibrary_abs';
open_system(model);

copyfile multipleFnRtwTargetInfo.txt rtwTargetInfo.m
```



The MATLAB® customization files `hcrl_make_abs` and `hcrl_sum_float` have already been executed. Run the sl_refresh_customizations function to register the libraries `ScalarFcn` and `ScalarOp`.

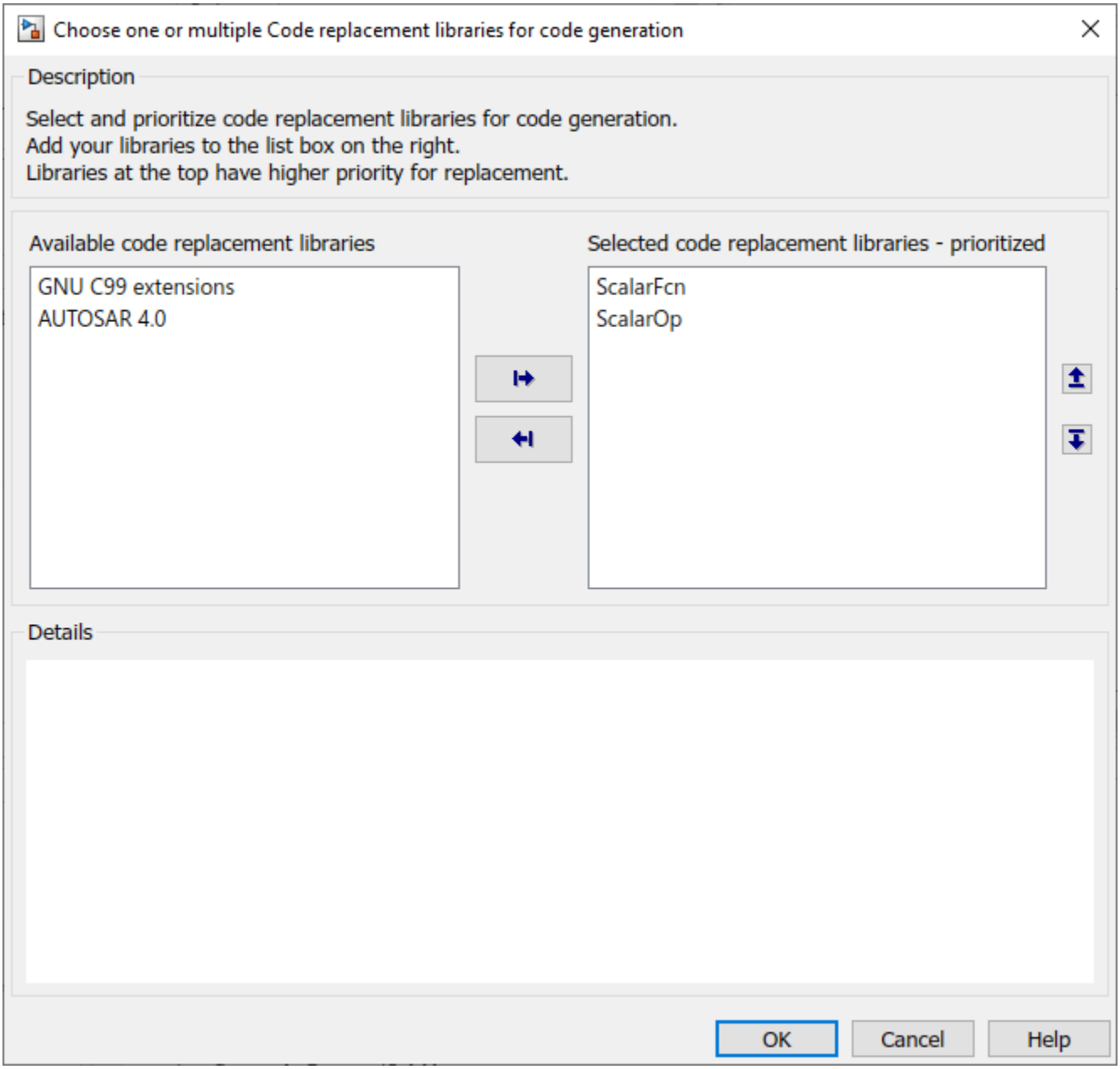
```
sl_refresh_customizations;
```

Select Code Replacement Libraries

1. Open the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, set the correct **Device Vendor** and **Device Type** depending on the hardware target. In this example `Intel` is the **Device Vendor** and `x86-64(Windows64)` is the **Device Type**.
3. On the **Interface** pane, set **Code Replacement Library** parameter by clicking **Select** and adding the code replacement libraries to the **Selected code replacement libraries -**

prioritized list pane. For this example, choose ScalarFcn and ScalarOp. The libraries work on the Windows® platforms. Linux® users must supply a different library.

4 Order the code replacement libraries according to the priority in which you want to see them replaced in the generated code. Code replacement libraries at the top of order have a higher priority during code replacement.



Alternatively, use the command-line API to enable the code replacement:

```
set_param('mMultiLibrary_abs','CodeReplacementLibrary','ScalarFcn,ScalarOp')
```

Generate code for the model.

```
evalc('slbuild(model)');
```

NOTE: When registering a code replacement library, you cannot use a comma in the code replacement library name.

View the generated code with the replacements. Here is a portion of `mMultiLibrary_abs.c`.

```
cfile = fullfile('mMultiLibrary_abs_ert_rtw','mMultiLibrary_abs.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void mMultiLibrary_abs_step(void)
{
  int32_T i;

  /* Outport: '<Root>/Out1' incorporates:
   *  Abs: '<Root>/Abs1'
   *  Inport: '<Root>/In1'
   */
  mMultiLibrary_abs_Y.Out1 = CRL_abs_d(mMultiLibrary_abs_U.In1);

  /* Outport: '<Root>/Output' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   *  Sum: '<Root>/Add'
   */
  for (i = 0; i < 2500; i++) {
    mMultiLibrary_abs_Y.Output[i] = CRL_d_add_d_d_cao(mMultiLibrary_abs_U.In2[i],
      mMultiLibrary_abs_U.In3[i]);
  }

  /* End of Outport: '<Root>/Output' */
}
```

The generated code contains the `CRL_abs_d` function enabled by the custom code replacement library `ScalarFcn`. Note that this library only works for Windows. The code also contains the function `CRL_d_add_d_d_cao` enabled by the `ScalarOp` code replacement library. Selecting multiple code replacement libraries enables multiple optimizations in the generated code.

Close the model and code generation report.

```
delete ./rtwTargetInfo.m
bdclose(model)
rtwdemoclean;
```

Limitations:

- When you use different argument types such as matrix and scalar, the priority ordering is ignored. A matrix code replacement library has a higher priority than a scalar code replacement library, regardless of the order specified by the code replacement library.

- When the code replacement library parameter specifies a Standard Math Library, for instance through its BaseTfl, and it does not match the **Standard Math Library** parameter, the library specified by the **Standard Math Library** parameter is used.

See Also

More About

- Code replacement libraries
- “Develop a Code Replacement Library” on page 72-20
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

Compare Rounding and Overflow Modes in C/C++ Code and Simulink

This example shows the use of a Simulink® C Caller block to determine the Simulink rounding mode that matches the results obtained from the C/C++ implementation. This technique can help you to identify the correct overflow mode or rounding mode that you need to match when creating entries in a Code Replacement Library.

Introduction to Rounding and Overflow Modes

While creating a code replacement entry for a Simulink model, the saturation mode and the rounding modes need to be specified for operations that can overflow or lose precision. To create Code Replacement Library entries that replace the correct operations, compare the rounding and saturating behavior of the model with the behavior of your implementation in C/C++.

The goal of this example is to find the rounding behavior of the division operation with integer inputs and outputs in C++.

Example Model Setup

The model `CrlMathModeComparison` performs division between two randomly generated inputs using the Simulink Divide block and the C Caller block and compares the results. If there is a mismatch, the Assertion block is activated, and the simulation stops.

The following blocks are used from the Simulink blocks library:

1. Assertion block
2. C Caller block
3. Divide block
4. Relational Operator block
5. Subsystem to generate random numbers

Define Custom C/C++ Implementation of the Division Function in C Caller Block

You can write a custom implementation of the division function to replace the function in the generated code. Specify the source and header files for your implementation by setting the model configuration parameter **Simulation Target > Include Headers**.

Once you set this parameter, the function name appears in the drop-down list inside the C Caller block parameters.

Results

The results from the custom division function using C++ code match the results from the Divide block when the **Integer rounding mode** is set to `Zero` or `Simplest`, as displayed in the scope.

For the other rounding modes, the simulation will generate different results from the C++ implementation.

Example Replacement Table

From the above example, we understand that a division operation between inputs of type `int32` and output of type `int32` in the custom C++ function rounds to zero. A Division block in Simulink with **Integer rounding mode** is set to `Zero` or `Simplest` will give the same behavior.

Hence, while registering a code replacement entry for an integer division operation using the code replacement tool, the rounding mode for the entry can be set to `RTW_ROUND_ZERO` or `RTW_ROUND_SIMPLEST`.

The table `DivisionCrlExample` demonstrates such an entry.

Limitations

For the given example, the model is unable to distinguish between different saturation modes in the Divide block. The overflow condition can be observed when the smallest number that can be represented by int32, i.e. -2147483648 is divided by -1. Since this model uses randomly generated values for division inputs, it is unable to verify the behavior for such edge cases. Hence, it is advisable to further compare the results from the custom C/C++ implementation with the results from a Simulink model using specific inputs that cover any special conditions or edge cases.

Code Replacement Customization for MATLAB Code

- “What Is Code Replacement Customization?” on page 73-3
- “Code You Can Replace from MATLAB Code” on page 73-5
- “Develop a Code Replacement Library” on page 73-13
- “Quick Start Library Development” on page 73-14
- “Identify Code Replacement Library Requirements” on page 73-21
- “Define Code Replacement Library Optimizations” on page 73-23
- “Register Code Replacement Library” on page 73-38
- “Troubleshoot Code Replacement Library Registration” on page 73-42
- “Verify Code Replacement Library” on page 73-43
- “Troubleshoot Code Replacement Misses” on page 73-47
- “Deploy Code Replacement Library” on page 73-50
- “Math Function Code Replacement” on page 73-51
- “Memory Function Code Replacement” on page 73-57
- “Specify In-Place Code Replacement” on page 73-62
- “Data Alignment for Code Replacement” on page 73-67
- “Array Layout and Code Replacement” on page 73-73
- “Allow Shape Agnostic Match” on page 73-75
- “Replace MATLAB Functions with Custom Code by Using the coder.replace Function” on page 73-79
- “Replace coder.ceval Calls to External Functions” on page 73-85
- “Reserved Identifiers and Code Replacement” on page 73-92
- “Customize Match and Replacement Process” on page 73-93
- “Scalar Operator Code Replacement” on page 73-99
- “Addition and Subtraction Operator Code Replacement” on page 73-104
- “Small Matrix Operation to Processor Code Replacement” on page 73-111
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 73-117
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 73-125
- “Remap Operator Output to Function Input” on page 73-132
- “Fixed-Point Operator Code Replacement” on page 73-134
- “Binary-Point-Only Scaling Code Replacement” on page 73-140
- “Slope Bias Scaling Code Replacement” on page 73-145
- “Net Slope Scaling Code Replacement” on page 73-151
- “Equal Slope and Zero Net Bias Code Replacement” on page 73-158

- “Data Type Conversions (Casts) and Operator Code Replacement” on page 73-164
- “Shift Left Operations and Code Replacement” on page 73-170
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - MATLAB” on page 73-175

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the **Code Replacement Tool** or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator's match and replacement process for functions

To get started, "Quick Start Code Replacement Library Development - Simulink" on page 72-21.

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

1. Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
2. Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
   - Conceptual name or key
   - Arguments, including quantity, type, type qualifiers, and complexity
   - Algorithm (computation method)
   - Fixed-point saturation and rounding modes
   - Priority
3. When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
4. Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See "Verify Code Replacement Library" on page 73-43.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is not available from the Code Replacement Tool. See Specify build information in "Define Code Replacement Library Optimizations" on page 73-23.
- Addition and subtraction operation replacements—See "Addition and Subtraction Operator Code Replacement" on page 73-104 for relevant limitations.

- `coder.replace` function — See `coder.replace` for relevant limitations.
- `coder.dataAlignment` function — See `coder.dataAlignment` for relevant limitations.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Develop a Code Replacement Library” on page 73-13
- “Quick Start Library Development” on page 73-14
- “What Is Code Replacement?”

Code You Can Replace from MATLAB Code

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

In this section...

"Math Functions" on page 73-5
"Memory Functions" on page 73-8
"Operators" on page 73-9

Math Functions

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Floating point	Scalar	Real
`acos`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`acosd`	Floating point	Scalar Vector Matrix	Real Complex
`acot`	Floating point	Scalar Vector Matrix	Real Complex
`acotd`	Floating point	Scalar Vector Matrix	Real Complex
`acoth`	Floating point	Scalar Vector Matrix	Real Complex
`acsc`	Floating point	Scalar Vector Matrix	Real Complex
`acscd`	Floating point	Scalar Vector Matrix	Real Complex
`acsch`	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`asec`	Floating point	Scalar Vector Matrix	Real Complex
`asecd`	Floating point	Scalar Vector Matrix	Real Complex
`asech`	Floating point	Scalar Vector Matrix	Real Complex
`asin`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`asind`	Floating point	Scalar Vector Matrix	Real Complex
`atan`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`atan2`	Floating point	Scalar Vector Matrix	Real
`atan2d`	Floating point	Scalar Vector Matrix	Real
`atand`	Floating point	Scalar Vector Matrix	Real Complex
`cos`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`ceil`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`cosd`	Floating point	Scalar Vector Matrix	Real Complex
`cosh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`cot`	Floating point	Scalar Vector Matrix	Real Complex
`cotd`	Floating point	Scalar Vector Matrix	Real Complex
`coth`	Floating point	Scalar Vector Matrix	Real Complex
`csc`	Floating point	Scalar Vector Matrix	Real Complex
`cscd`	Floating point	Scalar Vector Matrix	Real Complex
`csch`	Floating point	Scalar Vector Matrix	Real Complex
`exp`	Floating point	Scalar	Real
`fix`	Floating point	Scalar	Real
`floor`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`hypot`	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real
`log`	Floating point	Scalar Vector Matrix	Real Complex
`log10`	Floating point	Scalar Vector Matrix	Real Complex
`log2`	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point	Scalar	Real
`min`	Integer Floating point	Scalar	Real
`pow`	Floating point	Scalar	Real
`rem`	Floating point	Scalar	Real
`round`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`sec`	Floating point	Scalar Vector Matrix	Real Complex
`secd`	Floating point	Scalar Vector Matrix	Real Complex
`sech`	Floating point	Scalar Vector Matrix	Real Complex
`sign`	Floating point	Scalar	Real
`sin`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sind`	Floating point	Scalar Vector Matrix	Real Complex
`sinh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sqrt`	Floating point	Scalar	Real
`tan`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`tand`	Floating point	Scalar Vector Matrix	Real Complex
`tanh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
[1] Wrap on integer overflow only			

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`memcmp`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memcpy`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset2zero`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Operators

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates you can specify different data types of different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (`+`)[1]	`RTW_OP_ADD`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (`-`)[1]	`RTW_OP_MINUS`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (`*`)[2]	`RTW_OP_MUL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (`/`)	`RTW_OP_DIV`	Integer Floating point Fixed-point Mixed	Scalar	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Data type conversion (cast)	`RTW_OP_CAST`	Integer Floating point[3] Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	`RTW_OP_SL`	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right arithmetic (>>)[4]	`RTW_OP_SRA`	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right logical (>>)	`RTW_OP_SRL`	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Element-wise matrix multiplication (`.*`)[5]	`RTW_OP_ELEM_MUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	`RTW_OP_CONJUGATE`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (`.'`)	`RTW_OP_TRANS`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (`'`)	`RTW_OP_HERMITIAN`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition[2]	`RTW_OP_TRMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition[2]	`RTW_OP_HMMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication followed by shift right arithmetic (`u1*u2>>u3`)[6]	`RTW_OP_MUL_SRA`	Integer Fixed-point	Scalar	Real
Multiplication followed by division (`u1*u2/u3`)[7]	`RTW_OP_MULDIV`	Integer Fixed-point	Scalar	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Greater than (>)	`RTW_OP_GREATER_THAN`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal (>=)	`RTW_OP_GREATER_THAN_OR_EQUAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	`RTW_OP_LESS_THAN`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (<=)	`RTW_OP_LESS_THAN_OR_EUQAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	`RTW_OP_EUQAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (`!=`)	`RTW_OP_NOT_EUQAL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

[1] See “Addition and Subtraction Operator Code Replacement” on page 73-104 for details to consider when defining mappings for addition and subtraction code replacements.

[2] Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.

[3] Scaled floating point is not supported.

[4] Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

[5] Use the multiplication (*) operator (`RTW_OP_MUL`) for scalar multiplication.

[6] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; shift operand is an unsigned integer; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope` and `Mul_output_slope == output_slope_of_shift_operation`).

[7] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope == U3_slope * Div_output_slope`).

See Also

More About

- “Develop a Code Replacement Library” on page 73-13
- “Quick Start Library Development” on page 73-14
- “What Is Code Replacement?”

Develop a Code Replacement Library

Iterate through the following steps, as necessary, to develop a code replacement library:

1. “Identify Code Replacement Library Requirements” on page 73-21
2. “Define Code Replacement Library Optimizations” on page 73-23
3. “Register Code Replacement Library” on page 73-38
4. “Verify Code Replacement Library” on page 73-43
5. “Deploy Code Replacement Library” on page 73-50

To get started, see “Identify Code Replacement Library Requirements” on page 73-21.

To experiment with the process and tools, see “Quick Start Library Development” on page 73-14.

See Also

More About

- “Identify Code Replacement Library Requirements” on page 73-21
- “Code You Can Replace from MATLAB Code” on page 73-5
- “Quick Start Library Development” on page 73-14
- “What Is Code Replacement Customization?” on page 73-3

Quick Start Library Development

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the **Code Replacement Tool**.

Prerequisites

To complete this example, install the following software:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

For instructions on installing MathWorks products, see “Installation and Licensing”. If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

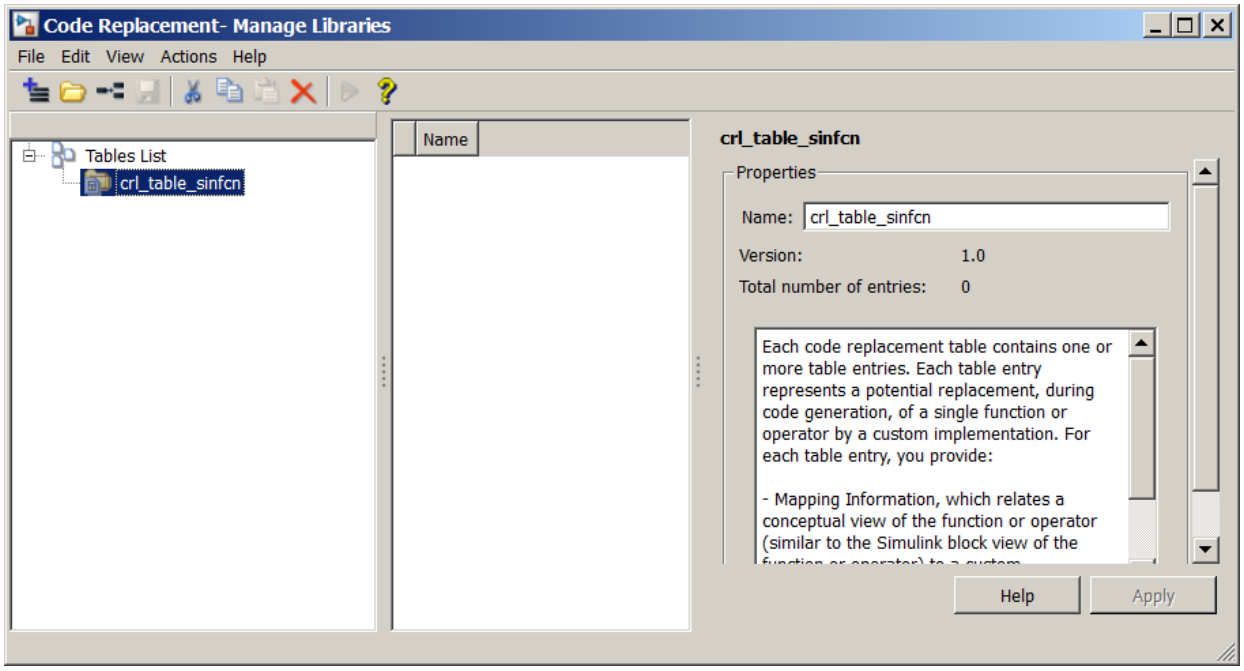
For a list of supported compilers, see `https://www.mathworks.com/support/compilers/current_release/`.

Open the Code Replacement Tool

1. Start a new MATLAB session.
2. Create or navigate (`cd`) to an empty folder.
3. At the command prompt, enter the `crtool` command. The **Code Replacement Tool** window opens.

Create Code Replacement Table

1. In the **Code Replacement Tool** window, select **File** > **New table**.
2. In the right pane, name the table `crl_table_sinfcn` and click **Apply**. Later, when you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.

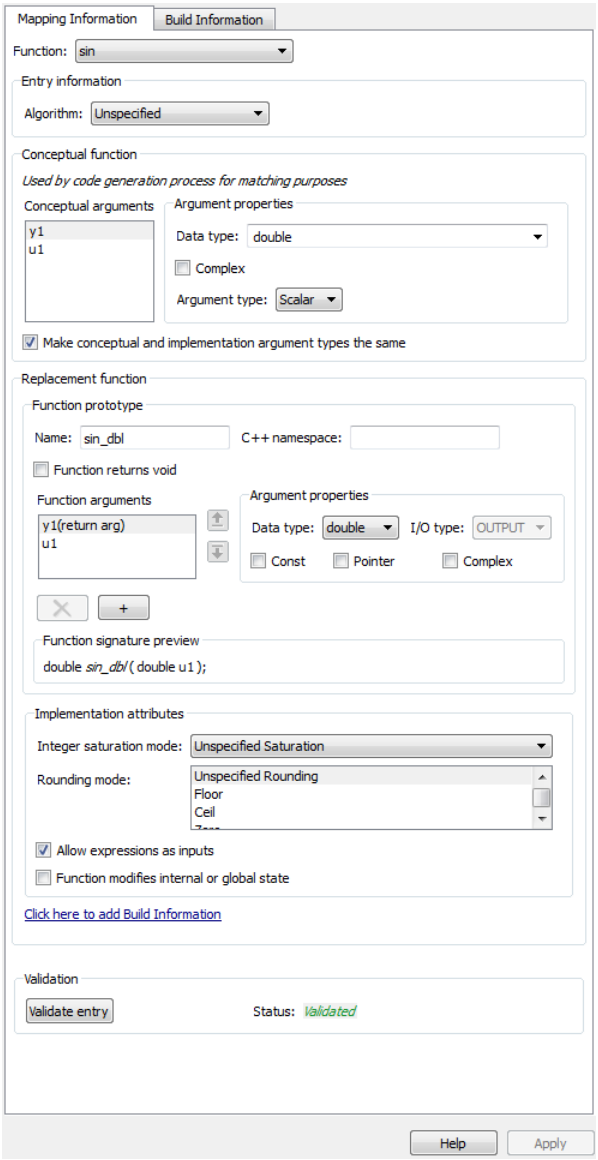


Create Table Entry

Create a table entry that maps a `sin` function with `double` input and `double` output to a custom implementation function.

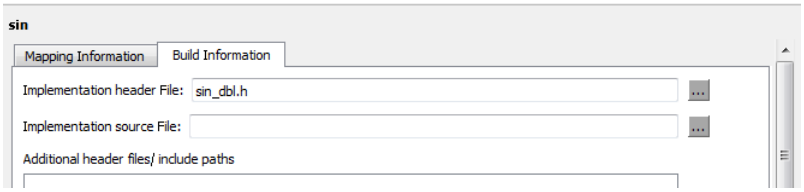
1. In the left pane, select table `crl_table_sinfcn`. Then, select **File** > **New entry** > **Function**. The new entry appears in the middle pane, initially without a name.
2. In the middle pane, select the new entry.
3. In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.
4. Leave **Algorithm** set to `Unspecified`, and leave parameters in the **Conceptual function** group set to default values.
5. In the **Replacement function** group, name the replacement function `sin_dbl`.
6. Leave the remaining parameters in the **Replacement function** group set to default values.
7. Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.
8. Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.



Specify Build Information for Replacement Code

1. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.
2. Leave the remaining parameters set to default values.
3. Click **Apply**.



4 Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

1 In the middle pane, select the `sin_dbl` entry.
2 Select **Edit** > **Copy**
3 Select **Edit** > **Paste**
4 On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.
5 In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.
6 Click **Apply**. Note the changes that appear for the **Function signature preview**.
7 On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

1 Select **Actions** > **Validate table**.
2 If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.

	Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
✓	sin	sin_dbl	1	double		double		100
✓	sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File** > **Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `crl_table_sinfcn.m`.

Review the Code Replacement Table Definition

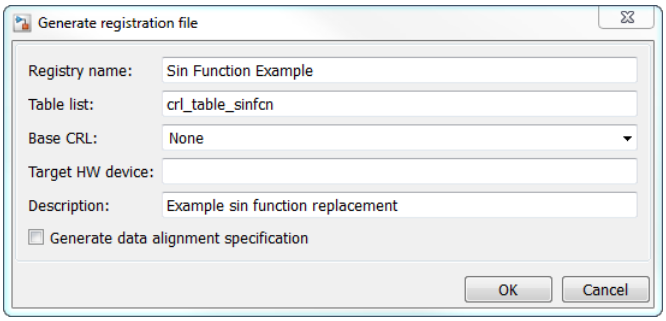
Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

To review it, in MATLAB or another text editor, open the file `crl_table_sinfcn.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

1 In the Code Replacement Tool, select **File** > **Generate registration file**.
2 In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and then click **OK**.



3 In the **Select location to save the registration file** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

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

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

1 Check for errors. At the command line, invoke the table definition file. For example:

If an error exists in the definition file, the invocation triggers a message to appear. Fix the error and try again.

```
tbl = crl_table_sinfcn

tbl =

TflTable with properties:
               Version: '1.0'
       ReservedSymbols: []
   StringResolutionMap: []
             AllEntries: [2x1 RTW.TflCFunctionEntry]
           EnableTrace: 1|
```

2 Use the Code Replacement Viewer to check your code replacement entries. For example:

```
crviewer('Sin Function Example')
```

In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

- Incorrect argument order.
- Conceptual argument names that do not match what is expected by the code generator.
- Incorrect priority settings.

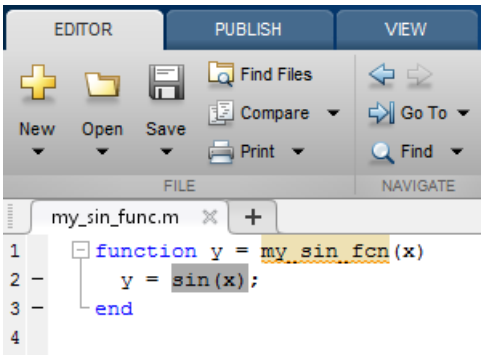
3 Identify existing or create new MATLAB code that calls the `sin` function. For example:

```
function y = my_sin_fnc(x)
  y = sin(x);
end
```

Note If you are replacing a custom function, use the `coder.replace` function.

4. Open the MATLAB Coder app.
5. Add the function that includes a call to the `sin` function as an entry-point file. For example, add `my_sin_func.m`. The app creates a project named `my_sin_func.prj`.
6. Click **Next** to go to the **Define Input Type** step. Define the types for the entry-point function inputs.
7. Click **Next** to go to the **Check for Run-Time Issues** step. This step is optional. However, it is a best practice to perform this step. Provide a test file that calls your entry-point function. The app generates a MEX function from your entry-point function. Then, the app runs the test file, replacing calls to the MATLAB function with calls to the generated MEX function.
8. Click **Next** to go to the **Generate Code** step. To open the **Generate** dialog box, click the **Generate** arrow .
9. Set **Build type** to generate a library or executable.
10. Click **More Settings**.
11. Configure the code generator to use your code replacement library. On the **Custom Code** tab, set the **Code replacement library** parameter to the name of your library. For example, `Sin Function Example`.
12. Configure the code generation report. On the **Debug** tab, set the **Always create a report**, **Code replacements**, and **Automatically launch a report if one is generated** parameters.
13. Configure the code generator to generate code only. For **Build type**, select **Source code**. You want to review your code replacements in the generated code before building an executable.
14. Click **Generate** to generate C code and a report.
15. Review code replacement results in the Code Replacements Report section of the code generation report.

    The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.
16. Review the code replacements. In the report, click the MATLAB function that triggered the replacement, `my_sin_func.m`. The MATLAB Editor opens and highlights the function call that triggers the code replacement.



See Also

More About

- “Develop a Code Replacement Library” on page 73-13
- “What Is Code Replacement Customization?” on page 73-3

Identify Code Replacement Library Requirements

What Is a Code Replacement Library?

A code replacement library defines optimizations for generated C or C++ code from Simulink models or MATLAB code for specific targets. You can configure the code generator to use your code replacement library to replace specified functions in your model or MATLAB code with optimized implementations.

What Functions Can Be Optimized with a Code Replacement Library?

You can use a code replacement library to optimize the performance of functions defined either by Simulink models or MATLAB code. To look up functions that are available to optimize, see “Code You Can Replace From Simulink Models” on page 72-6 and “Code You Can Replace from MATLAB Code” on page 73-5.

How to Develop a Code Replacement Library

You can develop a code replacement library either interactively, using the **Code Replacement Tool** (crtool), or programmatically, using the MATLAB programming interface. The Code Replacement Tool is quick, easy, and ideal for first time library development, while the MATLAB Programming interface provides more customization and control options. To determine which approach to use, identify your use case below:

Interactive Approach:

- Developing a code replacement library for the first time
- Rapid prototyping

Programmatic Approach:

- Specifying code replacement attributes not available with the Code Replacement Tool
- Creating or modifying code replacement library entries by copying, pasting, and editing
- Modifying a registration file by copying and pasting content

Identify Basic Requirements

- What model or MATLAB code do you want to optimize by using a code replacement library?
- What functions and operations within your model or MATLAB code do you want to replace? What is the signature for each function or operation that you want to replace?
- What build files (source, header, object) do you need to provide to the code generator?

See Also

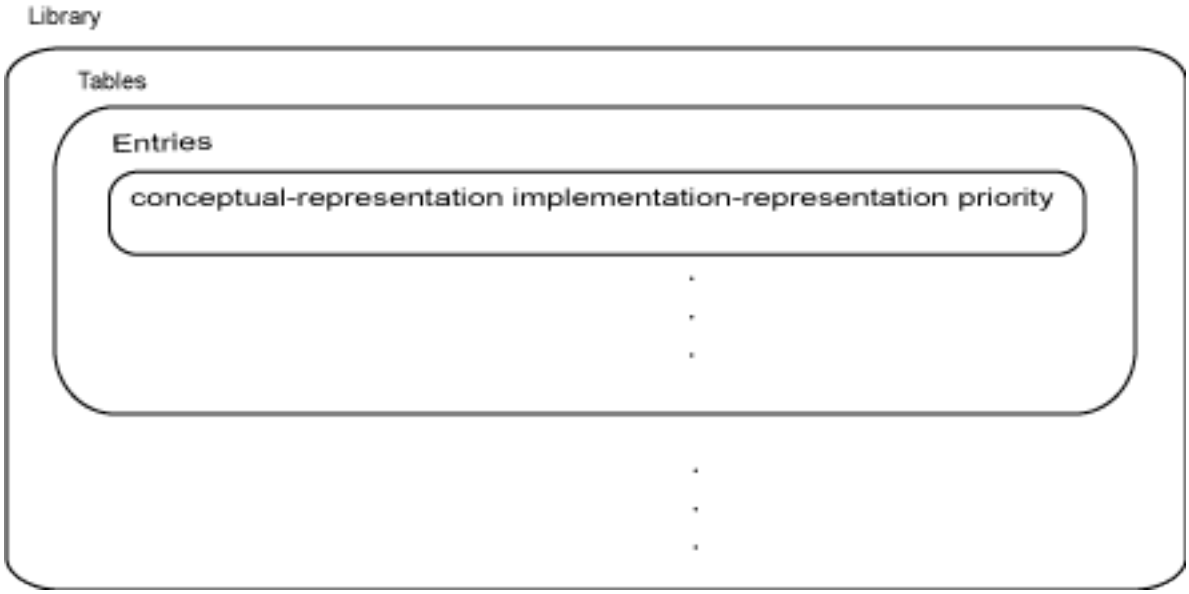
More About

- “Develop a Code Replacement Library” on page 73-13
- “What Is Code Replacement Customization?” on page 73-3

- “Code You Can Replace from MATLAB Code” on page 73-5

Define Code Replacement Library Optimizations

A code replacement library contains one or more code replacement tables. Each table contains one or more code replacement entries. Each entry contains a mapping and entry parameters. The entry maps the conceptual representation of an automatically generated MATLAB or Simulink function to an implementation representation of an optimized C or C++ function. The entry parameters describe additional properties for the code generator to honor during code replacement.



To define a code replacement library, first define code replacement tables in a customization file. Then, during the registration step, create a code replacement library by selecting tables to include in your library. You can define your code replacement tables both interactively or programmatically using the same work flow.

The interactive and programmatic approaches both follow these 8 steps:

1. Open the environment.
2. Create a table.
3. Create an entry.
4. Create entry parameters.
5. Create the conceptual representation.
6. Create the implementation representation.
7. Specify build information.
8. Validate and save.

Interactively Develop a Code Replacement Library

1. Open the environment.

   To interactively create a code replacement library, open the **Code Replacement Tool** (crtool) from the MATLAB command line with the command:

```
>> crtool
```

2. Create a table.

   A code replacement table organizes code replacement entries into groupings that are easier to manage.

   To create a code replacement table:

   a. From the crtool menu, select **File** > **New table**.
   b. In the right pane, name the table and click **Apply**.

3. Create an entry.

   A code replacement table entry contains entry parameters and a mapping of a conceptual representation that describes a generated function to an implementation representation that describes an optimized implementation of that function.

   To create a code replacement entry, use one of these methods:

   - From the crtool toolbar, click the **entry-type** icon the select the **entry-type**.
   - From the crtool menu, select **File** > **New entry** > **entry-type**.
   - Right-click on a table and select **New entry** > **entry-type**.

   The new entry appears without a name in the middle pane.

4. Create entry parameters.

   Entry parameters specify optimization and build requirements for the code generator to honor during code replacement. These parameters vary and automatically appear in various sections of the crtool interface depending on the type of function replacement.

   To create entry parameters:

   - In the **Function** or **Operation** drop-down list, select the name of the function you want to replace.

   This selection determines the entry parameters that appear in the crtool.

5. Create the conceptual representation.

   The conceptual representation describes the signature of the function you want to replace. The conceptual representation consists of the function name, return arguments, and parameters. The return arguments are referred to as output arguments, and the parameters are referred to as input arguments. Specify each conceptual argument with a name, I/O type, and, depending on the type of replacement, other properties that appear in the crtool.

   To define your conceptual representation:

   - In the **Conceptual Function** or **Conceptual Operation** subsection of the crtool, specify the input and output arguments of the function you want the code generator to replace.

6. Create the implementation representation.

   The implementation representation describes the signature of the replacement function defined in the code replacement library. The implementation representation consists of a function name, return arguments, and parameters. Specify each implementation argument with a type, type qualifier, and complexity.

To define your implementation representation:

- In the **Implementation Function** or **Operation Function** subsection of the crtool, specify the input and output arguments for the optimized function defined in the code replacement library.

7 Specify build information.

Build information provides the code generator with the files to perform function replacement.

To specify build information:

- In the crtool, click the **Build Information** tab to open the build requirements pane. Specify necessary build information in these fields:

Parameter	Description
Implementation header file	Name of header file needed for code replacement.
Implementation source file	Name of source file needed for code replacement.
Additional header files/include paths	Name and path of additional header files needed for code replacement.
Additional source files/paths	Name and path of additional source files needed for code replacement.
Additional object files/paths	Name and path of additional object files needed for code replacement.
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build directory	Flags whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as packNGo might find duplicate instances of files.

8 Validate and save.

Save your code replacement library with the same name as your code replacement table on your MATLAB path. Saving your library automatically validates unvalidated content.

To validate and save your library:

- From the crtool menu, select **File** > **Save Table** > **Save**.

Programmatically Develop a Code Replacement Library

1. Open the environment.

   To programmatically create a code replacement library, open a MATLAB function file. From the MATLAB menu select **New** > **Function**.

2. Create a table.

   A code replacement table organizes code replacement entries into groupings that are easier to manage.

   To create a code replacement table:

   a. Create a function you can use to call your code replacement library table. The function should not have arguments and return a table object.

   b. Create a table object with a call to `RTW.TflTable`.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

end
```

3. Create an entry.

   A code replacement table entry contains entry parameters and a mapping of a conceptual representation of a generated function to the implementation representation of an optimized implementation of that function.

   To create a code replacement table:

   a. Identify the type of function you want to replace.

   b. Call the corresponding Entry Function to create an entry in your table.

Type of Function You Want to Replace	Entry Function
Math Operation	`RTW.TflCOperationEntry`
Function	`RTW.TflCFunctionEntry`
BLAS operation	`RTW.TflBlasEntryGenerator`
CBLAS operation	`RTW.TflCBlasEntryGenerator`
Fixed-point addition and subtraction operations	`RTW.TflCOperationEntryGenerator`
Net slope fixed-point operation	`RTW.TflCOperationEntryGenerator_NetSlope`
Semaphore or mutex	`RTW.TflCSemaphoreEntry`
Custom function	`RTW.TflCFunctionEntryML`
Custom operation	`RTW.TflCOperationEntryML`

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table
```

```
%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

end
```

4 Create entry parameters.

Entry parameters specify optimization and build requirements for the code generator to honor during code replacement. These parameters vary depending on the type of function replacement.

To create your entry parameters:

- Call the Set Parameters function that corresponds with the type of function you want to replace identified in the previous step (step 3).

Type of Function You Want to Replace	Set Parameters function
Math operation, BLAS operation, CBLAS operation, Fixed-Point, Net Slope Fixed-Point, Custom operation	`setTflCOperationEntryParameters`
Function, Custom function	`setTflCFunctionEntryParameters`
Semaphore or Mutex	`setTflCSemaphoreEntryParameters`

Each Set Parameter function is shown with all available properties set to their default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;
```

```
%% Create entry parameters
% Pick one from list below
hEntry.setTflCOperationEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ...
        'AllowShapeAgnostic',           false, ...
        'ArrayLayout',                  'COLUMN_MAJOR', ...
        'EntryInfoAlgorithm',           'RTW_CAST_BEFORE_OP', ...
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'MustHaveZeroNetBias',          false, ...
        'NetFixedExponet',              0, ...
        'NetAdjustmentFactor', 1,       ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
        'SlopesMustBeTheSame',          false, ...
        'StoreFcnReturnInLocalVar',     false);

hEntry.setTflCFunctionEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ...
        'ArrayLayout',                  'COLUMN_MAJOR', ...
        'EntryInfoAlgorithm',           'RTW_DEFAULT', ...
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
```

```
        'StoreFcnReturnInLocalVar',     false);

hEntry.setTflCSemaphoreEntryParameters(...
        'Key',                          ' ', ...
        'Priority',                     100, ...
        'AcceptExprInput',              true, ...
        'AdditionalHeaderFiles',        {}, ...
        'AdditionalIncludePaths',       {}, ...
        'AdditionalLinkObjs',           {}, ...
        'AdditionalLinkObjsPaths',      {}, ...
        'AdditionalSourceFiles',        {}, ...
        'AdditionalSourcePaths',        {}, ...
        'AdditionalCompileFlags',       {}, ...
        'AdditionalLinkFlags',          {}, ....
        'GenCallback',                  ' ', ...
        'ImplementationHeaderFile',     ' ', ...
        'ImplementationHeaderPath',     ' ', ...
        'ImplementationName',           ' ', ...
        'ImplementationSourceFile',     ' ', ...
        'ImplementationSourcePath',     ' ', ...
        'ImplType',                     'FCN_IMPL_FUNCT', ...
        'RoundingModes',                'RTW_ROUND_UNSPECIFIED', ...
        'SaturationMode',               'RTW_SATURATE_UNSPECIFIED', ...
        'SideEffects',                  false, ...
        'StoreFcnReturnInLocalVar',     false);

end
```

5 Create the conceptual representation.

The conceptual representation describes the signature of the function you want to replace. It consists of the function name, return arguments, and parameters. The return arguments are referred to as output arguments, and the parameters are referred to as input arguments. Specify each conceptual argument with the required properties, and, depending on the type of replacement, other optional properties.

Required properties:

- **Name**- Defined as y1..yn for output arguments and u1..un for input arguments
- **I/O type**- Defined as `RTW_IO_OUTPUT` or `RTW_IO_INPUT`

To define your conceptual representation, use one of these approaches:

Approach	Function	Goal
1	`createAndAddConceptualArg`	Customization and control. If you want to explicitly specify arguments as scalar or matrix and other properties.
2	`getTflArgFromString`	Rapid prototyping. If you want to quickly specify arguments with built-in data types.

Programming approaches are shown with all available properties set to default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

%% Create entry parameters
% Pick one from list below
% To view properties, see step 4
hEntry.setTflCOperationEntryParameters( );
hEntry.setTflCFunctionEntryParameters( );
hEntry.setTflCSemaphoreEntryParameters( );

%% Create the conceptual representation

% Approach 1: createAndAddConceptualArg
% arg = createAndAddConceptualArg(hEntry, argType, varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type

% Define Output Arguments
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'y1', ...
        'IOType',                       'RTW_IO_OUTPUT', ...
        'IsSigned',                     true, ...
        'WordLength',                   32, ...
        'CheckSlope',                   true, ...
        'CheckBias',                    true, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'BaseType',                     'double', ...
        'DimRange',                     [2,2]);

% Define Input Arguments
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'u1', ...
        'IOType',                       'RTW_IO_INPUT', ...
```

```
        'IsSigned',                     true, ...
        'WordLength',                   32, ...
        'CheckSlope',                   true, ...
        'CheckBias',                    true, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'BaseType',                     'double', ...
        'DimRange',                     [2,2]);


% Approach 2: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, I/O type)
% name- y1..yn | u1..un
% I/O type- RTW_IO_OUTPUT | RTW_IO_INPUT

% Define Output Arguments
arg = getTflArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

% Define Input Arguments
arg = getTflArgFromString('u1', 'double');
arg.IOType = 'RTW_IO_INPUT';
hEntry.addConceptualArg(arg);

end
```

6 Create the implementation representation.

The implementation representation describes the signature of the replacement function defined by the code replacement library. It consists of the function name, return arguments, and parameters. Specify the implementation arguments with a type, type qualifier, and complexity. After you define your implementation representation, add your completed entry to the code replacement table with a call to the function `addEntry`.

To define your implementation representation, use one of these approaches:

Approach	Function	Goal
1	`copyConceptualArgsToImplementation`	Identical arguments. Specify implementation arguments and conceptual arguments have the same properties.
2	`createAndAddImplementationArg` `createAndSetCImplementationReturn`	Same order, different properties. Specify implementation arguments have the same order as conceptual arguments but have different properties.

Approach	Function	Goal
3	`getTflArgFromString`	Add constant value. Specify implementation arguments with a constant value.
4	`getTflDWorkFromString`	Mutex/Semaphore replacement. Specify implementation arguments for mutex or semaphore functions.
5	`createCRLEntry`	Create entry with one function. Declare and define a code replacement table entry (conceptual and implementation arguments) in one function.

Programming approaches are shown with all available properties set to their default values. The exact properties and values you need to specify depends on the function you replace. See examples for specific replacement details.

```
function hTable = code_replacement_library_table()
% Create a function to call code replacement library table

%% Create a code replacement library table object
hTable = RTW.TflTable;

%% Create a code replacement library entry
% Pick one from list below
hEntry = RTW.TFLCOperationEntry;
hEntry = RTW.TflCFunctionEntry;
hEntry = RTW.TflBlasEntryGenerator;
hEntry = RTW.TflCBlasEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator;
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
hEntry = RTW.TflCSemaphoreEntry;
hEntry = RTW.TflCFunctionEntryML;
hEntry = RTW.TflCOperationEntryML;

%% Create entry parameters
% Pick one from list below
% To view properties, see step 4
hEntry.setTflCOperationEntryParameters( );
hEntry.setTflCFunctionEntryParameters( );
hEntry.setTflCSemaphoreEntryParameters( );

%% Create the conceptual representation

% Approach 1: createAndAddConceptualArg
% arg = createAndAddConceptualArg(hEntry, argType, varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type, to view properties, see step 5

% Define Output Arguments
hEntry.createAndAddConceptualArg( );
```

```
% Define Input Arguments
hEntry.createAndAddConceptualArg( );



% Approach 2: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, I/O type)
% name- y1..yn | u1..un
% I/O type- RTW_IO_OUTPUT | RTW_IO_INPUT

% Define Output Arguments
arg = getTflArgFromString('y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

% Define Input Arguments
arg = getTflArgFromString('u1', 'double');
arg.IOType = 'RTW_IO_INPUT';
hEntry.addConceptualArg(arg);



%% Create the implementation representation

% Approach 1: copyConceptualArgsToImplementation
copyConceptualArgsToImplementation(hEntry);


% Approach 2: createAndSetCImplementation, createAndSetCImplementationArg
% arg = createAndAddImplementation(hEntry, argType,varargin),
% arg = createAndSetCImplementationReturn(hEntry, argType,varargin)
% argType - 'RTW.TflArgNumeric' | 'RTW.TflArgMatrix'
% varargin - 'name', 'value' pairs shown below set to default values,
% includes name and I/O type

% Define Output Arguments
hEntry.createAndSetCImplementationReturn(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'y1', ...
        'IOType',                       'RTW_IO_OUTPUT', ...
        'IsSigned',                     true, ...
        'WordLength',                   16, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1.0, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'Value',                        0);

% Define Input Arguments
hEntry.createAndAddImplementationArg(...
        'RTW.TflArgNumeric',            ...
        'Name',                         'u1', ...
        'IOType',                       'RTW_IO_INPUT', ...
```

```
        'IsSigned',                     true, ...
        'WordLength',                   16, ...
        'DataTypeMode',                 'Fixed-point:binary point scaling', ...
        'DataType',                     'Fixed', ...
        'Scaling',                      'BinaryPoint', ...
        'Slope',                        1.0, ...
        'SlopeAdjustmentFactor',        1.0, ...
        'FixedExponent',                -15, ...
        'Bias',                         0.0, ...
        'FractionLength',               15, ...
        'Value',                        0);


% Approach 3: getTflArgFromString
% arg = getTflArgFromString(hEntry, name, datatype)
% name- y1..yn | u1..un, datatype - built-in data type

% Define Output arguments
arg = getTflArgFromString(hEntry, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

% Define Input arguments
arg = getTflArgFromString(hEntry, 'u1', 'double', 0);
hEntry.Implementation.addArgument(arg);



% Approach 4: getTflDWorkFromString
% arg = getTflDWorkFromString(hEntry, name, datatype)

% Define arguments
arg = getTflDWorkFromString('d1', 'void*');
hEntry.addDWorkArg(arg);



% Approach 5: createCRLEntry
% tableEntry = createCRLEntry(crTable,conceptualSpecification,implementationSpecification)

% Define code replacement conceptual and implementation arguments
hEntry = createCRLEntry(...
        hTable, ...
        'conceptualSpecification', ...
        'implementationSpecification');


%% Add code replacement library entry to table
hTable.addEntry(hEntry);

end
```

7 Specify build information.

Specify build information for your code replacement library. Build information provides the code generator with the files necessary to perform function replacement.

To specify build information:

- Specify your build parameters in the Entry Parameters function you defined in step 4 or with the specified functions. Specify build information with the following properties:

Parameter	Description
Implementation header file Set properties `ImplementationHeaderFile` and `ImplementationHeaderPath`.	Name of header file needed for code replacement.
Implementation source file Set properties `ImplementationSourceFile` and `ImplementationSourcePath`.	Name of source file needed for code replacement.
Additional header files/include paths For each file, specify the file name and path in calls to the functions `addAdditionalHeaderFile` and `addAdditionalIncludePath`.	Name and path of additional header files needed for code replacement.
Additional source files/paths For each file, specify the file name and path in calls to the functions `addAdditionalSourceFile` and `addAdditionalSourcePath`.	Name and path of additional source files needed for code replacement.
Additional object files/paths For each file, specify the file name and path in calls to the functions `addAdditionalLinkObj` and `addAdditionalLinkObjPath`.	Name and path of additional object files needed for code replacement.
Additional link flags Set the entry property `AdditionalLinkFlags` to a cell array of character vectors or string array representing the required compile flags.	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags Set the entry property `AdditionalCompileFlags` to a cell array of character vectors or string array representing the required link flags.	Flags the compiler needs to generate object code for the replacement code.

Parameter	Description
Copy files to build directory. Set property `GenCallback` to `'RTW.copyFileToBuildDir'`. If a match occurs for a table entry, a call to the function `RTW.copyFileToBuildDir` copies required files to the build folder. If you specify additional header files/include paths or additional source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.	Flags whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as packNGo might find duplicate instances of files.

- If an entry uses header, source, or object files, consider whether to make the files accessible to the code generator. You can copy files to the build folder or you can specify individual file names and paths explicitly.
- If you specify *additional* header files/include paths or source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files (an instance in the build folder and an instance in the original folder).
- If you choose to copy files to the build folder and you are using the `packNGo` function to relocate static and generated code files to another development environment:
  - In the call to `packNGo`, specify the property-value pair `'minimalHeaders' true` (the default). That setting instructs the function to include the minimal header files required to build the code in the zip file.
  - Do not collocate files that you copy with files that you do not copy. If the `packNGo` function finds multiple instances of the same file, the function returns an error.
- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace that you enclose with dollar signs ($*variable*$). The code generator evaluates and replaces a token with the defined value. For example, consider the path `$myfolder$\folder1`, where `myfolder` is a character vector or string scalar variable defined in the MATLAB workspace as `'d:\work\source\module1'`. The code generator generates the custom path as `d:\work\source\module1\folder1`.

8 Validate and save.

Save your code replacement library customization file with the same name as your code replacement library table on your MATLAB path.

To save and validate your library:

a Save your file. From the MATLAB menu, select **File** > **Save**.

b Validate your library by calling it from the MATLAB command line. For example:

```
>> hTable = code_replacement_library_table()
```

See Also

More About

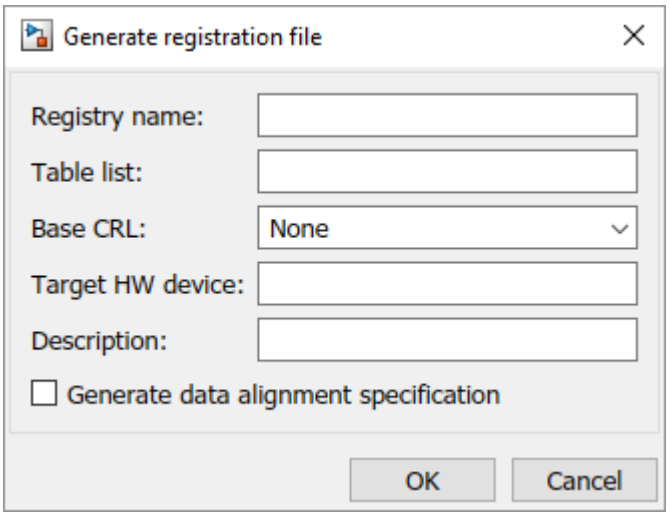
- “What Is Code Replacement Customization?” on page 73-3
- “Code You Can Replace from MATLAB Code” on page 73-5
- “Develop a Code Replacement Library” on page 73-13
- “Math Function Code Replacement” on page 73-51
- “Memory Function Code Replacement” on page 73-57
- “Specify In-Place Code Replacement” on page 73-62
- “Replace MATLAB Functions with Custom Code by Using the coder.replace Function” on page 73-79
- “Reserved Identifiers and Code Replacement” on page 73-92
- “Customize Match and Replacement Process” on page 73-93
- “Scalar Operator Code Replacement” on page 73-99
- “Addition and Subtraction Operator Code Replacement” on page 73-104
- “Small Matrix Operation to Processor Code Replacement” on page 73-111
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 73-117
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 73-125
- “Remap Operator Output to Function Input” on page 73-132
- “Customize Match and Replacement Process for Operators” on page 73-94
- “Fixed-Point Operator Code Replacement” on page 73-134
- “Binary-Point-Only Scaling Code Replacement” on page 73-140
- “Slope Bias Scaling Code Replacement” on page 73-145
- “Net Slope Scaling Code Replacement” on page 73-151
- “Equal Slope and Zero Net Bias Code Replacement” on page 73-158
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 73-164
- “Shift Left Operations and Code Replacement” on page 73-170

Register Code Replacement Library

Create a code replacement library by defining in a registration file the library name, code replacement tables, and other information. After you define and register your library it becomes available to the code generator for code replacement.

Interactively Develop and Load a Registration file

1 In the crtool, select **File** > **Generate registration file**.



2 In the **Generate registration file** dialog box, specify registration information. Then click **OK**.

Parameter	Description
Registry name (Required)	Code replacement library name.
Table list (Required)	Comma-separated list of one or more code replacement tables that you want in the library. Specify tables by using either their names (if located on the MATLAB search path), absolute paths, or relative paths.
Base CRL	Enables you to specify library hierarchies. Provide the name of a registered library to serve as a base for the library that you are currently registering. For example, a TI device library can be a base for a TI C28x library.
Target HW device	Comma-separated list of one or more hardware devices that this library supports. Indicate support of all devices with an asterisk (*).
Description	Description of the purpose and content of the library.

Parameter	Description
Generate data alignment specification	Flag that enables data alignment.

3 To use your registered code replacement library, refresh your current MATLAB session with the command:

```
>> sl_refresh_customizations
```

Programmatically Develop and Load a Registration File

1 Open a MATLAB function file. From the MATLAB menu, select **New** > **Function**.

2 Save the function file with the name `rtwTargetInfo.m`.

3 Copy and paste the following code into your file to use as a template.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry; %instantiates a registration entry
this(1).Name = 'Example Code Replacement Library';
this(1).TableList = {'example_code_replacement_table.m'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'This registers an example library';

end
```

4 In the template, specify the following registration information:

Parameter	Description
Registry name (Required)	Code replacement library name.
Table list (Required)	Comma-separated list of one or more code replacement tables that you want in the library. Specify tables by using either their names (if located on the MATLAB search path), absolute paths, or relative paths.
Base CRL	Enables you to specify library hierarchies. Provide the name of a registered library to serve as a base for the library that you are currently registering. For example, a TI device library can be a base for a TI C28x library.
Target HW device	Comma-separated list of one or more hardware devices that this library supports. Indicate support of all devices with an asterisk (*).
Description	Description of the purpose and content of the library.
Generate data alignment specification	Flag that enables data alignment.

5 Save the file on your MATLAB path.

6 To use your registered code replacement library, refresh your current MATLAB session with the command:

```
>> sl_refresh_customizations
```

Examples

Register one code replacement library

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry; %instantiates a registration entry
this(1).Name = 'Example Code Replacement Library';
this(1).TableList = {'example_code_replacement_table.m'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'This registers an example library';

end
```

Register multiple code replacement libraries

```
function rtwTargetInfo(cm)
cm.registerTargetInfo(@loc_register_crl);
end

function thisCRL = loc_register_crl
  % Register a code replacement library for use with model: rtwdemo_crladdsub
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Addition & Subtraction Examples';
  thisCrl(1).Description = 'Example of addition/subtraction op replacement';
  thisCrl(1).TableList = {'crl_table_addsub'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlmuldiv
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'Multiplication & Division Examples';
  thisCrl(2).Description = 'Example of mult/div op repl for built-in integers';
  thisCrl(2).TableList = {'c:/work_crl/crl_table_muldiv'};
  thisCrl(2).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlfixpt
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'Fixed-Point Examples';
  thisCrl(3).Description = 'Example of fixed-point operator replacement';
  thisCrl(3).TableList = {fullfile('$(MATLAB_ROOT)', ...
     'toolbox','rtw','rtwdemos','crl_demo','crl_table_fixpt')};
   thisCrl(3).TargetHWDeviceType = {'*'};

end
```

Register a code replacement hierarchy

```
function rtwTargetInfo(cm)
```

```
cm.registerTargetInfo(@loc_register_crl);
end

function thisCRL = loc_register_crl
  % Register a code replacement library that includes common entries
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Common Replacements';
  thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
  thisCrl(1).TableList = {'crl_table_general'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for TI devices
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'TI Device Replacements';
  thisCrl(2).Description = 'Code replacement entries shared across TI devices';
  thisCrl(2).TableList = {'crl_table_TI_devices'};
  thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
  thisCrl(1).BaseTfl = 'Common Replacements'

  % Register a code replacement library for TI c6xx devices
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c6xx Device Replacements';
  thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
  thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
  thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

  % Register a code replacement library for the TI c67x device
  thisCrl(4) = RTW.TflRegistry;
  thisCrl(4).Name = 'TI c67x Device Replacements';
  thisCrl(4).Description = 'Code replacement entries for the TI C67x device';
  thisCrl(4).TableList = {'crl_table_TIC67x_device'};
  thisCrl(4).TargetHWDeviceType = {'TI 67x'};


end
```

See Also

More About

- “Troubleshoot Code Replacement Library Registration” on page 73-42
- “Develop a Code Replacement Library” on page 73-13
- “What Is Code Replacement Customization?” on page 73-3
- “Code You Can Replace from MATLAB Code” on page 73-5

Troubleshoot Code Replacement Library Registration

If a code replacement library is not listed as a configuration option or does not appear in the Code Replacement Viewer:

- Refresh the library registration information within the current MATLAB session (`RTW.TargetRegistry.getInstance('reset');` or for the Simulink environment,`sl_refresh_customizations`).
- See whether the registration file, `rtwTargetInfo.m`, contains an error.

See Also

More About

- “Register Code Replacement Library” on page 73-38

Verify Code Replacement Library

Verify your code replacement library by examining its structure, replacement, and performance. Use the Code Replacement Viewer (crviewer) to verify the structure of your library. Use the code replacement report to verify the replacements made with your library in generated code. Use crviewer again to verify the performance of your library during code generation by examining the replacement hit and miss logs.

Verify Library Structure

Use the Code Replacement Viewer (crviewer) to open and view the structure of your code replacement library by examining its tables and table entries.

1. From the MATLAB command prompt, open your code replacement library using the crviewer with the command:

   ```
   >> crviewer('library')
   ```

2. The viewer opens your code replacement library and shows its code replacement tables in the left pane. Verify that your library lists all required tables in the correct relative order.
3. Open and view the entries for each code replacement table. In the crtool, click each table listed in the left pane to view its entries in the middle pane.

   If you are not using the crviewer, you can open each table directly from the MATLAB command prompt with the command:

   ```
   crviewer(table)
   ```

4. For each table in your library, verify that it lists all required entries in the correct relative order. Verify that the conceptual representation and implementation representation are correct for each entry in the table.

Verify Library Replacement

Use the Code Replacements Report to verify that when the code generator uses your code replacement library it creates replacements as expected in the generated code. For each instance of replacement, the Code Replacements Report provides a mapping between the Simulink block that triggered the replacement and its implementation in the generated code. Use this mapping to verify correct code replacement by tracing between the report and the code in either direction (from report to code or code to report).

To generate the Code Replacements Report:

1. Configure your model to use your code replacement library.
2. Configure your model to open the Code Generation Report. The Code Replacements Report is a tab within this report. In the Model Configuration Parameters dialog box, in the **Code Generation** > **Report** pane, select:

   - **Create code generation report**
   - **Open report automatically**
   - **Model-to-code**

- **Summarize which block triggered code replacements**

3. Configure the generated code to include comments. In the Model Configuration Parameters dialog box, in the **Code Generation** > **Comments** pane, select:

   - **Include comments**
   - Either or both of **Simulink block comments** and **Simulink block descriptions**

4. Build your model to generate code only. The Code Generation Report opens automatically. On the left pane, click to open the Code Replacements Report.

5. Verify that code replacements occur as expected.

   - **Navigate from report to model**: In the Code Replacements Report, click each block link to see the source that triggered replacement.
   - **Navigate from model to report**: In the Simulink model window, use model-to-code highlighting to trace code replacements. Right-click blocks where you expected code replacement to occur and select **C/C++ Code** > **Navigate to C/C++ Code**. The report appears with the corresponding replacement code highlighted.

Verify Library Performance

Verify the performance of your code replacement library by examining the hits and misses that occurred during code generation using your library. During code generation, when the code generator finds a match for code replacement from your code replacement library, it logs the hit in a hit cache. When it does not find a code replacement match, it logs the failed attempt in the miss cache. Each time that you generate code, the code generator overwrites its previous hit and miss cache logs.

To review trace information from the hit and miss cache data, use the Code Replacement Viewer (crviewer). From the hit cache, you can identify when code replacement successfully occurred, and from the miss cache, you can troubleshoot when code replacement did not take place. The miss cache trace information provides you information about the reason for the miss, a link to the relevant source location for the miss, and the call site object of the miss. For more information, see “Troubleshoot Code Replacement Misses” on page 72-59.

Review Code Replacements

After you review the content of your code replacement library and tables, generate code and a code generation report. Verify that the code generator replaces code as you expect.

The Code Replacements Report details the code replacement library functions that the code generator uses for code replacements. The report provides a mapping between each replacement instance and the line of MATLAB code that triggered the replacement. The Code Replacements report is not available for generated MEX functions.

The following example illustrates two complementary approaches for reviewing code replacements:

- Check the Code Replacements Report section of the code generation report for expected replacements.
- Trace code replacements.

1. Identify the MATLAB function where you anticipate that a function or operator replacement occurs. This example uses the function *matlabroot*/toolbox/rtw/rtwdemos/crl_demo/addsub_two_int16.m.

   ```
   function [y1, y2] = addsub_two_int16(u1, u2)

   y1 = int16(u1 + u2);
   y2 = int16(u1 - u2);
   ```

2. Identify or create code or a script to exercise the function. For example, consider test file addsub_to_int16_test.m, which includes the following code:

   ```
   disp('Input')
   u1 = int16(10)
   u2 = int16(10)

   [y1, y2] = addsub_two_int16(u1, u2);

   disp('Output')
   disp('y1 =')
   disp(y1);
   disp('y2 =')
   disp(y2);
   ```

3. Open the MATLAB Coder app.
4. On the **Select Source Files** page, add your function to the project. For this example, add function addsub_two_int16. Click **Next**.
5. On the **Define Input Types** page, use the test file addsub_to_int16_test to automatically define the input types. Click **Next**.
6. On the **Check for Run-Time Issues** page, specify the test file addsub_to_int16_test. The app runs the test file, replacing calls to addsub_to_int16_test with calls to a MEX version of addsub_to_int16_test. Click **Next**.
7. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
8. Set **Build type** to generate source code. Before you build an executable, you want to review your code replacements in the generated code.
9. In the Generate dialog box, click **More Settings**.
10. Configure the code generator to use your code replacement library. On the **Custom Code** tab, set the **Code replacement library** parameter to the name of your library. For this example, set the library to Addition & Subtraction Examples.
11. Configure the code generation report to include the Code Replacements Report. On the **Debugging** tab, select:

    - **Always create a code generation report**
    - **Code replacements**
    - **Automatically launch a report if one is generated**
12. To generate code and a report, set **Build type** to Source Code. Then, click **Generate**.
13. Open the **Code Replacements Report** section of the code generation report.

    That report lists the replacement functions that the code generator used. The report provides a mapping between each replacement instance and the MATLAB code that triggered the replacement.

Review the report:

- Check whether expected function and operator code replacements occurred.
- In the replacements sections, click each code link to see the source that triggered the reported code replacement.

If a function or operator is not replaced as expected, the code generator used a higher-priority (lower-priority value) match or did not find a match.

To analyze and troubleshoot code replacement misses, use the trace information that the **Code Replacement Viewer** provides.

See Also

More About

- “Troubleshoot Code Replacement Misses” on page 73-47
- “Register Code Replacement Library” on page 73-38
- “Deploy Code Replacement Library” on page 73-50
- “Develop a Code Replacement Library” on page 73-13
- “What Is Code Replacement Customization?” on page 73-3

Troubleshoot Code Replacement Misses

Use miss reason messages that appear in the **Code Replacement Viewer** to analyze and correct code replacement misses.

Miss Reason Messages

The Code Replacement Viewer displays miss reason messages in trace information for code replacement misses. A legend listing each message that appears in the miss report precedes the report details. A message consists of:

- Numeric identifier, which identifies the message in the report details.
- Message text, which in some cases includes placeholders for names of arguments, call site object values, table entry values, and property names.

For example:

```
1. Mismatched data types (argument name, CSO value, table entry value)
```

The parenthetical information represents placeholders for actual values that appear in the report details.

In the **Miss Source Locations** table that lists the miss details, the **Reason** column includes:

- The message identifier, as listed in the legend.
- The placeholder values for that instance of the miss reason message.

The following **Reason** details indicate a data type mismatch because the call site object specifies data type `int8` for arguments `y1`, `u1`, and `u2`, while the code replacement table entry specifies `uint32`.

```
1. y1, int8, uint32
   u1, int8, uint32
   u2, int8, uint32
```

Depending on your situation and the reported miss reason, troubleshoot reported misses by looking for instances of the following:

- A typo in the code replacement table entry definition or a source parameter setting.
- Information missing from the code replacement table entry or a source parameter setting.
- Invalid or incorrect information in the code replacement table entry definition or a source parameter setting.
- Arguments incorrectly ordered in the code replacement table entry definition or the source being replaced with replacement code.
- Failed algorithm classification for an addition or subtraction operation due to:
  - An ideal accumulator not being calculated because the type of an input argument is not fixed-point or the slope adjustment factors of the input arguments are not equal.
  - Input or output casts with a floating-point cast type.
  - Input or output casts with cast types that have different slope adjustment factors or biases.
  - Output casts not being convertible to a single output cast.

- Input casts resulting in loss of bits.

Analyze and Correct Code Replacement Misses

The following example shows how to use Code Replacement Viewer trace information to troubleshoot code replacement misses. You must have already reviewed and tested code replacements for your MATLAB code.

1. Review the code generated for a specific code element, looking for expected code replacement. Regenerate or reopen the code generation report for your MATLAB code. If you already generated the code generation report that includes the Code Replacements Report for *matlabroot*`/toolbox/rtw/rtwdemos/crl_demo/addsub_two_int16.m`, open the file `codegen/lib/addsub_two_int16/html/mldatx.html`. For information on how to regenerate the report, see "Verify Code Replacement Library" on page 73-43.

   To examine the code generated for the function, from the code generation report, open the generated file `addsub_two_int16.h`.

   ```
   extern void addsub_two_int16(double u1, double u2, short *b_y1, short *y2);
   ```

   The code generator replaced code, but the replacement is for the signed version of the 16-bit addition and subtraction operations. You expected code replacements for operations on unsigned data.
2. Open the Code Replacements Report for the MATLAB code.
3. Click the link to open the Code Replacement Viewer.
4. In the viewer left pane, select your code replacement table. For this example, select code replacement table `crl_table_addsub`.
5. In the middle pane, select table entry `RTW_OP_ADD` with implementation function `u16_add_u16_u16`.
6. In the right pane, select the **Trace Information** tab.

   The **Trace Information** is a table that lists the following information for each miss:

   - Call site object preview. The call site object is the conceptual representation of addition operator. The code generator uses this object to query the code replacement library for a match.
   - A link to the source location in the MATLAB function where the code generator considered replacing code.
   - The reasons that the miss occurred. See "Miss Reason Messages" on page 73-47.

   For this example, the report shows misses for function `addsub_two_int16.m`.
7. Find the source in the trace information. Depending on your situation and the reported miss reason, consider looking for a condition such as a typo in the code replacement table entry definition or a source parameter setting. For a list of conditions to consider, see"Miss Reason Messages" on page 73-47.

   For this example, determine why code for function `addsub_two_int16` is not replaced with code for an unsigned 16-bit addition operation. The miss reasons for the function indicate data type and algorithm mismatches:

   - The data type in the call site object is a signed 16-bit integer. The code replacement entry specifies an unsigned 16-bit integer.

- The algorithm property in the call site object is `RTW_CAST_AFTER_OP` while the code replacement entry specifies `RTW_CAST_BEFORE_OP`.

8 Fix the specified MATLAB code and relevant specifications or code replacement table entry. If the issue concerns the MATLAB code, use the source location in the trace information to find the code to fix. For this example, you expected an unsigned addition operation to occur for the `addsub_two_int16` function.

To fix the mismatches, in the test file `addsub_to_int16_test`, change the data types definitions for `u1` and `u2` as follows:

```
u1 = uint16(10)
u2 = uint16(10)
```

In the MATLAB Coder app:

- Open the project that contains the `addsub_to_int16` function.
- Use the updated test file `addsub_to_int16_test` to automatically redefine the input types.
- Change the addition and subtraction algorithm setting. Open the code replacement library `crl_table_addsub`. Change the entry **Algorithm** parameter setting from `Cast before operation` (`RTW_CAST_BEFORE_OP`) to `Cast after operation` (`RTW_CAST_AFTER_OP`).
- Regenerate code and a report.

9 From the Code Replacements Report, open the Code Replacement Viewer. Use the Code Replacement Viewer trace information to verify that your MATLAB code or code replacement table entry corrects the code replacement issue. The trace information shows a hit for function `addsub_two_int16`.

See Also

More About

- "Verify Code Replacement Library" on page 73-43

Deploy Code Replacement Library

Package and deploy your code replacement library.

1. Move your code replacement library customization and registration files to a location on the MATLAB® search path that is accessible to other users. If another registration file already exists in that location, copy and paste the information for your code replacement library into the existing registration file.
2. Refresh your session and verify that the library is available as a configuration option for the code generator.
3. Verify that code replacement occurs as expected in generated code.
4. Inform users that the code replacement library is available.

See Also

More About

- “Verify Code Replacement Library” on page 73-43
- “Package Code for Other Development Environments”
- “Develop a Code Replacement Library” on page 73-13
- “What Is Code Replacement Customization?” on page 73-3

Math Function Code Replacement

This example shows how to develop a code replacement library to optimize the performance of math functions by providing information on how to define code replacement for a `sin` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_sin`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

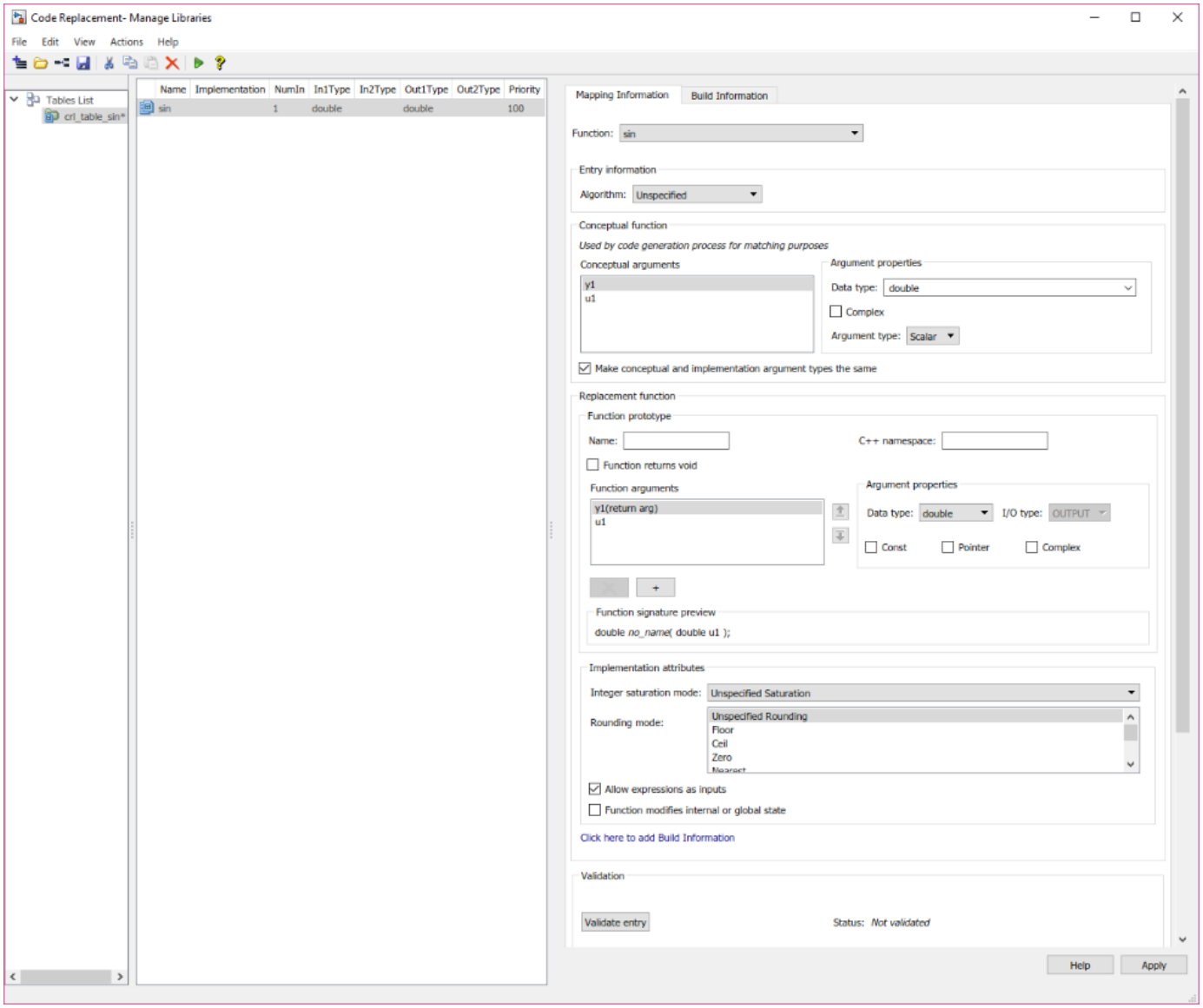
4. Create entry parameters. In the **Function** drop-down list, select `sin`. Algorithm information appears in the crtool. For this example, leave the parameter unspecified.

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input argument, u1, with the **Data Type** of double and the **Argument Type** of Scalar.

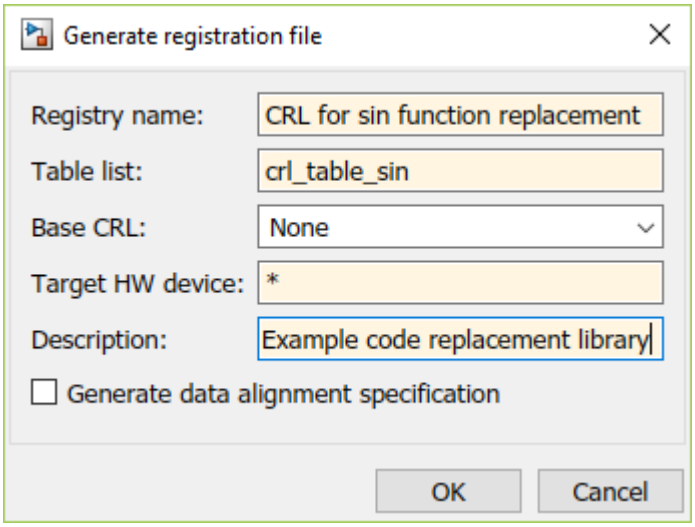
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_sin
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_sin
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
        'DataTypeMode',           'double');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'double');
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_sin()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
        'Key',                          'sin', ...
        'Priority',                     30, ...
        'ImplementationName',           'mySin', ...
        'ImplementationHeaderFile',     'basicMath.h', ...
        'ImplementationSourceFile',     'basicMath.c');

%% Create the conceptual representation
hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'y1',...
        'IOType',                 'RTW_IO_OUTPUT',...
        'DataTypeMode',           'double');

hEntry.createAndAddConceptualArg(...
        'RTW.TflArgNumeric',      ...
        'Name',                   'u1', ...
        'IOType',                 'RTW_IO_INPUT',...
        'DataTypeMode',           'double');

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_sin
```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for sin function replacement’;
this(1).TableList = {'crl_table_sin.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';
```

```
end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Memory Function Code Replacement” on page 73-57

Memory Function Code Replacement

This example shows how to develop a code replacement library to optimize the performance of memory functions by providing information on how to define code replacement for a `memcpy` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_memcpy`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `memcpy`.

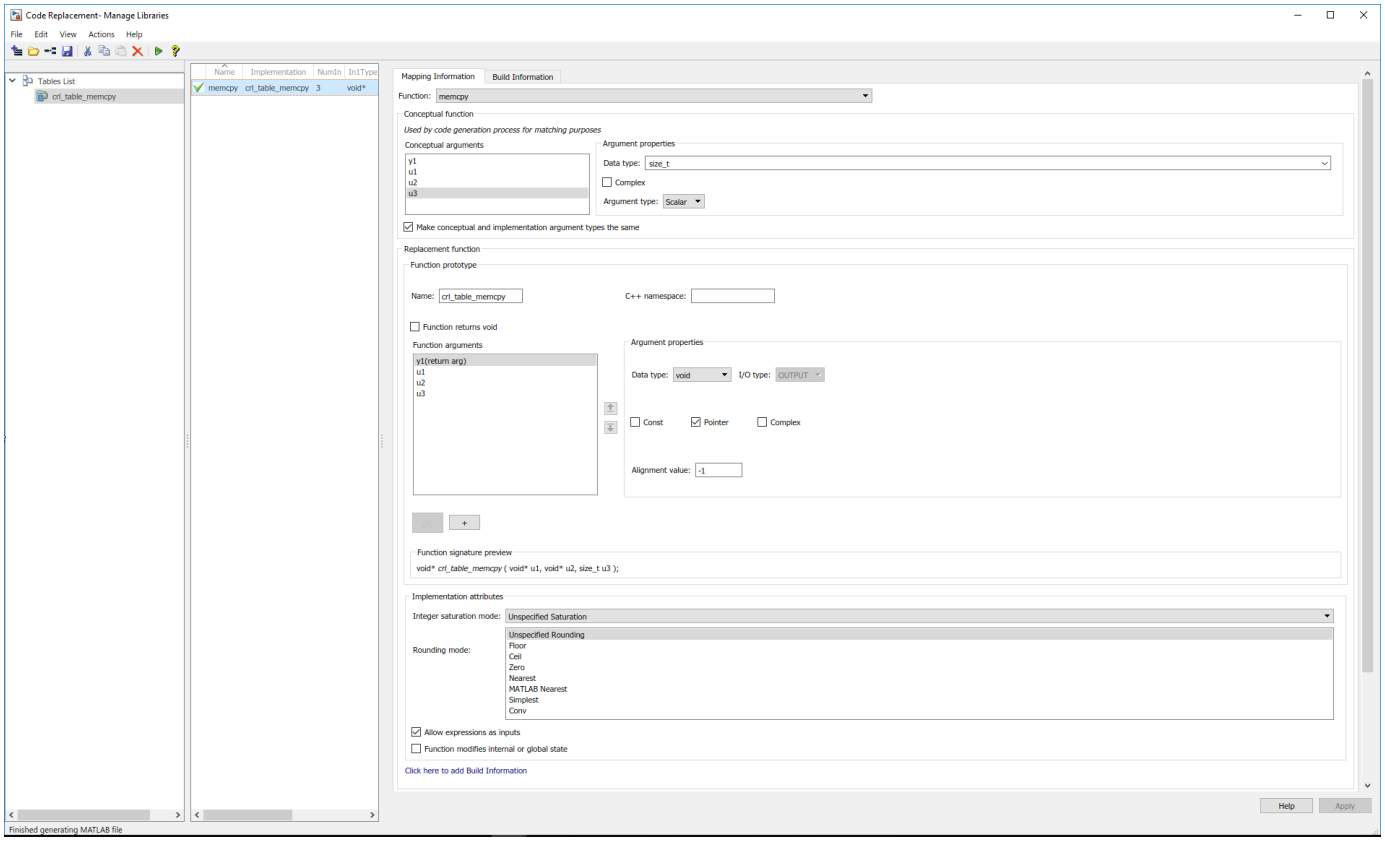
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, with the **Data Type** of void and the **Argument Type** of Pointer, the input arguments, `u1` and `u2` with the **Data Type** of void and the **Argument Type** of Pointer and `u3` with the **Data Type** of unsigned integral and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

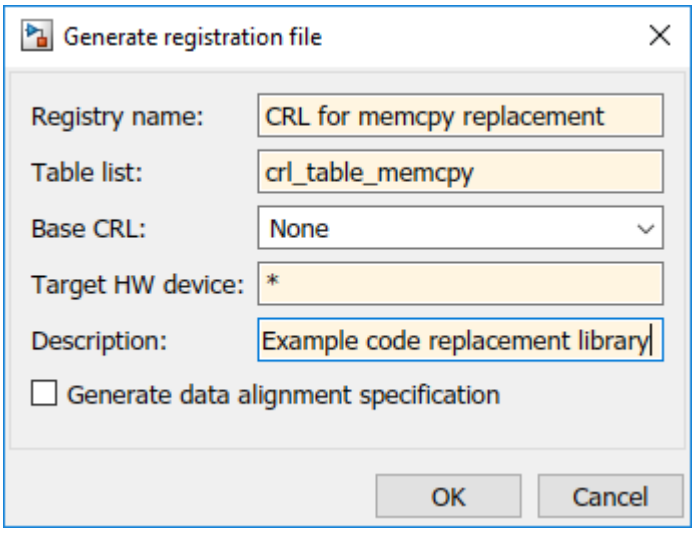
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_memcpy()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;


%% Create entry parameters
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
hEntry.setTflCFunctionEntryParameters(...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;


%% Create entry parameters
% Set SideEffects to 'true' for function returning void to prevent it from
% being optimized away.
hEntry.setTflCFunctionEntryParameters(...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);

%% Create conceptual arguments
arg = getTflArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u2', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_memcpy()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
                              'Key',                      'memcpy', ...
                              'Priority',                 90, ...
                              'ImplementationName',       'memcpy_int', ...
                              'ImplementationHeaderFile', 'memcpy_int.h',...
                              'SideEffects',              true);


%% Create conceptual arguments
arg = getTflArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u2', 'void*');
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(hEntry, arg);
```

```
%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_memcpy
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for memory function replacement’;
   this(1).TableList = {'crl_table_memcpy.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Math Function Code Replacement” on page 73-51

Specify In-Place Code Replacement

In-place code replacement is an optimization technique that uses a single buffer, that is, the same memory, to store function input and output data, as in `x=foo(x)`.

You can develop a code replacement library to optimize in-place function argument replacement . To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_inplace`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

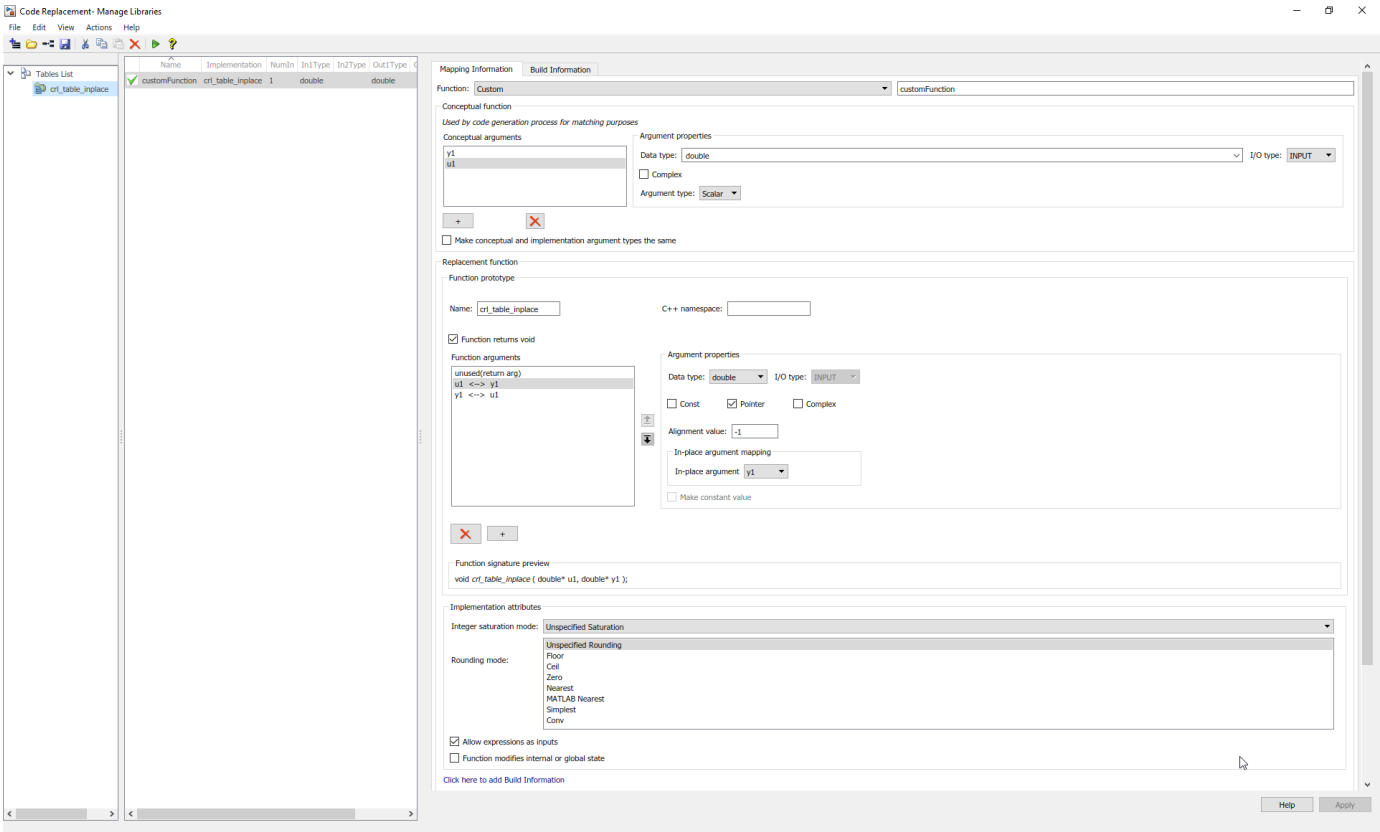
4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `customFunction` .

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input argument, u1, with the **Data Type** of double and the **Argument Type** of **Pointer**.

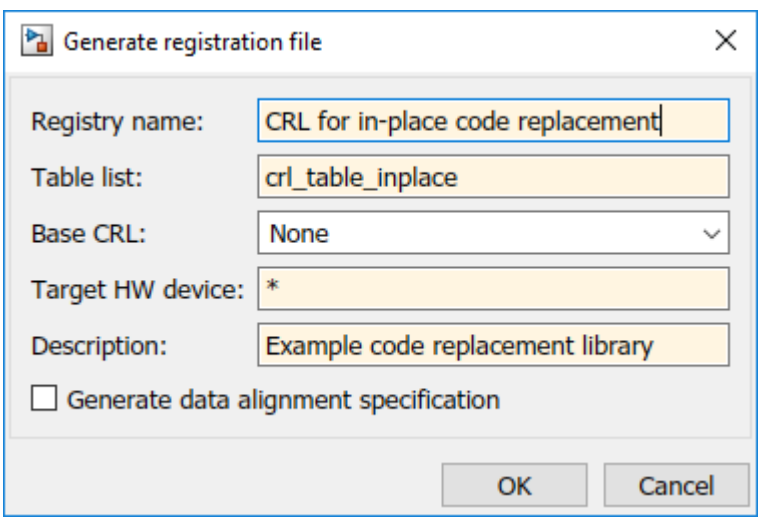
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. Add the implementation parameter `y1` and `u1`. For each input argument that you want to specify as in-place with a corresponding output argument, in the **Argument** properties box, select the **Pointer** check box. For the implementation parameter `y1`in the **Argument properties** section of the dialog box, set the **In-place argument** to `u1`.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For

more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_inplace()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_table_inplace
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = rtw.TflCFunctionEntry;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_inplace()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
       'Key', 'customFunction', ...
       'Priority', 100, ...
       'ImplementationName', 'custom_function_inplace_impl', ...
       'SideEffects', true);
```

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString` and `addConceptualArg`.

```
function hTable = crl_table_inplace()
% Create a code replacement library table
```

```
%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
       'Key', 'customFunction', ...
       'Priority', 100, ...
       'ImplementationName', 'custom_function_inplace_impl', ...
       'SideEffects', true);

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hEntry, 'u1','double');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. This example uses calls to the `getTflArgFromString` function to create implementation arguments that map to arguments in the replacement function prototype: output argument *y1* and input argument *u1*. For each argument, the example uses the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument. For each argument, this example adds the argument to the entry array of implementation arguments.

```
function hTable = crl_table_inplace()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters(...
       'Key', 'customFunction', ...
       'Priority', 100, ...
       'ImplementationName', 'custom_function_inplace_impl', ...
       'SideEffects', true);

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

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

%% Create the Implementation Representation

arg = getTflArgFromString(hEntry, 'y2','void');
arg.IOType = 'RTW_IO_OUTPUT';
```

```
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hEntry, 'u1','double*');
arg.ArgumentForInPlaceUse = 'y1';
hEntry.Implementation.addArgument(arg);

%% Add the entry to the table

hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_inplace
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for inplace function replacement';
this(1).TableList = {'crl_table_inplace.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Math Function Code Replacement” on page 73-51

Data Alignment for Code Replacement

Code replacement libraries can align data objects passed into a replacement function to a specified boundary.

Code Replacement Data Alignment

You can take advantage of function implementations that require aligned data to optimize application performance when using MATLAB Coder. To configure data alignment for a function implementation:

1. Specify the data alignment requirements in a code replacement entry. Specify alignment separately for each implementation function argument or collectively for all function arguments. See “Specify Data Alignment Requirements for Function Arguments” on page 72-105.
2. Specify the data alignment capabilities and syntax for one or more compilers. Include the alignment specifications in a library registration entry in the `rtwTargetInfo.m` file. See “Provide Data Alignment Specifications for Compilers” on page 72-106.
3. Register the library containing the table entry and alignment specification object.
4. Configure the code generator to use the code replacement library and generate code. Observe the results.

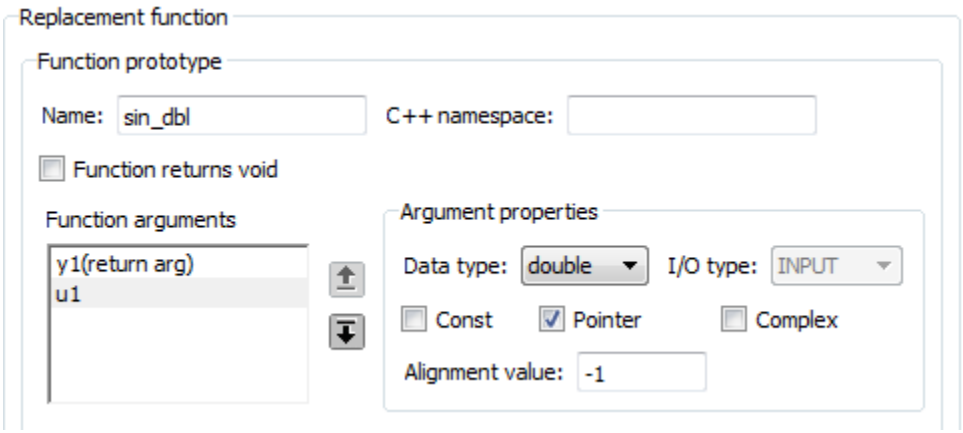
For examples, see “Basic Example of Code Replacement Data Alignment” on page 72-109 and the “Data Alignment for Function Implementations” section of the “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31 example page.

Note If replacement that requires alignment uses imported data (for example, I/O of an entry-point function or exported function), specify the data alignment with `coder.dataAlignment` statements in the MATLAB code. Specify alignment separately for each instance of imported data. See “Specify Data Alignment in MATLAB Code for Imported Data” on page 73-72.

Specify Data Alignment Requirements for Function Arguments

To specify the data alignment requirement for an argument in a code replacement entry:

- If you are defining a replacement function in a code replacement table registration file, create an argument descriptor object (`RTW.ArgumentDescriptor`). Use its `AlignmentBoundary` property to specify the required alignment boundary and assign the object to the argument `Descriptor` property.
- If you are defining a replacement function using the **Code Replacement Tool**, on the **Mapping Information** tab, in the **Argument properties** section for the replacement function, enter a value for the **Alignment value** parameter.



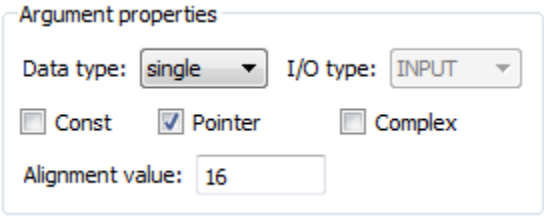
The `AlignmentBoundary` property (or **Alignment value** parameter) specifies the alignment boundary for data passed to a function argument, in number of bytes. The `AlignmentBoundary` property is valid only for addressable objects, including matrix and pointer arguments. It is not applicable for value arguments. Valid values are:

- `-1` (default) — If the data is a `Simulink.Bus`, `Simulink.Signal`, or `Simulink.Parameter` object, specifies that the code generator determines an optimal alignment based on usage. Otherwise, specifies that there is not an alignment requirement for this argument.
- Positive integer that is a power of 2 — Specifies number of bytes in the boundary. The starting memory address for the data allocated for the function argument is a multiple of the specified value. If you specify an alignment boundary that is less than the natural alignment of the argument data type, the alignment directive is emitted in the generated code. However, the target compiler ignores the directive.

The following code specifies the `AlignmentBoundary` for an argument as 16 bytes.

```
hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
arg = getTflArgFromString(hLib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);
```

The equivalent alignment boundary specification in the Code Replacement Tool dialog box is in this figure.



Note If your model imports `Simulink.Bus`, `Simulink.Parameter`, or `Simulink.Signal` objects, specify an alignment boundary in the object properties, using the **Alignment** property. For more information, see `Simulink.Bus`, `Simulink.Parameter`, and `Simulink.Signal`.

Provide Data Alignment Specifications for Compilers

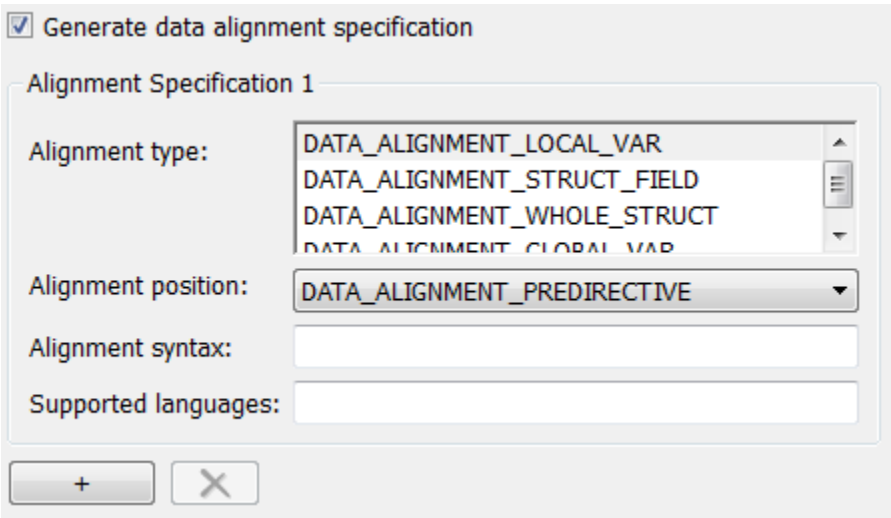
To support data alignment in generated code, describe the data alignment capabilities and syntax for your compilers in the code replacement library registration. Provide one or more alignment specifications for each compiler in a library registry entry.

To describe the data alignment capabilities and syntax for a compiler:

- If you are defining a code replacement library registration entry in a `rtwTargetInfo.m` customization file, add one or more `AlignmentSpecification` objects to an `RTW.DataAlignment` object. Attach the `RTW.DataAlignment` object to the `TargetCharacteristics` object of the registry entry.

  The `RTW.DataAlignment` object also has the property `DefaultMallocAlignment`, which specifies the default alignment boundary, in bytes, that the compiler uses for dynamically allocated memory. If the code generator uses dynamic memory allocation for a data object involved in a code replacement, this value determines if the memory satisfies the alignment requirement of the replacement. If not, the code generator does not use the replacement. The default value for `DefaultMallocAlignment` is `-1`, indicating that the default alignment boundary used for dynamically allocated memory is unknown. In this case, the code generator uses the natural alignment of the data type to determine whether to allow a replacement.

  Additionally, you can specify the alignment boundary for complex types by using the `addComplexTypeAlignment` function.
- If you are generating a customization file function using the Code Replacement Tool, fill out the following fields for each compiler.



  Click the plus (+) symbol to add additional compiler specifications.

For each data alignment specification, provide the following information.

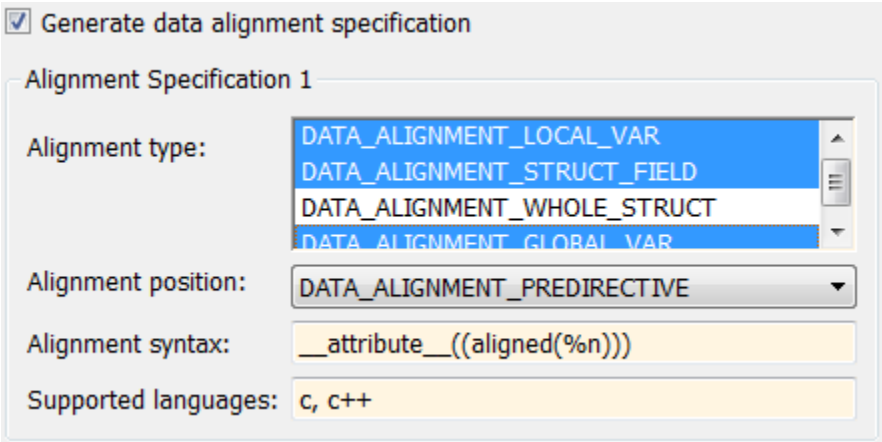
Alignment-Specification Property	Dialog Box Parameter	Description
`AlignmentType`	**Alignment type**	Cell array of predefined enumerated strings, specifying which types of alignment this specification supports. • `DATA_ALIGNMENT_LOCAL_VAR` — Local variables. • `DATA_ALIGNMENT_GLOBAL_VAR` — Global variables. • `DATA_ALIGNMENT_STRUCT_FIELD` — Individual structure fields. • `DATA_ALIGNMENT_WHOLE_STRUCT` — Whole structure, with padding (individual structure field alignment, if specified, is favored and takes precedence over whole structure alignment). Each alignment specification must specify at least `DATA_ALIGNMENT_GLOBAL_VAR` and `DATA_ALIGNMENT_STRUCT_FIELD`.
`AlignmentPosition`	**Alignment position**	Predefined enumerated string specifying the position in which you must place the compiler alignment directive for alignment type `DATA_ALIGNMENT_WHOLE_STRUCT`: • `DATA_ALIGNMENT_PREDIRECTIVE` — The alignment directive is emitted before `struct st_tag{…}`, as part of the type definition statement (for example, MSVC). • `DATA_ALIGNMENT_POSTDIRECTIVE` — The alignment directive is emitted after `struct st_tag{…}`, as part of the type definition statement (for example, gcc). • `DATA_ALIGNMENT_PRECEDING_STATEMENT` — The alignment directive is emitted as a standalone statement immediately preceding the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. • `DATA_ALIGNMENT_FOLLOWING_STATEMENT` — The alignment directive is emitted as a standalone statement immediately following the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. For alignment types other than `DATA_ALIGNMENT_WHOLE_STRUCT`, code generation uses alignment position `DATA_ALIGNMENT_PREDIRECTIVE`.

Alignment-Specification Property	Dialog Box Parameter	Description
`AlignmentSyntax-Template`	**Alignment syntax**	Specifies the alignment directive string that the compiler supports. The string is registered as a syntax template that has placeholders in it. These placeholders are supported: • `%n` — Replaced by the alignment boundary for the replacement function argument. • `%s` — Replaced by the aligned symbol, usually the identifier of a variable. For example, for the gcc compiler, you can specify `__attribute__((aligned(%n)))`, or for the MSVC compiler, `__declspec(align(%n))`.
`SupportedLanguages`	**Supported languages**	Cell array specifying the languages to which this alignment specification applies, among `c` and `c++`. Sometimes alignment syntax and position differ between languages for a compiler.

Here is a data alignment specification for the GCC compiler:

```
da = RTW.DataAlignment;

as = RTW.AlignmentSpecification;
as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                    'DATA_ALIGNMENT_STRUCT_FIELD', ...
                    'DATA_ALIGNMENT_GLOBAL_VAR'};
as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
as.AlignmentPosition = 'DATA_ALIGNMENT_PREDIRECTIVE';
as.SupportedLanguages = {'c', 'c++'};
da.addAlignmentSpecification(as);

tc = RTW.TargetCharacteristics;
tc.DataAlignment = da;
```

Here is the corresponding specification in the **Generate customization** dialog box of the Code Replacement Tool.



Specify Data Alignment in MATLAB Code for Imported Data

If MATLAB Coder code replacements that require data alignment use imported data, such as an entry-point or exported function I/O, specify data alignment to external code with `coder.dataAlignment` statements in the MATLAB code.

If MATLAB Coder code replacements occur that require data alignment (uses imported data), such as an entry-point or exported function with I/O, specify code replacement data alignment with `coder.DataAlignment` statements in the MATLAB code.

To specify the data alignment requirements for imported data in a MATLAB code:

- For each instance of imported data that requires data alignment, specify the alignment in the function with a `coder.dataAlignment` statement of the form:

  `coder.dataAlignment('varName', align_value)`
- The *`varName`* is a character array of the variable name that requires alignment information specification. The *`align_value`* is an integer number which should be a power of 2. This number specifies the power-of-2 byte alignment boundary.
- An example function that specifies data alignment is:

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

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

y = x1 + x2;

end
```

If `testFunction` is an entry-point or exported function, imported data `x1`, `x2`, and `y` are not aligned automatically by the code generator. The `coder.DataAlignment` statements for these variables are only meant as information for the code generator. The call sites allocating memory for the data need to ensure that the data is aligned as specified.

You also can specify code replacement data alignment for exported data, such as a `global` variable or an `ExportedGlobal` storage class. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75.

See Also

More About

- “Code You Can Replace From Simulink Models” on page 72-6
- “Define Code Replacement Library Optimizations” on page 72-36
- “Develop a Code Replacement Library” on page 72-20

Array Layout and Code Replacement

To improve execution speed of algorithms or simplify integration with external code or data, you can define code replacement mappings that apply a specific layout for storing array elements in memory. By default, the code generator stores array elements in a column-major layout. To change the layout to row-major, change the value of the entry parameter `ArrayLayout` to `ROW_MAJOR`.

The following table definition file for a sum operation uses the row-major array layout.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_rowmajor_matrix
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   hEnt = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. In the function call, set the parameter `ArrayLayout` to `ROW_MAJOR`

   ```
   setTflCOperationEntryParameters(hEnt, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 100, ...
                       'ArrayLayout',              'ROW_MAJOR', ...
                       'ImplementationName',        'MyMul_ROW', ...
                       'ImplementationHeaderFile', 'MyMul_ROW.h', ...
                       'ImplementationSourceFile', 'MyMul_ROW.c');
   ```

5. Create conceptual arguments y1, u1, and u2. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'double');
   arg.DimRange = [2 2; 50 50];
   hEnt.addConceptualArg(arg);

   arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'double');
   arg.DimRange = [2 3; 2 3];
   hEnt.addConceptualArg(arg);

   arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'double');
   arg.DimRange = [3 4; 3 4];
   hEnt.addConceptualArg(arg);
   ```

6. Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. When defining the implementation function return argument, create a `void` output argument. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Implementation Args
arg = hEnt.getTflArgFromString('unused','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

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

arg = hEnt.getTflArgFromString('u2','double*');
hEnt.Implementation.addArgument(arg);

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

arg = hEnt.getTflArgFromString('u3','uint32',2);
arg.Type.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u4','uint32',3);
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u5','uint32',3);
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u6','uint32',4);
hEnt.Implementation.addArgument(arg);
```

7. Add the entry to a code replacement table with a call to the `addEntry` function.

   ```
   addEntry(hTable, hEnt);
   ```

8. Save the table definition file. To name the file, use the name of the table definition function . Then, generate code and a code generation report and review the code replacements.

See Also

`coder.replace` | `setTflCFunctionEntryParameters` | `setTflCOperationEntryParameters`

More About

- “Row-Major and Column-Major Array Layouts”
- “Define Code Replacement Library Optimizations” on page 72-36

Allow Shape Agnostic Match

Code replacement shape-agnostic match allows matrix inputs for certain operations to be matched based on the number of elements as opposed to matrix shape. For example, a `6x1` matrix matches a `2x3` matrix in shape-agnostic match because both have a total of 6 elements. Such matches can only be made if the output and inputs are contiguously allocated in memory and if the replacement function can be performed element-wise. Shape-agnostic matrix match increases replacement rate so it is advantageous for replacement that does not require matrix shape to be considered.

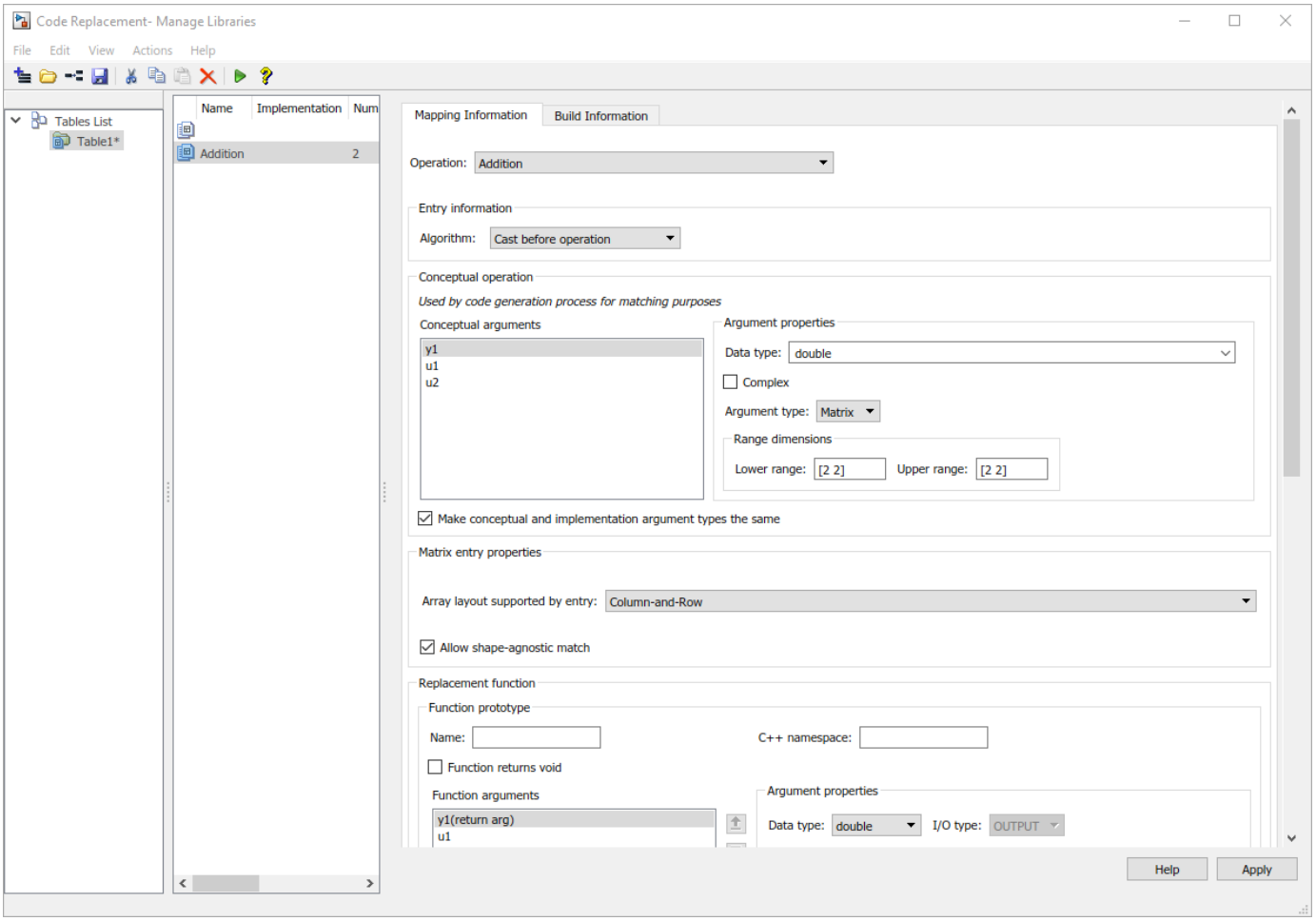
Shape-agnostic code replacement entries can be created either with the user interface of the **Code Replacement Tool** or programmatically.

User Interface Method- Code Replacement Tool

This example shows how to set up shape-agnostic matrix replacement for an addition code replacement table entry.

1. Open the Code Replacement Tool. From the MATLAB command line, enter `crtool`.
2. Create a code replacement table. Go to `File > New Table`.
3. Create a table entry. Shape-agnostic matrix replacement only supports **Math Operation** table entries. Either `right-click` on your table or go to `File > New Entry` and select **Math Operation**. Mapping information appears. From the `Operation` drop down menu, select `addition`.
4. Create conceptual arguments. The `Conceptual operation` section defines each of your conceptual arguments. To enable shape-agnostic replacement set the `Argument type:` to `matrix` to open the `Matrix entry properties` menu.
5. Set entry parameters. In the `Matrix entry properties` menu, select the `Allow shape-agnostic match` checkbox to enable shape-agnostic replacement.
6. Validate your table entry. Either click the `validate entry` button or right-click the entry and select `Validate entries`.
7. Click `save`.

   This display shows the **Code Replacement Tool** settings for this example.



Programmatic Method

This example shows how to set up a shape-agnostic matrix replacement for an addition code replacement table entry.

1. Create the code replacement table. Create a function definition with the table name `'ShapeAgnosticTable'`. Call `RTW.TflTable` to create the table.

   ```
   function hLib = ShapeAgnosticTable
   hLib = RTW.TflTable;
   ```

2. Create a table entry for code replacement. Shape-agnostic replacement only supports operation entries, as specified with a call to `RTW.TflCOperationEntry`.

   ```
   hEnt = RTW.TflCOperationEntry;
   ```

3. Set entry parameters to customize input processing. You can enable shape-agostic matrix replacement by setting the entry parameter `'AllowShapeAgnosticMatch'` to `true`.

   ```
   hEnt.setTflCOperationEntryParameters( ...
            'Key', 'RTW_OP_ADD', ...
            'Priority', 100, ...
            'AllowShapeAgnosticMatch', true, ...
   ```

```
            'ImplementationName', 'MyAdd_Matrix', ...
            'SideEffects', true);
```

4. Create conceptual arguments. For this example, create conceptual arguments y1, u1, and u2 with calls to the create function, `RTW.TflArgMatrix`, and add function, `addConceptualArg`.

```
arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'double');
arg.DimRange = [2 2; 50 50];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'double');
arg.DimRange = [2 3; 2 3];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'double');
arg.DimRange = [2 3; 2 3];
hEnt.addConceptualArg(arg);
```

5. Create implementation arguments. For this example, call the function `getTflArgFromString` to create the arguments. Define the implementation return argument as a `void` output argument. Define the conceptual output argument, y1, as a pointer with the IOType as `output`.

```
arg = hEnt.getTflArgFromString('unused','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

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

arg = hEnt.getTflArgFromString('u2','double*');
hEnt.Implementation.addArgument(arg);

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

arg = hEnt.getTflArgFromString('numElements','uint32',6);
hEnt.Implementation.addArgument(arg);
```

6. Add this entry to a code replacement table with the function `addEntry`.

```
hLib.addEntry( hEnt );
```

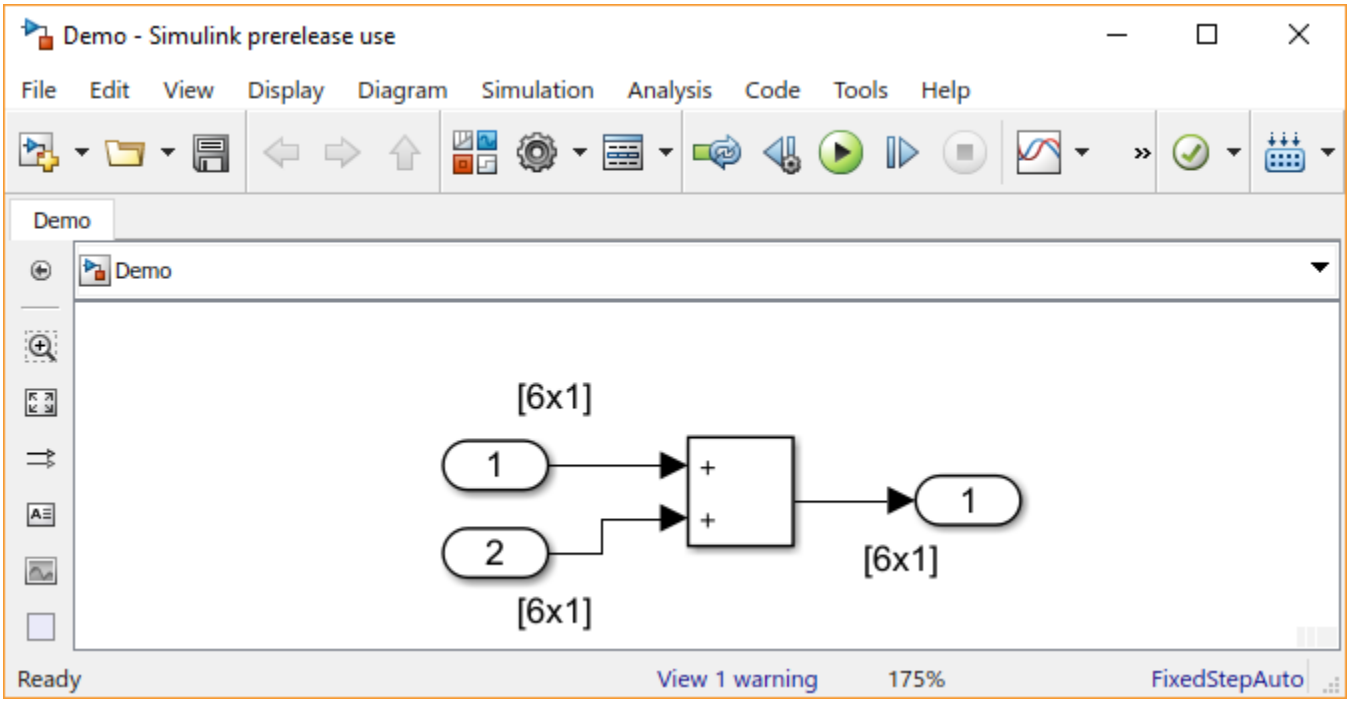
7. Save the table as its function name. For this example, `ShapeAgnosticTable`.

8. Validate your entry at the MATLAB command line. Invoke the table definition file with the following command.

```
hTbl = ShapeAgnosticTable
```

Example Model

An example of this code replacement table entry is demonstrated through the following model. Even though the matrix shape in this model does not match the entry, the operation can be performed element-wise and the entry enables shape-agnostic match, so code replacement is observed in the generated code.




```
/* Outport: '<Root>/Out1' incorporates:
  *  Inport: '<Root>/In1'
  *  Inport: '<Root>/In2'
  *  Sum: '<Root>/Add'
  */
 MyAdd_Matrix(Demo_U.In1, Demo_U.In2, Demo_Y.Out1, 6U);
```

Limitations

- If you enable shape-agnostic match for an operation that is inherently not computed element-wise, a regular match is performed.
- When shape-agnostic match is enabled, code replacement does not honor the setting for `'DimRange'`. Instead, it computes the number of elements and uses that value to match.
- If you are working with customized code replacement entries, adjust the `'do_match'` function to accept the total number of elements as opposed to matrix dimensions. For more information, see “Customize Match and Replacement Process” on page 72-132.

See Also

`setTflCOperationEntryParameters`

More About

- “What Is Code Replacement Customization?” on page 72-3
- “Define Code Replacement Library Optimizations” on page 73-23

Replace MATLAB Functions with Custom Code by Using the coder.replace Function

You can develop a code replacement library to optimize the performance of MATLAB functions containing a `coder.replace` function. To develop a code replacement library, use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_matlab_fcn_coder_replace`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `calculate`.

5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, and the input argument, u1, with the **Data Type** of double and the **Argument Type** of Scalar.

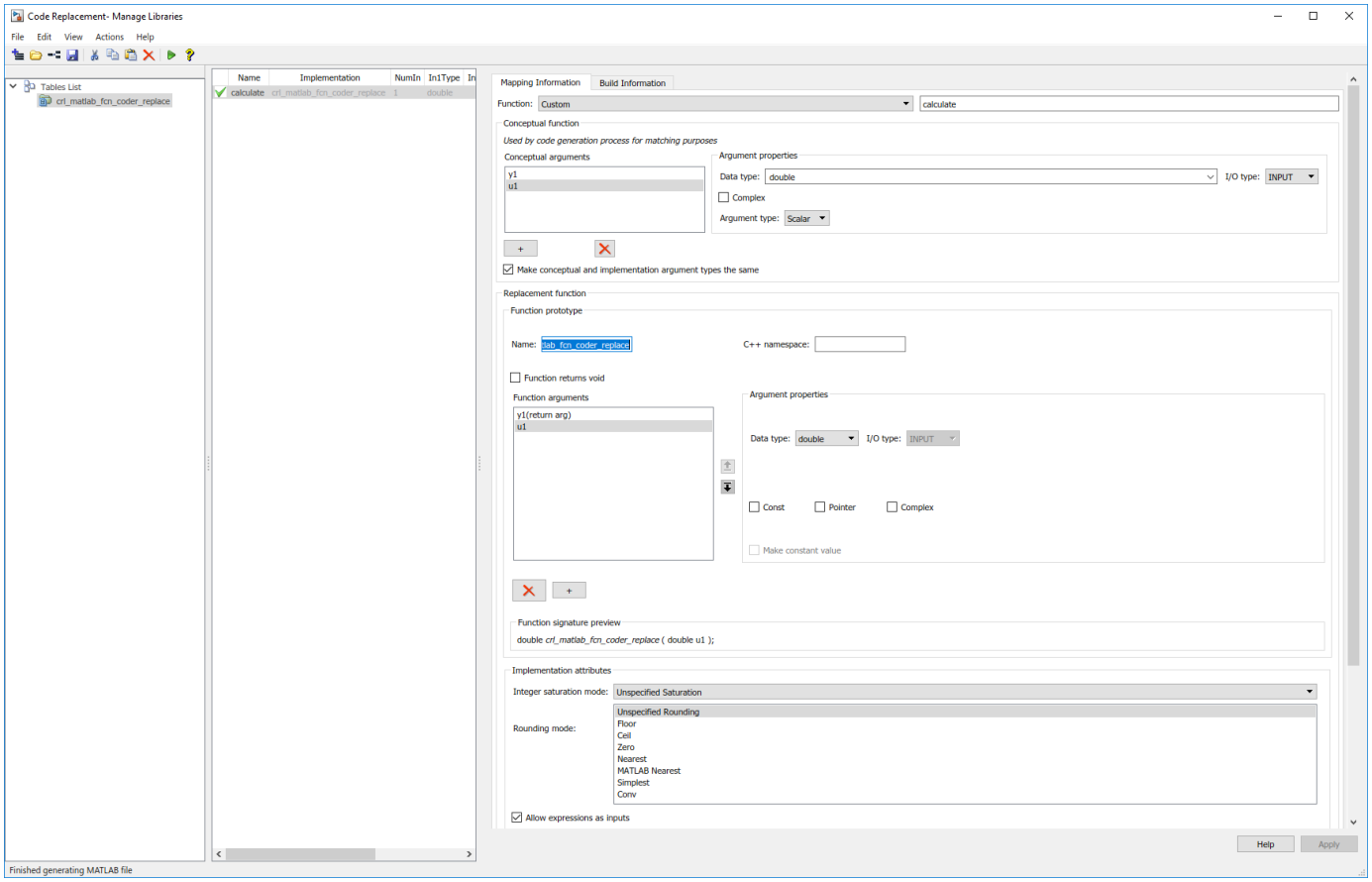
6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

   Add the return argument `y1` and the arguments `u1`.

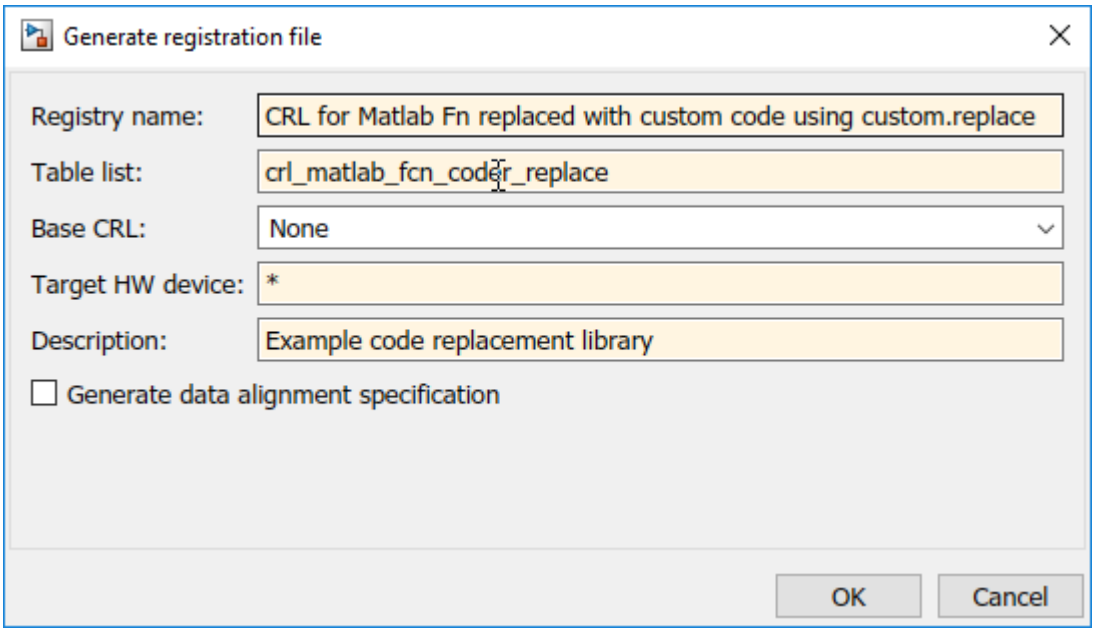
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
```

```
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hEntry, 'u1','double');
addConceptualArg(hEntry, arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable = crl_matlab_fcn_coder_replace
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
setTflCFunctionEntryParameters(hEntry, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
```

```
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c');

%% Create the conceptual representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

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

%% Create the Conceptual Representation
arg = getTflArgFromString(hEntry, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

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

%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_matlab_fcn_coder_replace
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for matlab function with custom code using coder.replace';
   this(1).TableList = {'crl_matlab_fcn_coder_replace.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Replace coder.ceval Calls to External Functions” on page 73-85

Replace coder.ceval Calls to External Functions

The `coder.ceval` function calls external C/C++ functions from code generated from MATLAB code. The code replacement software supports replacement of the function that you specify in a call to `coder.ceval`. An application of this code replacement scenario is to write generic MATLAB code that you can customize for different platforms with code replacements. A code replacement library can define hardware-specific code replacements for the function call. Use `coder.ceval` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Example Files

For the examples in “Interactively Develop a Code Replacement Library” on page 73-86 and “Programmatically Develop a Code Replacement Library” on page 72-121 you must have set up the following:

- Custom C function `my_add.c`.

```
/* my_add.c */

#include "my_add.h"

double my_add(double in1, double in2)
{
  return in1 + in2;
}
```

- Custom C header file `my_add.h`.

```
/* my_add.h */

double my_add(double in1, double in2);
```

- MATLAB function `call_my_add.m`, which uses `coder.ceval` to invoke `my_add.c`.

```
function y = call_my_add(in1, in2)  %#codegen

y=0.0;

if ~coder.target('Rtw')
% Executing in MATLAB, call MATLAB equivalent of C function my_add
  y= in1+in2;
else
% Executing in generated code, call C function my_add
  y = coder.ceval('my_add', in1, in2);
end
```

- MATLAB test function `call_my_add_test.m`, which calls `call_my_add.m`.

```
in1=10;
in2=20;
```

```
y = call_my_add(in1, in2);

disp('Output')
disp('y =')
disp(y);
```

- Replacement C function `my_add_replacement.c`.

```
/* my_add_replacement.c */

#include "my_add_replacement.h"

double my_add_replacement(double in1, double in2)
{
  return in1 + in2;
}
```

- Replacement C header file `my_add_replacement.h`.

```
/* my_add_replacement.h */

double my_add_replacement(double in1, double in2);
```

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_coder_cevals_calls`. Click **Apply**.

3. Create an entry. From the crtool menu, select **File** > **New entry** > **Function**.

4. Create entry parameters. In the **Function** drop-down list, select `custom`. Function information appears in the crtool. For this example, specify the parameter as `my_add` .

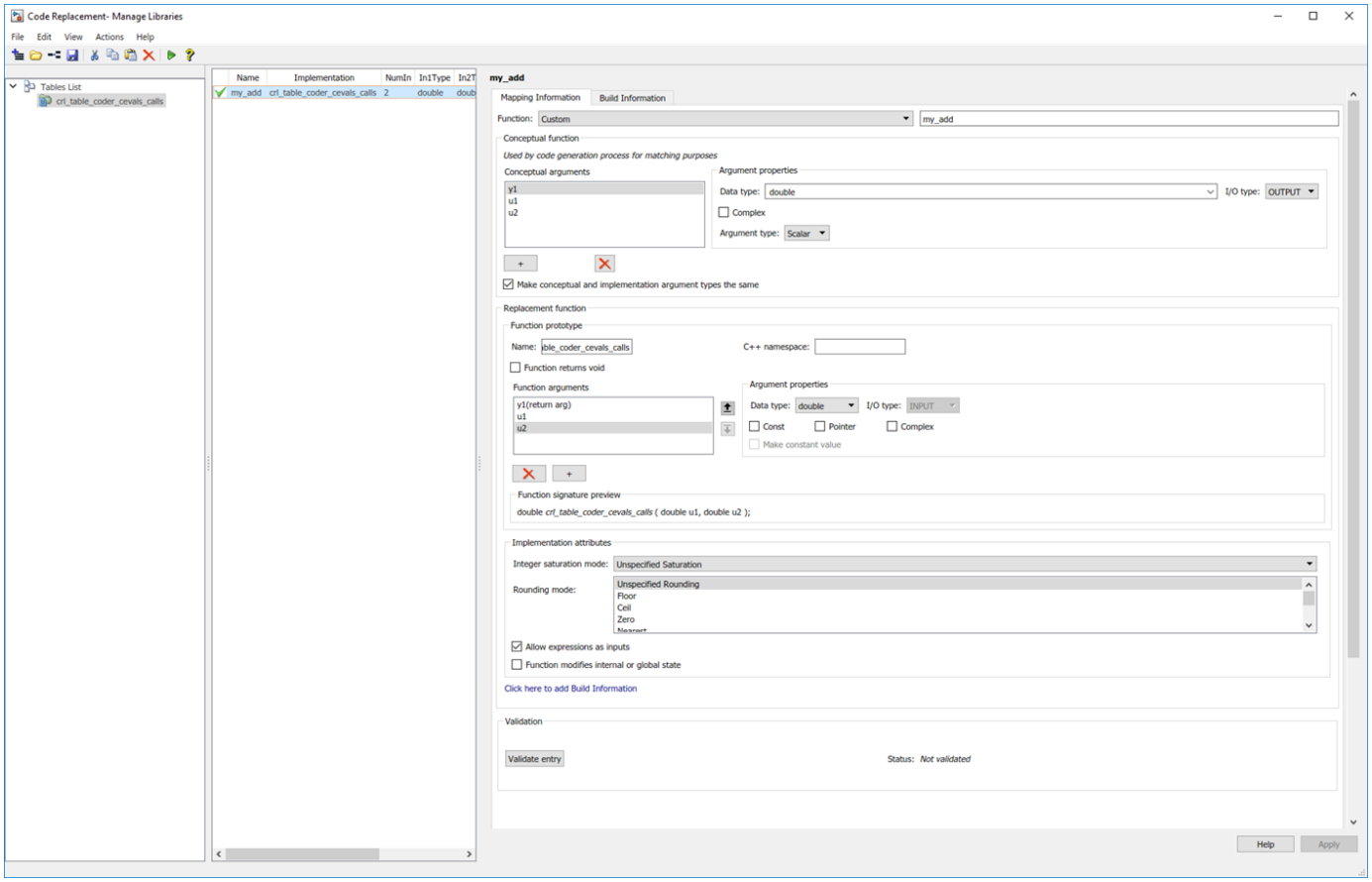
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1` and `u2` with the **Data Type** of double and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

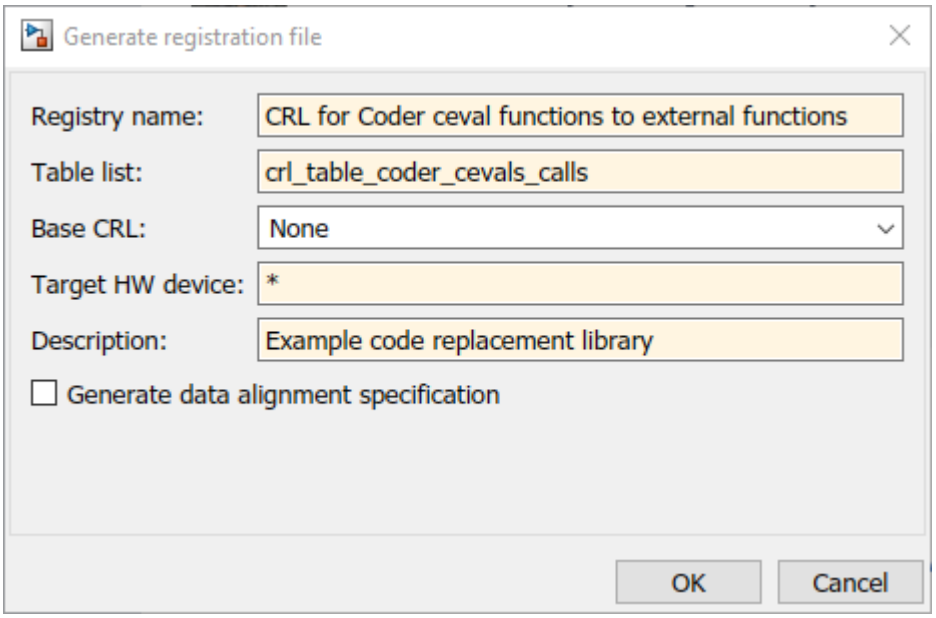
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, set the **Implementation header file** to `my_add_replacement.h`.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCFunctionEntry`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.addConceptualArg(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable =  crl_table_coder_cevals_calls()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCFunctionEntry;

%% Create entry parameters
hEntry.setTflCFunctionEntryParameters( ...
          'Key', 'my_add', ...
          'Priority', 100, ...
          'ImplementationName', 'my_add_replacement', ...
          'ImplementationHeaderFile', 'my_add_replacement.h', ...
          'ImplementationSourceFile', 'my_add_replacement.c');

%% Create the conceptual representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.addConceptualArg(arg);
```

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.addConceptualArg(arg);


%% Create the implementation representation
arg = hEntry.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

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

arg = hEntry.getTflArgFromString('u2','double');
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable =  crl_table_coder_cevals_calls
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for coder.ceval calls to external functions code replacement’;
   this(1).TableList = {' crl_table_coder_cevals_calls.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Replace MATLAB Functions with Custom Code by Using the coder.replace Function” on page 73-79

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`. For example:

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

In a code replacement table, the code generator registers each function implementation name defined by a table entry as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

To register additional code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Customize Match and Replacement Process” on page 73-93
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13

Customize Match and Replacement Process

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations replace default implementations.
- Preset replacement function signatures.

It is possible that preset match criteria and preset replacement function signatures do not completely meet your function and operator replacement needs. For example:

- You want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

To add extra logic into the code replacement match and replacement process, create custom code replacement table entries. With custom entries, you can specify additional match criteria and modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

1. Create a custom code replacement entry class, derived from `RTW.TflCFunctionEntryML (for function replacement) or RTW.TflCOperationEntryML` (for operator replacement).
2. In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations that instantiate the class:
   - Add match criteria that the base class does not provide. The base class provides a match based on:
     - Argument number
     - Argument name
     - Signedness
     - Word size
     - Slope (if not specified with wildcards)
     - Bias (if not specified with wildcards)
     - Math modes, such as saturation and rounding
     - Operator or function key
   - Modify the implementation signature by adding additional arguments or setting constant input argument values. You can inject a constant value, such as an input scaling value, as an additional argument to the replacement function.
3. Create code replacement entries that instantiate the custom entry class.
4. Register a library containing the code replacement table that includes your entries.

During code generation, the code replacement match process tries to match function or operator call sites with the base class of your derived entry class. If the process finds a match, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Customize Match and Replacement Process for Operators

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a scalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

For example:

- When fraction lengths are within a specific range, replace an operator with a fixed-point implementation function.
- When a match occurs, modify the replacement function signature based on compile-time information, such as passing fraction-length values into the function.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1. Create a class, for example `TflCustomOperationEntry`, that is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

   ```
   function ent = do_match(hThis, ...
           hCSO, ...
           targetBitPerChar, ...
           targetBitPerShort, ...
           targetBitPerInt, ...
           targetBitPerLong, ...
           targetBitPerLongLong)
   ```

   In the `do_match` signature:

   - `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
   - `hThis` is the handle to the class instance.
   - `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
   - Remaining arguments are the number of bits for various data types of the current target.

   The `do_match` method adds match criteria that the base class does not provide. The method makes modifications to the implementation signature. In this case, the `do_match` method relies on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to value 0. If the code generator finds a match, `do_match`:

   - Sets the return handle.
   - Removes slope and bias wild cards from the conceptual arguments (the match is for specific slope and bias values).
   - Writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

   You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this

class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = 0;
        end

        % Set the fraction-length values in the implementation function.
        ent.Implementation.Arguments(3).Value = ...
           -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
        ent.Implementation.Arguments(4).Value = ...
           -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
        ent.Implementation.Arguments(5).Value = ...
           -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
      end
    end
  end
end
```

Exit the class folder and return to the previous working folder.

2 Create and save the following code replacement table definition file, `crl_table_custom_add_ufix32.m`. This file defines a code replacement table that contains a

single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. The table entry:

- Instantiates the derived class `TflCustomOperationEntry` from the previous step. If you want to replace word sizes and signedness attributes, you can use the same derived class, but not the same entry, because you cannot use a wild card with the `WordLength` and `IsSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, add five other distinct entries. To use different implementation functions for saturation and rounding modes other than overflow and round to floor, add entries for those match permutations.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Calls the `createAndAddConceptualArg` function to create conceptual arguments `y1`, `u1`, and `u2`.
- Calls `createAndSetCImplementationReturn` and `createAndAddImplementationArg` to define the signature for the replacement function. Three of the calls to `createAndAddImplementationArg` create implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead, the `do_match` method of the derived class `TflCustomOperationEntry` can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.
- Calls `addEntry` to add the entry to a code replacement table.

```
function hTable = crl_table_custom_add_ufix32

hTable = RTW.TflTable;

% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
```

```
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

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

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_out', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
```

```
    'FractionLength', 0, ...
    'Value',       0);

addEntry(hTable, op_entry);
```

3 Check the validity of the operator entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_add_ufix32
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_add_ufix32)
  ```

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13

Scalar Operator Code Replacement

This example shows how to develop a code replacement library to optimize the performance of scalar operators by providing information on how to define code replacement for a `multiply` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_scalar`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Multiply`.

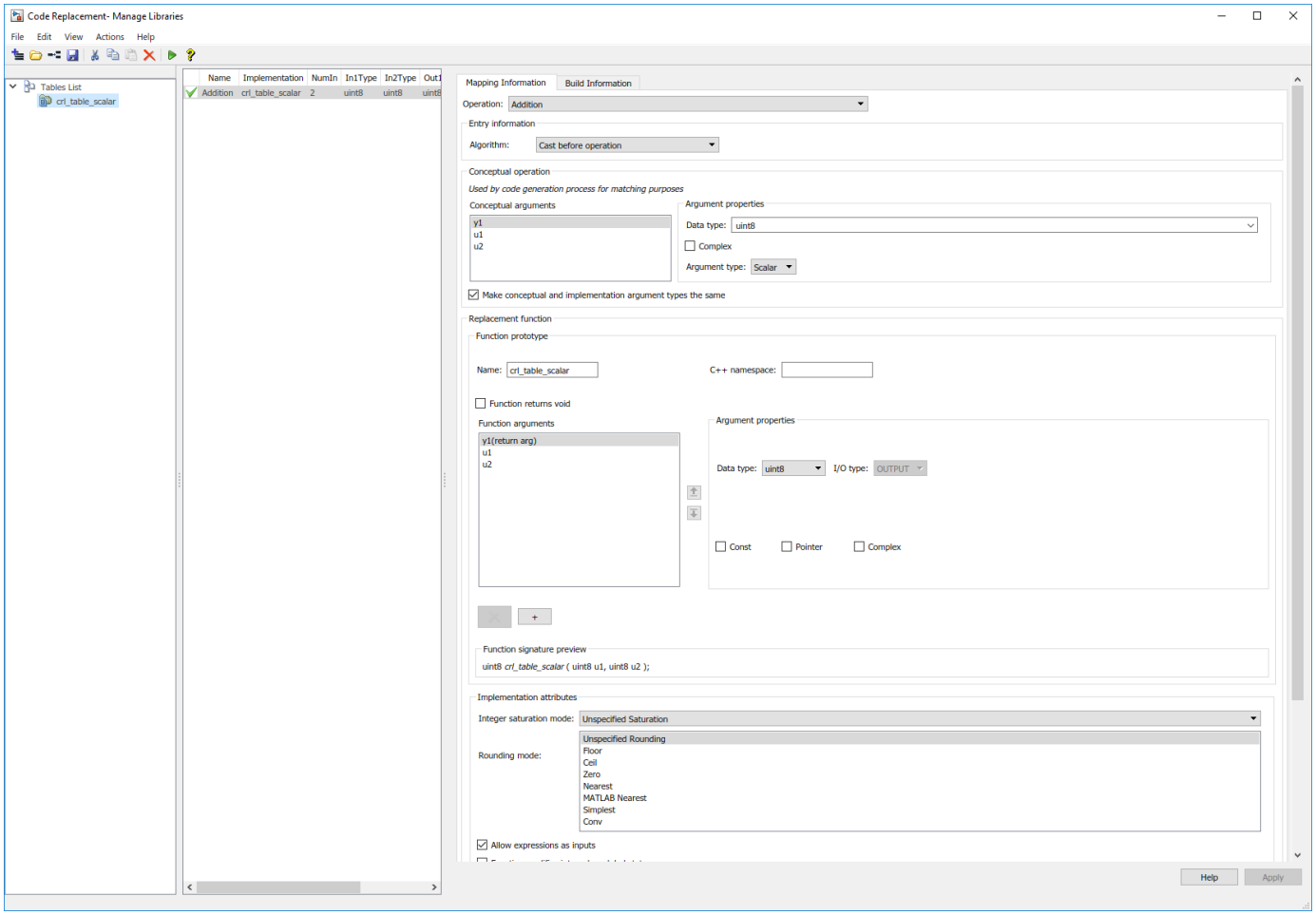
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1` and `u2` with the **Data Type** of `uint8` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

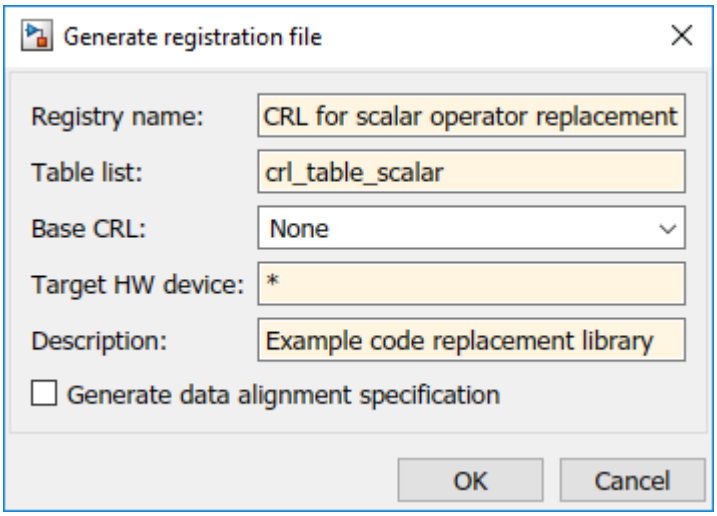
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_scalar()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function .

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(hEntry, arg );
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_scalar()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_MUL', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                    'ImplementationName',       'u8_mul_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_mul_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_mul_u8_u8.c' );

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(hEntry, arg );

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_scalar
   ```

9. Create a registration file for the code replacement library with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for scalar operator replacement';
   this(1).TableList = {'crl_table_scalar.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Addition and Subtraction Operator Code Replacement” on page 73-104

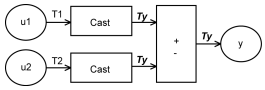
Addition and Subtraction Operator Code Replacement

This example shows how to develop a code replacement library to optimize the performance of addition and subtraction operators by providing information on how to define code replacement for a `addition` operator. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

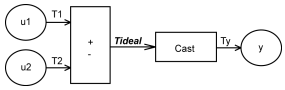
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- Cast-before-operation (CBO), default — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.



- Cast-after-operation (CAO) — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operand to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.



Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`double`	`double`	`double`	CBO, CAO
`double`	`double`	`double`	`single`	—
`double`	`double`	`single`	`double`	—
`double`	`double`	`single`	`single`	CBO
`double`	`single`	`double`	`double`	CBO, CAO
`double`	`single`	`double`	`single`	—

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`single`	`single`	`double`	—
`double`	`single`	`single`	`single`	CBO
`single`	`single`	`single`	`single`	CBO, CAO
`single`	`single`	`single`	`double`	—
`single`	`single`	`double`	`single`	—
`single`	`single`	`double`	`double`	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: • Operation type is `double`. • Operation type is `single` and input types are `single` or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: • Accumulator type equals output type (`Tacc == Tout`). • Output type is a superset of input types (`Tacc >= {Tin1, Tin2}`) and accumulator type is a superset of output type (`Tacc >= Tout`). • Operation does not incur range or precision loss.
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see “Addition” or “Subtraction” in “Fixed-Point Operator Code Replacement” on page 73-134 (for MATLAB code) or “Fixed-Point Operator Code Replacement” on page 72-188 (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 –> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

b In the right pane, name the table `crl_table_addition`. Click **Apply**.

3 Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4 Create entry parameters. In the **Function** drop-down list, select `Addition`. Algorithm information appears in the crtool. For this example, select the parameter to `Cast before operation`.

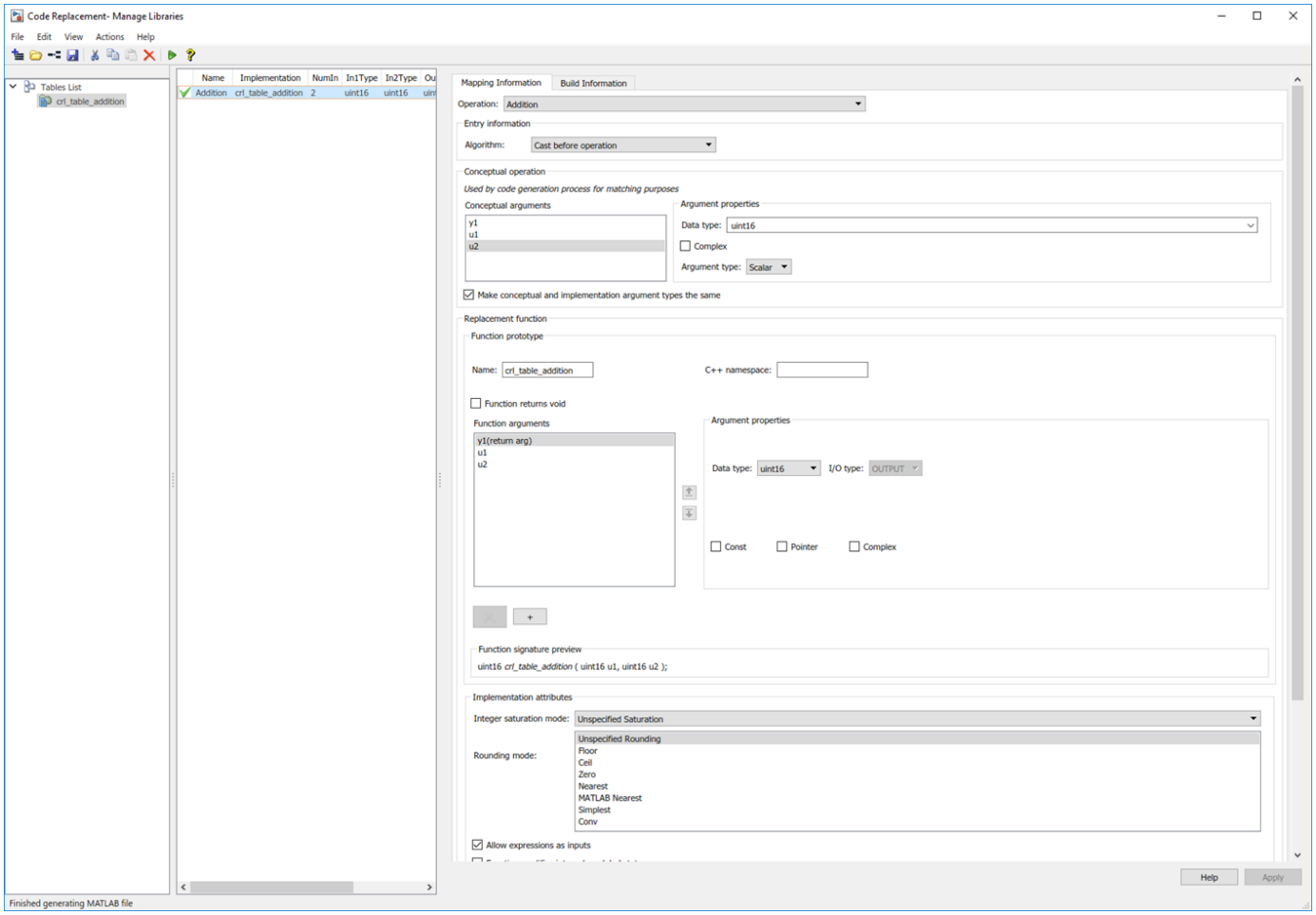
5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, y1, with the **Data Type** of `uint8` and the **Argument Type** of Scalar, the input arguments, u1 and u2 with the **Data Type** of `uint16` and the **Argument Type** of Scalar.

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

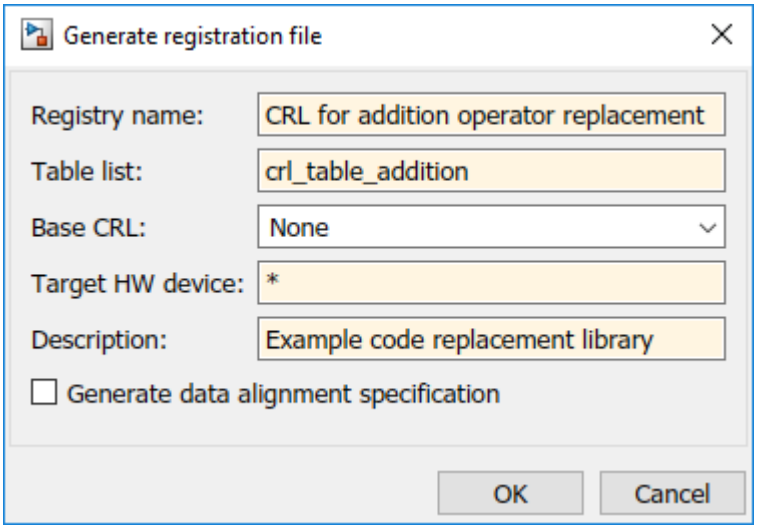
   Specify a **Name** for the replacement function under **Function prototype**.

7 Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8 Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                   'Key',                      'RTW_OP_ADD', ...
                   'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                   'ImplementationName',       'u8_add_cao_u16_u16');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTflTable = make_cbo_add_crl_table
% Create a function to call the code replacement library table
```

```
%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_ADD', ...
                    'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                    'ImplementationName',       'u8_add_cao_u16_u16');

%% Create the conceptual representation
arg = getTflArgFromString(hTflTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTflTable, 'u1', 'uint16');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTflTable, 'u2', 'uint16');
addConceptualArg(hEntry, arg );
```

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `copyConceptualArgsToImplementation`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTflTable = make_cbo_add_crl_table()
% Create a function to call the code replacement library table

%% Create a table object
hTflTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;


%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
                    'Key',                      'RTW_OP_ADD', ...
                    'EntryInfoAlgorithm',       'RTW_CAST_BEFORE_OP', ...
                    'ImplementationName',       'u8_add_cao_u16_u16');

%% Create the conceptual representation
arg = getTflArgFromString(hTflTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);

arg = getTflArgFromString(hTflTable, 'u1', 'uint16');
addConceptualArg(hEntry, arg );

arg = getTflArgFromString(hTflTable, 'u2', 'uint16');
addConceptualArg(hEntry, arg );

%% Create the Implementation Representation
copyConceptualArgsToImplementation(hEntry);

%% Add the entry to the table
addEntry(hTflTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = make_cbo_add_crl_table
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for addition subtraction operator replacement';
this(1).TableList = {'make_cbo_add_crl_table.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Limitations

- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.
- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A)(cast B)(u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (`+`) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Scalar Operator Code Replacement” on page 73-99

Small Matrix Operation to Processor Code Replacement

This example shows how to develop a code replacement library to optimize the performance of non-scalar small matrix operations with processor-specific intrinsic functions by providing information on how to define code replacement for a `addition` operator. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_small_matrix`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Function** drop-down list, select `Addition`. Algorithm information appears in the crtool. For this example, select the parameter to `Cast before operation`.

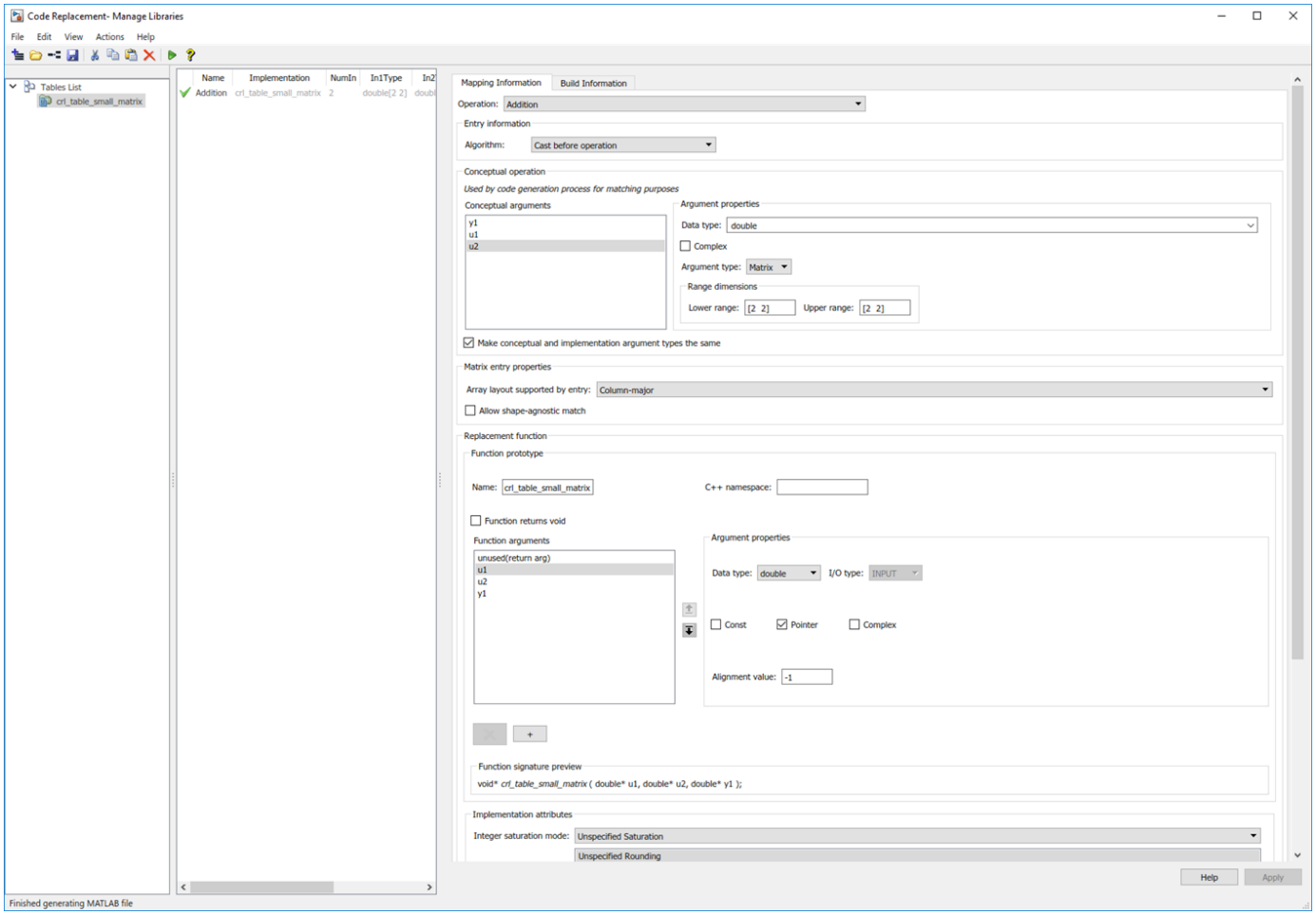
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, with the **Data Type** of `double` and the **Argument Type** of `Matrix`, the input argument, `u1` with the **Data Type** of `double` and the **Argument Type** of `Matrix` and the input argument, `u2` with the **Data Type** of `double` and the **Argument Type** of `Matrix`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Specify the return argument `y1, u1` and `u2`, with the **Data Type** of `double` and select the **Pointer** dialog box.

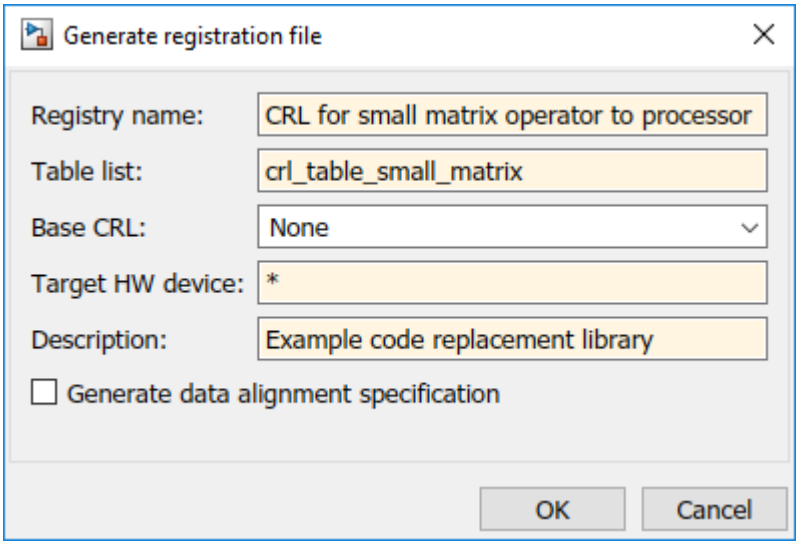
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File** > **Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
```

```
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);


%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`.

```
function hTable = crl_table_small_matrix_operator()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

```
%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
     'Key',                      'RTW_OP_ADD', ...
     'Priority',                 30, ...
     'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
     'ImplementationName',       'matrix_sum_2x2_double', ...
     'ImplementationHeaderFile', 'MatrixMath.h', ...
     'ImplementationSourceFile', 'MatrixMath.c', ...
     'GenCallback',              'RTW.copyFileToBuildDir', ...
     'SideEffects',              true);


%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);


%% Create the Implementation Representation
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);

arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
hTable.addEntry(hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_small_matrix
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for small matrix operator to processor';
this(1).TableList = {'crl_table_small_matrix.m'};
% table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Scalar Operator Code Replacement” on page 73-99

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

You can develop a code replacement library for floating-point matrix/matrix and matrix/vector multiplication operations with the multiplication functions `dgemm` and `dgemv` defined in the MathWorks BLAS library. If you use a third-party BLAS library for replacement, you must change the build requirements in this example to point to your library. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_blas`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Blas Operation (Fortran)**.

4. Create entry parameters. In the **Function** drop-down list, select `Multiply`.

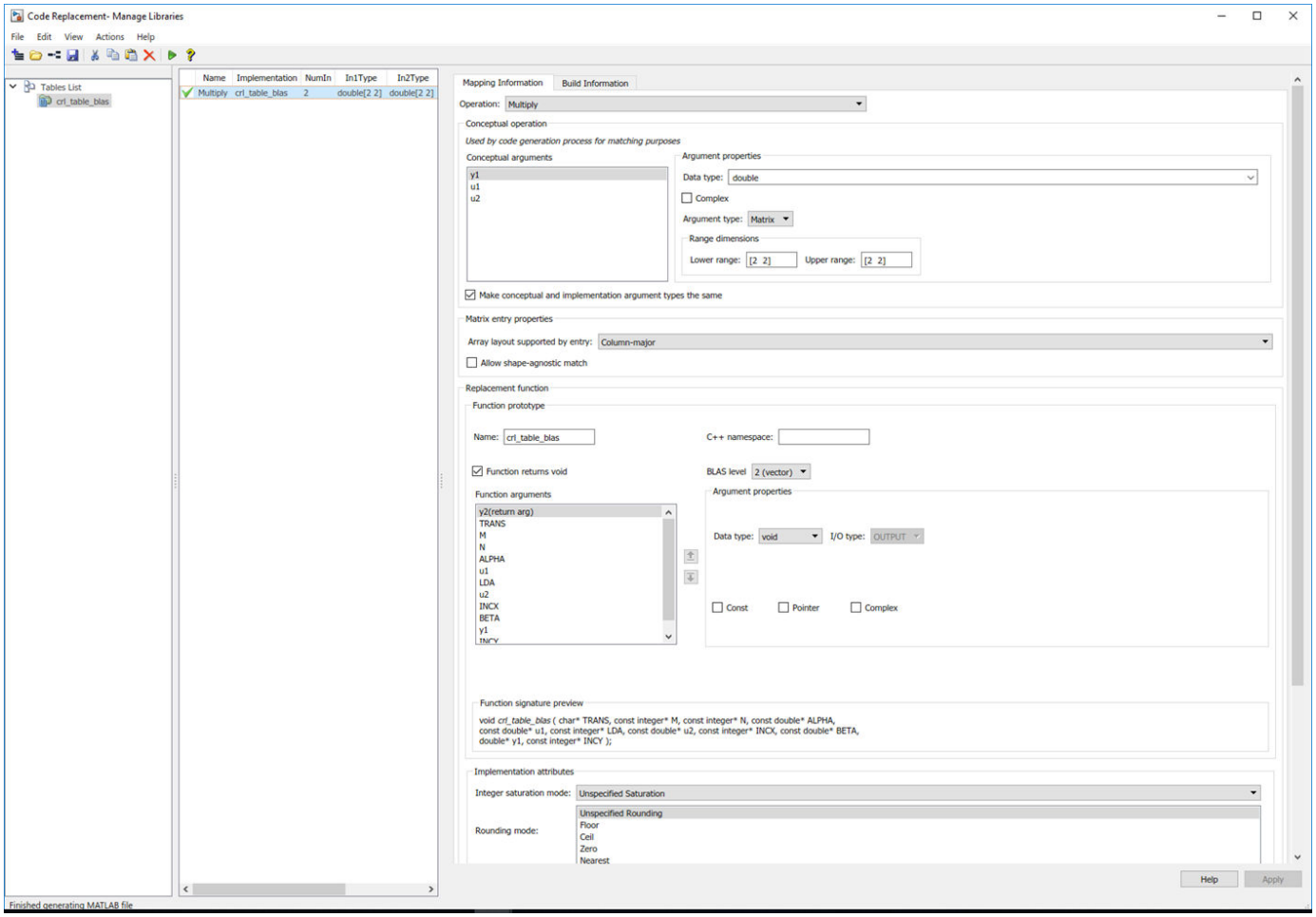
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of double and the **Argument Type** of Matrix.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Set the **BLAS level** to `2(Vector)`.

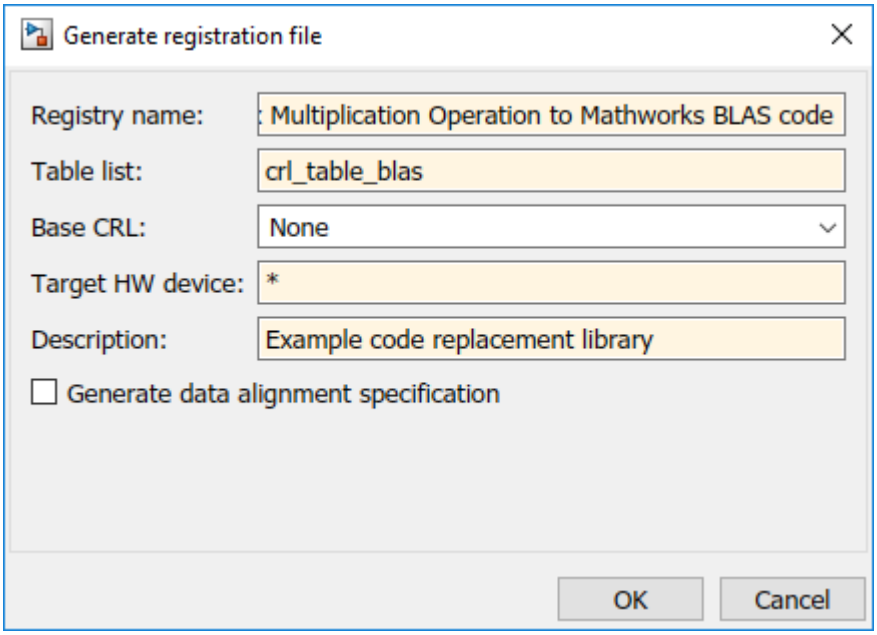
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_blas
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflBlasEntryGenerator`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
```

```
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, [2 2; inf inf] means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the dgemm32 entry for matrix/matrix multiplication replacement specifies dimensions [2 2; inf inf], while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions [2 1; inf 1].

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
```

```
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'u1', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'u2', ...
                            'BaseType',     'double', ...
                            'DimRange',     [1 1; inf inf]);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_blas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path for Windows or UNIX block%%%%%%%%%%%%%%%%%%

arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end


if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


%%%%%%Beginning of entries for the first dgemm block%%%%%%%%%%%%%%%%%%


% Create table entry for dgemm32
hEntry = RTW.TflBlasEntryGenerator;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                            'Name',         'y1', ...
                            'IOType',       'RTW_IO_OUTPUT', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
```

```
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf inf]);

%% Create the Implementation Representation
% Using RTW.TflBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANSA');
% Possible values for PassByType property are
%  RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
%  RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = RTW.TflArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_blas
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for matrix multiplication for Mathworks Blas code’;
   this(1).TableList = {'crl_table_blas.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 73-125

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

You can develop a code replacement library for floating-point matrix/matrix and matrix/vector multiplication operations with the multiplication functions `sgemm` defined in the MathWorks C BLAS library. Use a third-party C BLAS library for replacement and change the build requirements in this example to point to your library. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_cblas`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **CBlas Operation (Atlas)**.

4. Create entry parameters. In the **Function** drop-down list, select `Multiply`.

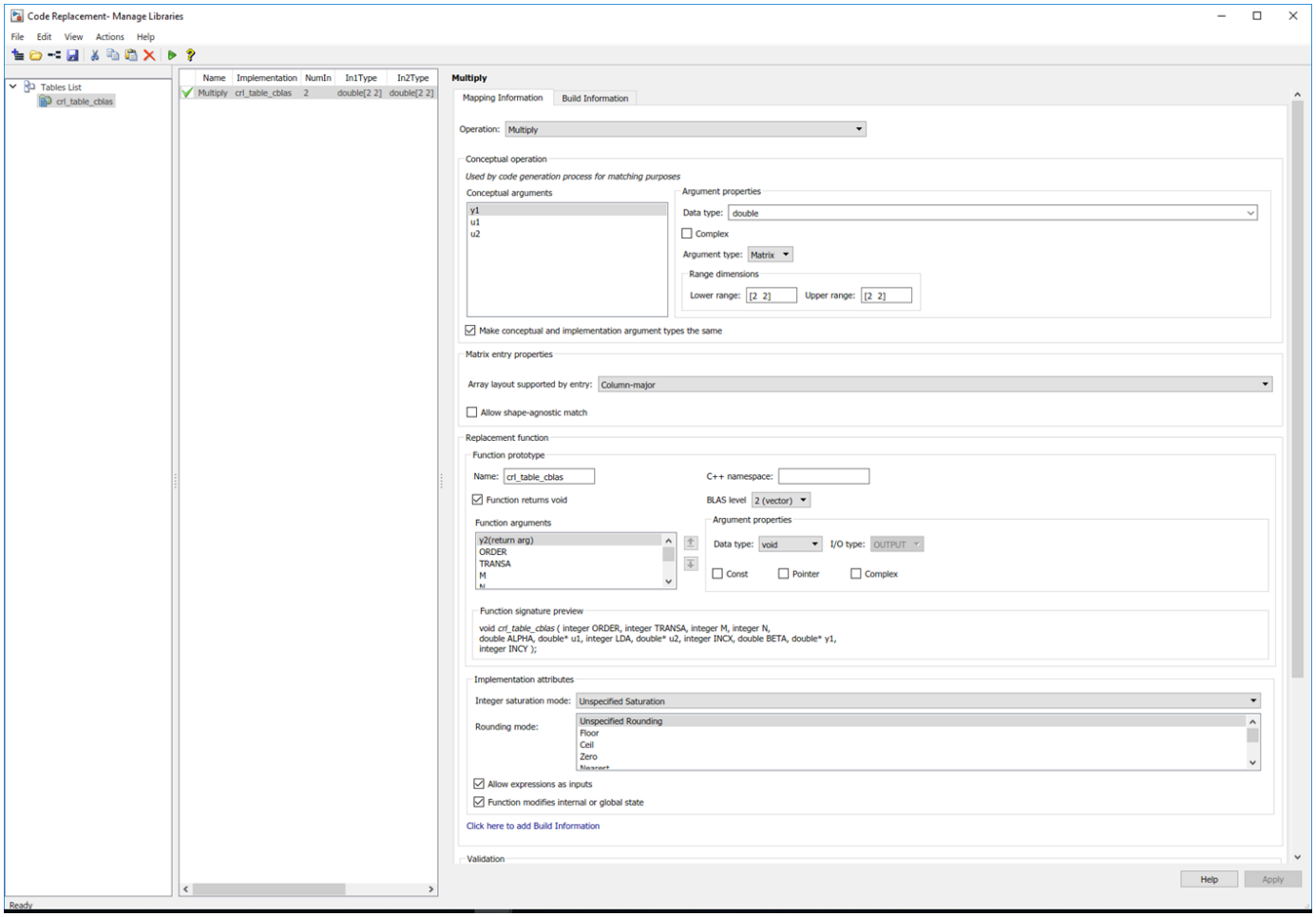
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of single and the **Argument Type** of Matrix.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. Set the **BLAS level** to `3(Vector)`.

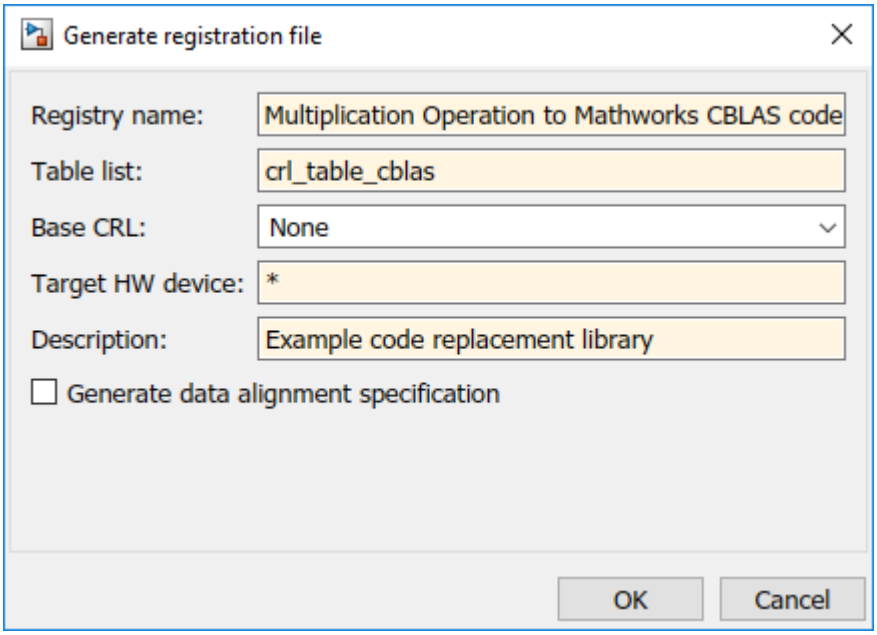
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_cblas
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCBlasEntryGenerator`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
```

```
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',        {'-lgfortran'},...
    'SideEffects', true);
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, [2 2; inf inf] means a two-dimensional matrix of size 2x2 or larger.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
```

```
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',      {'-lgfortran'},...
    'SideEffects', true);

%% Create the conceptual representation
arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'single');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'single');
arg.DimRange = [2 1; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'single');
arg.DimRange = [1 2; Inf Inf];
hEnt.addConceptualArg(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `getTflArgFromString`. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_cblas
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
%%%%%%%%%%%% Define library path Cblas library file%%%%%%%%%%%%%%%%%%

LibPath = fullfile('<Specify the path of your cblas library>');

if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end


% Create table entry for dgemm32
hEntry = RTW.TflCBlasEntryGenerator;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters( ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_sgemm', ...
    'ImplementationSourceFile', 'ccblas.c', ...
    'ImplementationHeaderFile', 'ccblas.h', ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'AdditionalLinkObjs',       {['libcblas' libExt]}, ...
    'AdditionalLinkFlags',      {'-lgfortran'},...
    'SideEffects', true);

%% Create the conceptual representation
```

```
arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'single');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'single');
arg.DimRange = [2 1; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'single');
arg.DimRange = [1 2; Inf Inf];
hEnt.addConceptualArg(arg);

%% Create the Implementation Representation
% Using RTW.TflCBlasEntryGenerator for sgemm requires the following
% implementation signature:
%
% void no_name( const integer ORDER, const integer TRANSA, const integer TRANSB,
% const integer M, const integer N, const integer K, const double ALPHA,
% const double* u1, const integer LDA, const double* u2, const integer LDB,
% const double BETA, double* y1, const integer LDC );
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature
arg = hEntry.getTflArgFromString('y2','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.setReturn(arg);

arg = hEntry.getTflArgFromString('ORDER','integer',102.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('TRANSA','integer',111.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('TRANSB','integer',111.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('M','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('N','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('K','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('ALPHA','double',1.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u1','single*');
arg.Type.BaseType.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('LDA','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('u2','single*');
arg.Type.BaseType.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('LDB','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('BETA','double',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);

arg = hEntry.getTflArgFromString('y1','single*');
arg.IOType = 'RTW_IO_OUTPUT';
hEntry.Implementation.addArgument(arg);
```

```
arg = hEntry.getTflArgFromString('LDC','integer',0.000000000000000);
arg.Type.ReadOnly = true;
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_cblas
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for matrix multiplication for Mathworks CBlas code';
   this(1).TableList = {'crl_table_cblas.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 73-117

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
                       'SideEffects',              true );
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

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

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

   arg = getTflArgFromString(hTable, 'u2', 'uint8');
   addConceptualArg(op_entry, arg );
   ```

6. Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the

arguments. When defining the implementation function return argument, create a new `void` output argument, for example, `y2`. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Create new void output y2
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);
```

7. Add the entry to a code replacement table with a call to the `addEntry` function.

   ```
   addEntry(hTable, op_entry);
   ```

8. Save the table definition file. Use the name of the table definition function to name the file.

See Also

More About

- "Code You Can Replace from MATLAB Code" on page 73-5
- "Define Code Replacement Library Optimizations" on page 73-23
- "Develop a Code Replacement Library" on page 73-13

Fixed-Point Operator Code Replacement

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

Common Ways to Match Fixed-Point Operator Entries

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point binary-point-only scaling. • `FractionLength`: Specify a fraction length (for example, 3).
A specific slope bias scaling combination on the operator inputs and output.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/-FixedExponent` equivalent): Specify a slope value (for example, 15). • `Bias`: Specify a bias value (for example, 2).

Match	Create entry	Minimally specify parameters
Net slope between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntry-Generator_NetSlope`	`setTflCOperationEntryParameters` function: • `NetSlopeAdjustmentFactor`: Specify the slope adjustment factor (`F`) part of the net slope, $F2^E$ (for example, `1.0`). • `NetFixedExponent`: Specify the fixed exponent (`E`) part of the net slope, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.
Relative scaling between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `RelativeScalingFactorF`: Specify the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$ (for example, `1.0`). • `RelativeScalingFactorE`: Specify the fixed exponent (`E`) part of the relative scaling factor, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.
Equal slope and zero net bias across operator inputs and output (addition and subtraction).	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes $\tilde{V}$, referred to as the quantized integer.
- S is a coefficient of Q, referred to as the slope.
- B is an additive correction, referred to as the bias.

The general equation for an operation between fixed-point operands is:

$$(S_O Q_O + B_O) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

The objective of fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 + \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. To match for replacement, the slopes must be the same for all addition conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 - \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the operator entry must set the operator entry parameters

`SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. To match for replacement, the slopes must be the same for all subtraction conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few known slope and bias combinations. Use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0Q_0 = (S_1Q_1)(S_2Q_2)$$

$$Q_0 = \left(\frac{S_1S_2}{S_0}\right)Q_1Q_2$$

$$Q_0 = S_nQ_1Q_2$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0. Also, set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all multiplication conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few known slope and bias combinations. Use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry (see “Customize Match and Replacement Process” on page 72-132).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0Q_0 = \left(\frac{S_1Q_1}{S_2Q_2}\right)$$

$$Q_0 = S_n\left(\frac{Q_1}{Q_2}\right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. For the `s16_netslope0p5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0. Also, set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all division conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope. Set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all cast conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n}\right)$$

$$Q_0 = \left(\frac{S_1}{S_0}\right)\left(\frac{Q_1}{2^n}\right)$$

$$Q_0 = S_n\left(\frac{Q_1}{2^n}\right)$$

where S_n is the net slope. Set the operator entry parameter `SlopesMustBeTheSame` to `false` and the parameter `MustHaveZeroNetBias` to `true`. To match for replacement, the biases must be zero for all shift conceptual arguments. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Binary-Point-Only Scaling Code Replacement” on page 73-140
- “Slope Bias Scaling Code Replacement” on page 73-145
- “Net Slope Scaling Code Replacement” on page 73-151
- “Equal Slope and Zero Net Bias Code Replacement” on page 73-158
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 73-164
- “Shift Left Operations and Code Replacement” on page 73-170
- “Remap Operator Output to Function Input” on page 73-132
- “Customize Match and Replacement Process” on page 73-93
- “Develop a Code Replacement Library” on page 73-13

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination of operator inputs and output. This example shows how to develop a code replacement library to optimize the performance of fixed point data type operations by providing information that maps a binary point scale entry to a `addition` operator. To develop a code replacement library use only the programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process see “Customize Code Match and Replacement for Scalar Operations” on page 72-139. You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntryBinaryPoint < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = 0;
        end

        % Set the fraction-length values in the implementation function.
        ent.Implementation.Arguments(3).Value = ...
```

```
            -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
         ent.Implementation.Arguments(4).Value = ...
            -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
         ent.Implementation.Arguments(5).Value = ...
            -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
      end
    end
  end
end
```

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_fixed_binptscale()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the custom entry function .

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function `createAndSetCImplementationReturn` and `createAndAddImplementationArg`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_fixed_binptscale()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntryBinaryPoint;


%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_binarypoint', ...
```

```
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);

%% Create the implementation Representation
% Specify replacement function signature
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_out', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);
```

```
%% Add the entry to the table
hTable.addEntry(op_entry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_fixed_binptscale
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for binary point scaling CRL’;
this(1).TableList = {'crl_table_fixed_binptscale.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Slope Bias Scaling Code Replacement” on page 73-145

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. This example shows how to develop a code replacement library to optimize the performance of fixed point data type operations by providing information on how to define code replacement for a `division` operator. To develop a code replacement library use only the programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process, see “Customize Code Match and Replacement for Scalar Operations” on page 72-139. You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntrySlopeBias < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
          ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
          ent.ConceptualArgs(idx).Type.Bias = hCSO.ConceptualArgs(idx).Type.Bias;
        end

        % Set the Slope and Bias values in the implementation function.
        ent.Implementation.Arguments(3).Value = hCSO.ConceptualArgs(2).Type.Slope;
        ent.Implementation.Arguments(4).Value = hCSO.ConceptualArgs(2).Type.Bias;
        ent.Implementation.Arguments(5).Value = hCSO.ConceptualArgs(3).Type.Slope;
        ent.Implementation.Arguments(6).Value = hCSO.ConceptualArgs(3).Type.Bias;
```

```
        ent.Implementation.Arguments(7).Value = hCSO.ConceptualArgs(1).Type.Slope;
        ent.Implementation.Arguments(8).Value = hCSO.ConceptualArgs(1).Type.Bias;
      end
    end
  end
end
```

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New > Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the custom entry function .

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;
```

4. Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `createAndAddConceptualArg`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'SlopeBias', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'SlopeBias', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'SlopeBias', ...
       'IsSigned',   false, ...
       'WordLength', 32);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the same order and properties as the conceptual arguments, call the function

`createAndSetCImplementationReturn` and `createAndAddImplementationArg`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_custom_add_ufix32_slopebias
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = TflCustomOperationEntrySlopeBias;;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd_slopebias', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'SlopeBias', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'SlopeBias', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'SlopeBias', ...
       'IsSigned',   false, ...
       'WordLength', 32);

%% Create the implementation Representation
% Specify replacement function signature
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
```

```
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 6 args for slopes and biases.
% Actual values are set during code generation.
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'slope_out1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',       'bias_out1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_custom_add_ufix32_slopebias
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for slope bias scaling function replacement';
this(1).TableList = {'crl_table_custom_add_ufix32_slopebias.m'};
% table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Binary-Point-Only Scaling Code Replacement” on page 73-140

Net Slope Scaling Code Replacement

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example shows how to develop a code replacement library to optimize the performance of fixed point net slope operations by providing information on how to define code replacement for a `divide` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_netslopeScaling`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Divide`.

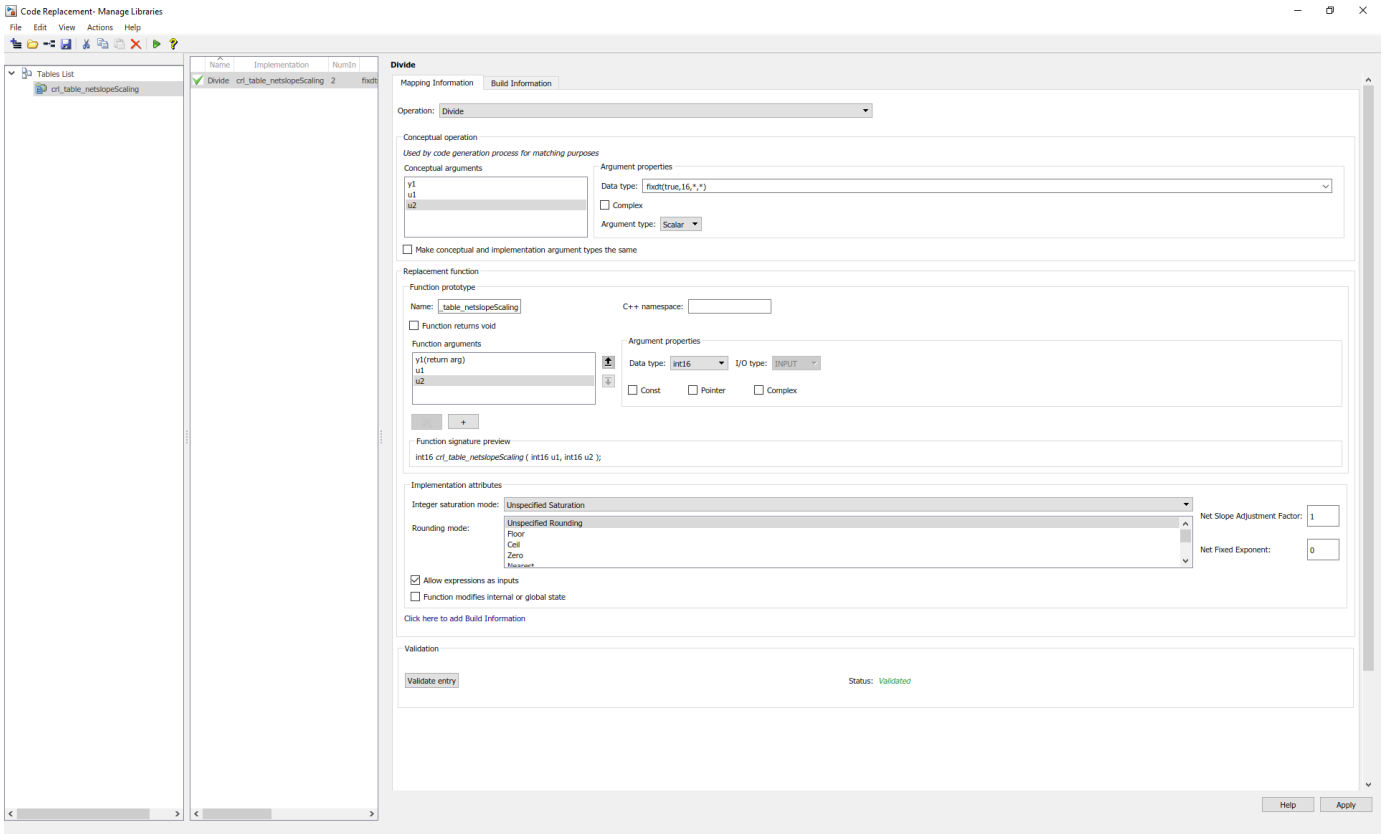
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `fixdt(true,16,*,*)` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `int 16`. Also set the **Net slope Adjustment Factor** as 1 and **Net Fixed Exponent** as 0.

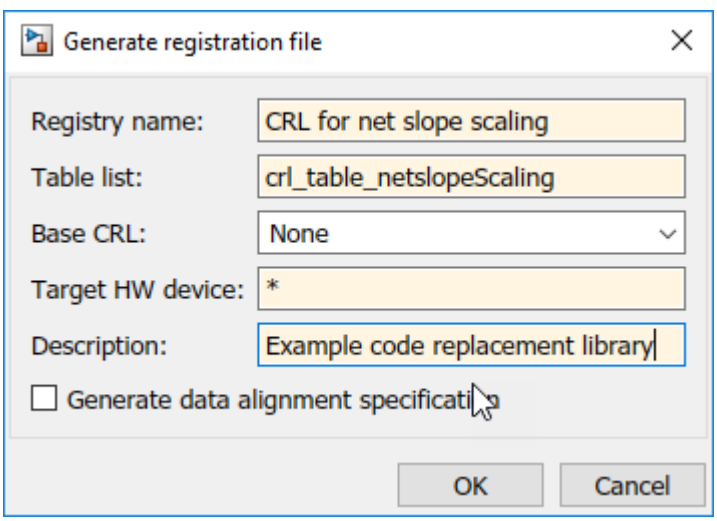
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For

more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1. Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.
2. Create a table.
   a. Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.
   b. Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3. Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntryGenerator_NetSlope`.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
```

```
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'y1', ...
   'IOType',         'RTW_IO_OUTPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
function hTable = crl_table_netslopeScaling
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
   'ImplementationName',       's16_div_s16_s16', ...
   'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
   'ImplementationSourceFile', 's16_div_s16_s16.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'y1', ...
   'IOType',         'RTW_IO_OUTPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
   'IOType',         'RTW_IO_INPUT', ...
   'CheckSlope',     false, ...
   'CheckBias',      false, ...
   'DataType',       'Fixed', ...
   'IsSigned',       true, ...
   'WordLength',     16);

%% Create the Implementation Representation
```

```
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
                                  'Name',           'y1', ...
                                  'IOType',         'RTW_IO_OUTPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
                                  'Name',           'u1', ...
                                  'IOType',         'RTW_IO_INPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
                                  'Name',           'u2', ...
                                  'IOType',         'RTW_IO_INPUT', ...
                                  'IsSigned',       true, ...
                                  'WordLength',     16, ...
                                  'FractionLength', 0);

%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

   ```
   >> hTable = crl_table_netslopeScaling
   ```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file by using these specifications:

   ```
   function rtwTargetInfo(cm)

   cm.registerTargetInfo(@loc_register_crl);
   end

   function this = loc_register_crl

   this(1) = RTW.TflRegistry;
   this(1).Name = 'CRL for net slope scaling’;
   this(1).TableList = {'crl_table_netslopeScaling.m'}; % table created in this example
   this(1).TargetHWDeviceType = {'*'};
   this(1).Description = 'Example code replacement library';

   end
   ```

   To use your code replacement library, refresh your current MATLAB session with the command:

   ```
   >>sl_refresh_customizations
   ```

10. Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Equal Slope and Zero Net Bias Code Replacement” on page 73-158

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match the relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values when mapping relative slope and bias values to a replacement function for addition or subtraction.

This example shows how to develop a code replacement library to optimize the performance of fixed point net slope operations by providing information on how to define code replacement for an `addition` operation. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_equal_slope_netbias_add`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

4. Create entry parameters. In the **Operation** drop-down list, select `Addition`.

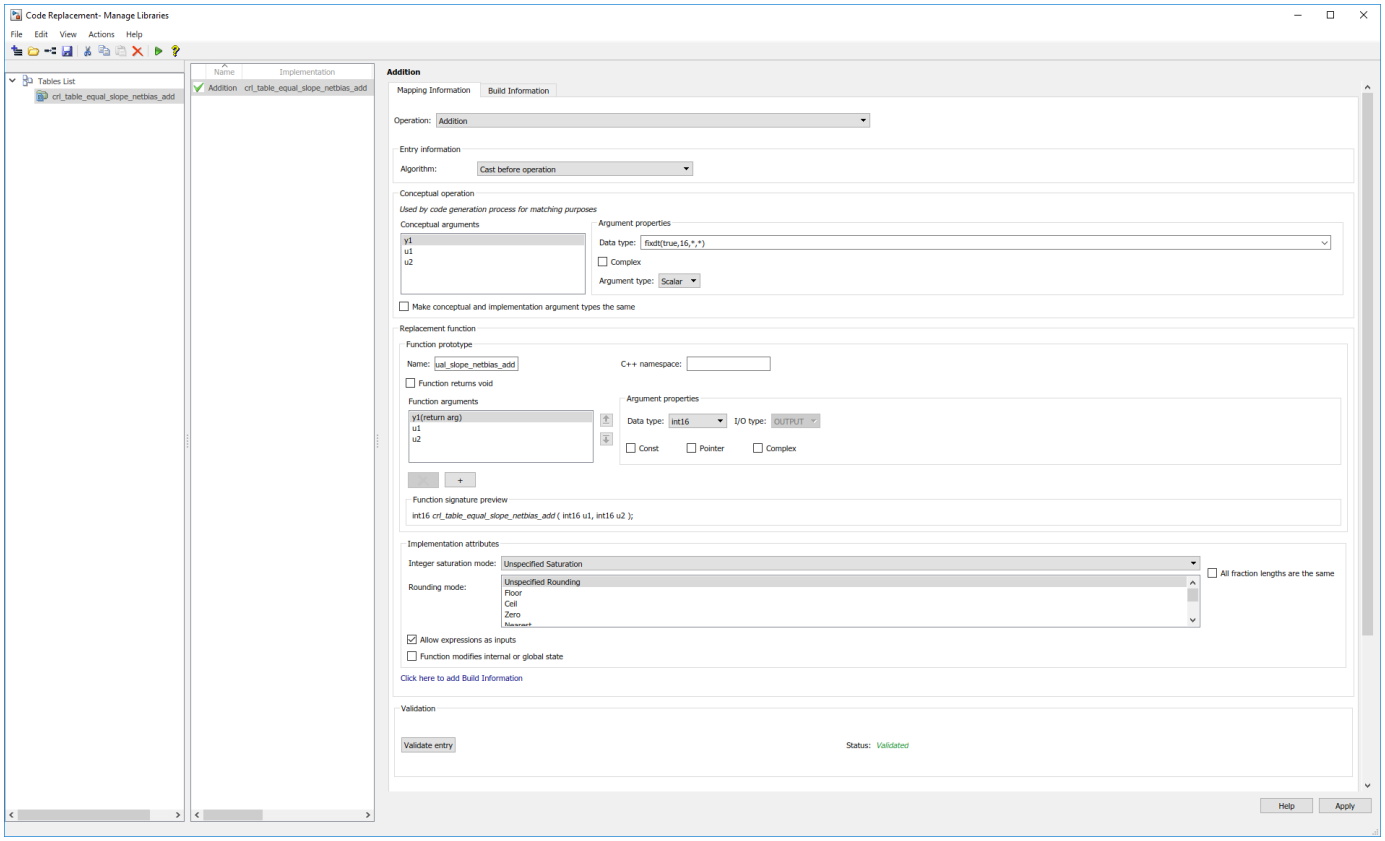
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `fixdt(true,16,*,*)` and the **Argument Type** of Scalar.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` and `u2` with the **Data Type** of `int 16`.

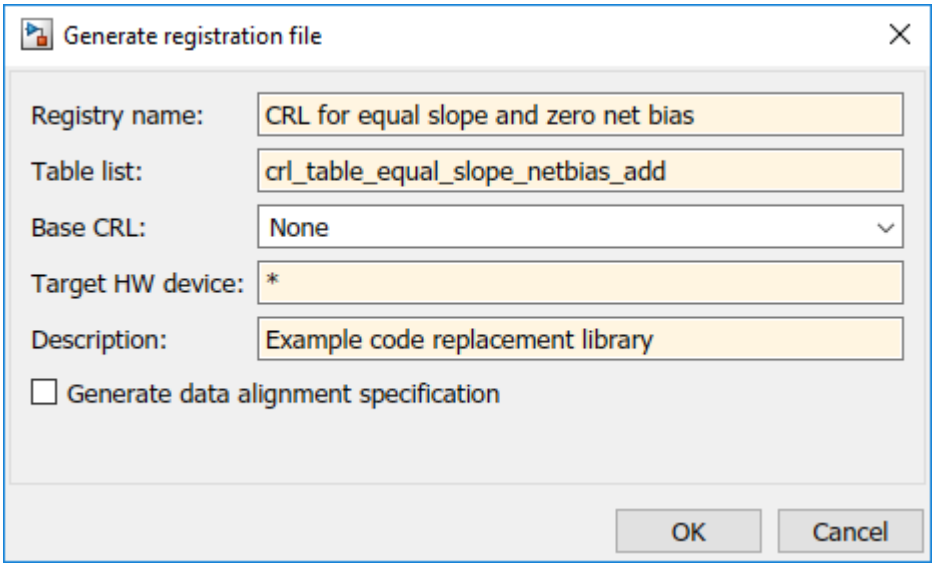
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntryGenerator_NetSlope`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. This example uses calls to the `createAndSetCImplementationReturn` function to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
function hTable = crl_table_equal_slope_netbias_add
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                  90, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',             {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',       true, ...
    'MustHaveZeroNetBias',       true, ...
```

```
    'ImplementationName',       'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

%% Create the conceptual representation
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);

createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'CheckSlope',     false, ...
    'CheckBias',      false, ...
    'IsSigned',       false, ...
    'WordLength',     16);


%% Create the Implementation Representation
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);

createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric',...
    'Name',           'u2', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     16, ...
    'FractionLength', 0);


%% Add the entry to the table
addEntry(hTable, hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_equal_slope_netbias_add
```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for equal slope and zero net bias code replacement’;
```

```
this(1).TableList = {'crl_table_equal_slope_netbias_add.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Net Slope Scaling Code Replacement” on page 73-151

Data Type Conversions (Casts) and Operator Code Replacement

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), at the MATLAB command line, enter:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_cast_int32_to_int16`. Click **Apply**.

3. Create entry parameters. In the **Operation** drop-down list, select `Cast`.

4. Create an entry. From the crtool context menu, select **File** > **New entry** > **Fixed Point Net Slope Operation**.

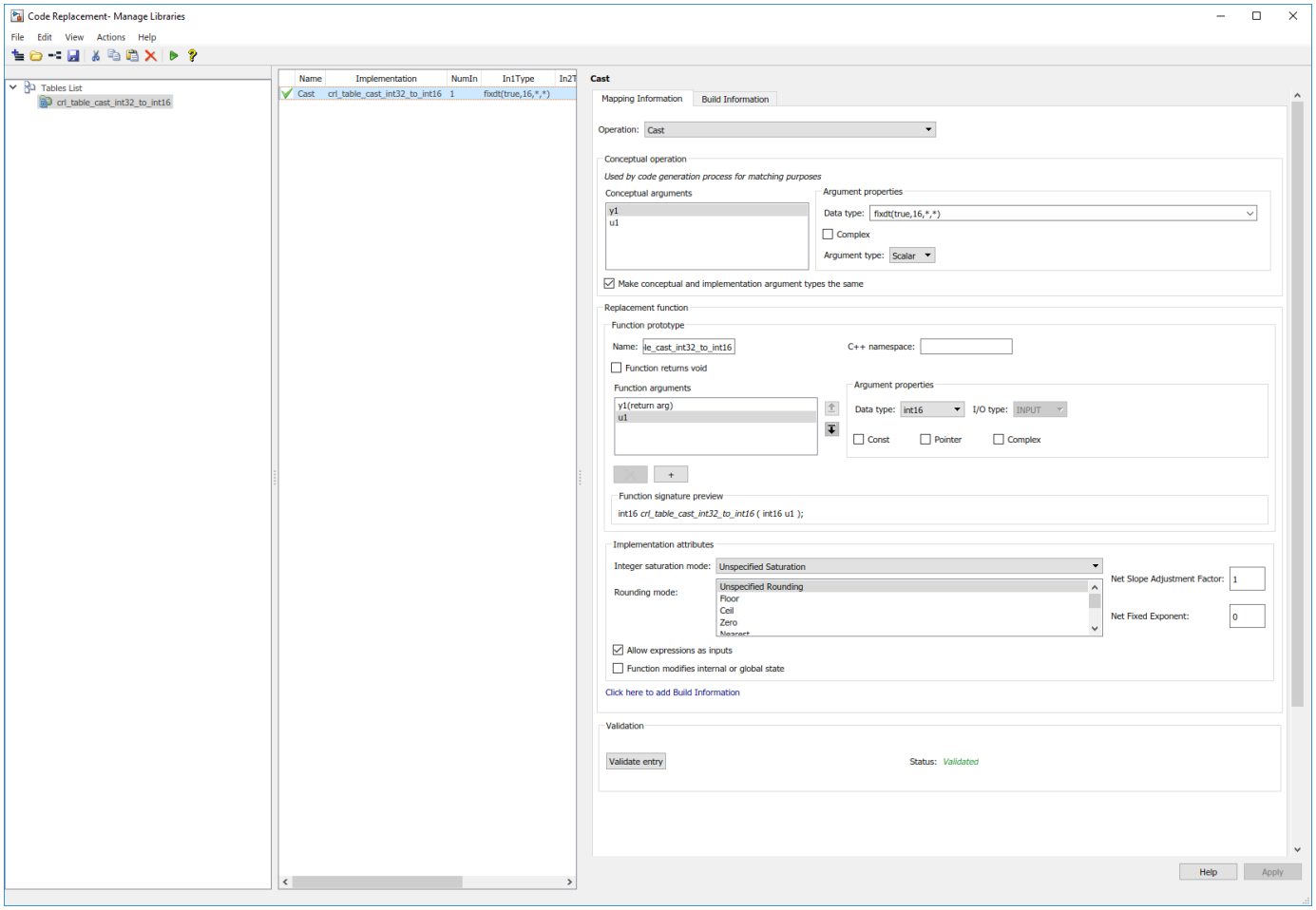
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1` with the **Data Type** of `int16` , and the input argument, `u1` with the **Data Type** of `int32` and the **Argument Type** of `Scalar`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box. In the **Function arguments** subsection of the crtool, specify the return argument, `y1`, and the input argument, `u1` with the **Data Type** of `int16` and `int32`.

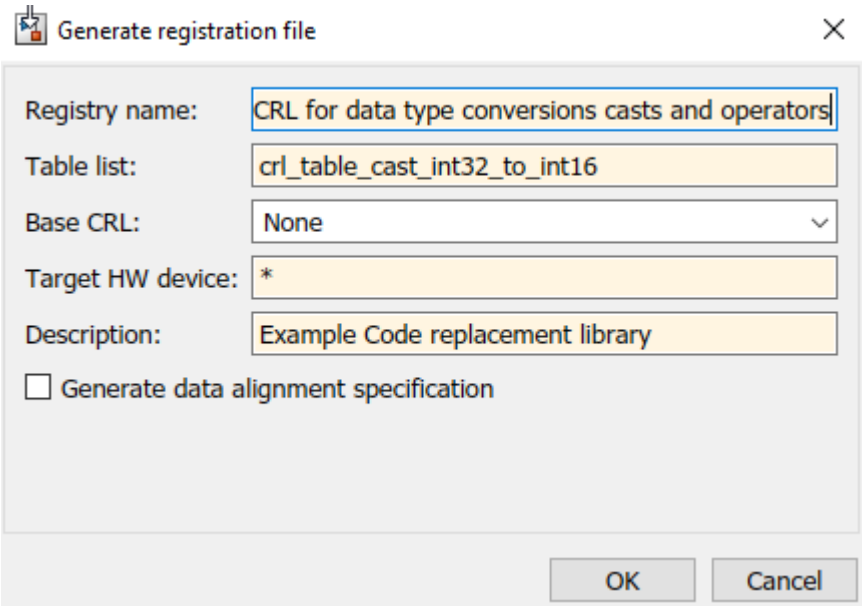
   Specify a **Name** for the replacement function under **Function prototype**.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. In the **Mapping Information** tab, click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see "Verify Code Replacement Library" on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programmatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function to call your code replacement library table. The function should not have arguments and return a table object.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

```
%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this examples replaces a function, create entry parameters by calling the function `setTflCOperationEntryParameters`.The parameters specify the type of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual representation

arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To specify that the implementation arguments have the

same order and properties as the conceptual arguments, call the function `getTflArgFromString`. Add the complete entry to the table by calling the function `addEntry`.

```
function hTable = crl_table_cast_int32_to_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
hEntry.setTflCOperationEntryParameters(...
    'Key',                      'RTW_OP_CAST', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_sat_cast', ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual and implementation representation

arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);


%% Add the entry to the table
addEntry(hTable,hEntry);
```

7. Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.
8. Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File** > **Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_cast_int32_to_int16
```

9. Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file with these specifications:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for data type conversions(casts) abd operator code replacement’;
this(1).TableList = {'crl_table_cast_int32_to_int16.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Net Slope Scaling Code Replacement” on page 73-151

Shift Left Operations and Code Replacement

This example shows how to develop a code replacement library to optimize the performance of shift left (<<) operation by providing information on how to define code replacement for a `shift left` function. To develop a code replacement library use either the interactive or programmatic approach. For more information, see “Develop a Code Replacement Library” on page 73-13.

Interactively Develop a Code Replacement Library

1. Open the **Code Replacement Tool** (crtool), from the MATLAB command line with the following command:

   ```
   >>crtool
   ```

2. Create a table.

   a. From the crtool context menu, select **File** > **New Table**.

   b. In the right pane, name the table `crl_table_shift_left_int16`. Click **Apply**.

3. Create an entry. From the crtool context menu, select **File** > **New entry** > **Math Operation**.

4. Create entry parameters. In the **Operator** drop-down list, select `Shift Left`.

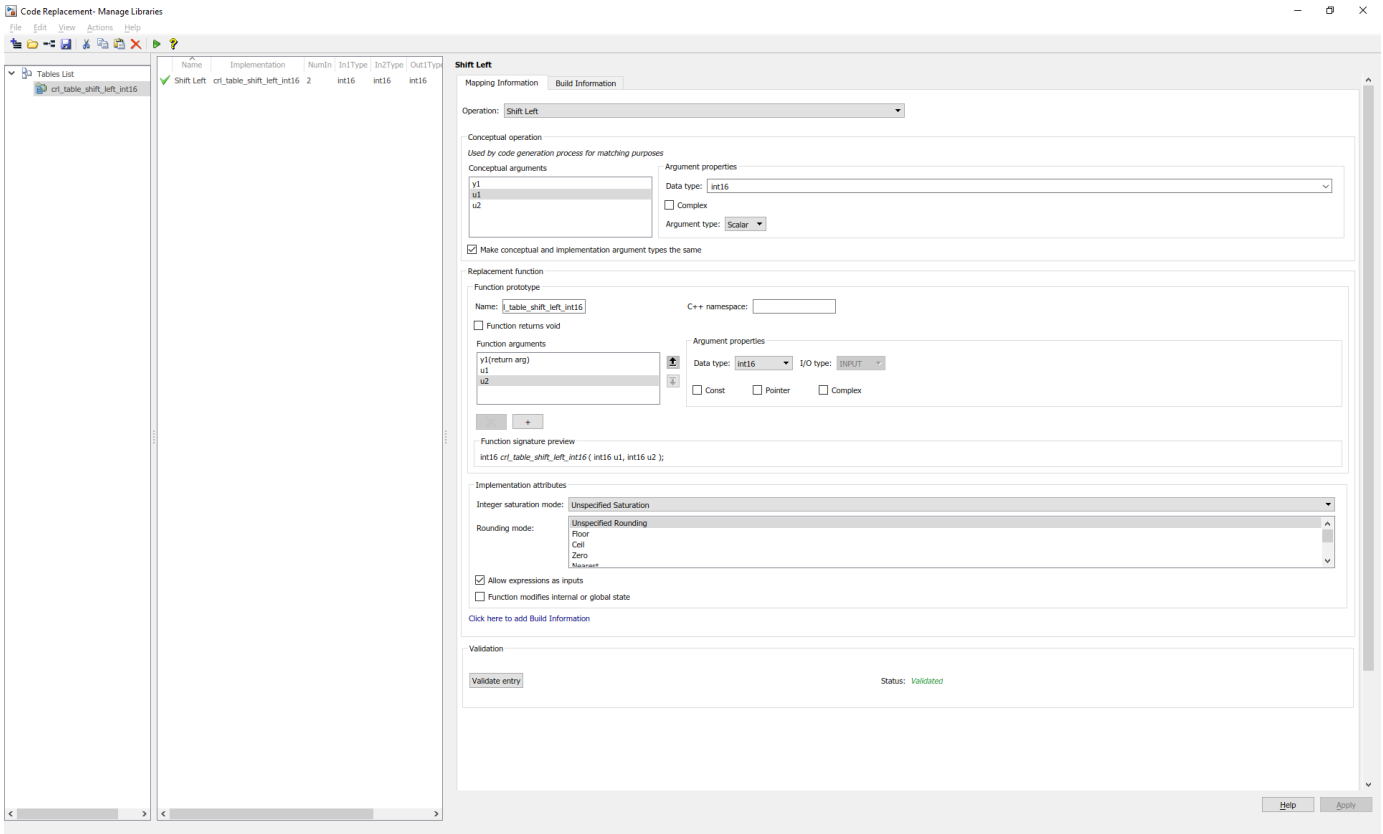
5. Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. In the **Conceptual function** subsection of the crtool, specify the return argument, `y1`, and the input arguments, `u1`, and `u2`, with **Data Type** of `int16` and the **Argument Type** of `scalar`.

6. Create the implementation representation. The implementation representation describes the signature of the optimization function. For this example, to specify that the implementation arguments have the same order and properties as the conceptual arguments, select the **Make conceptual and implementation argument types the same** check box.

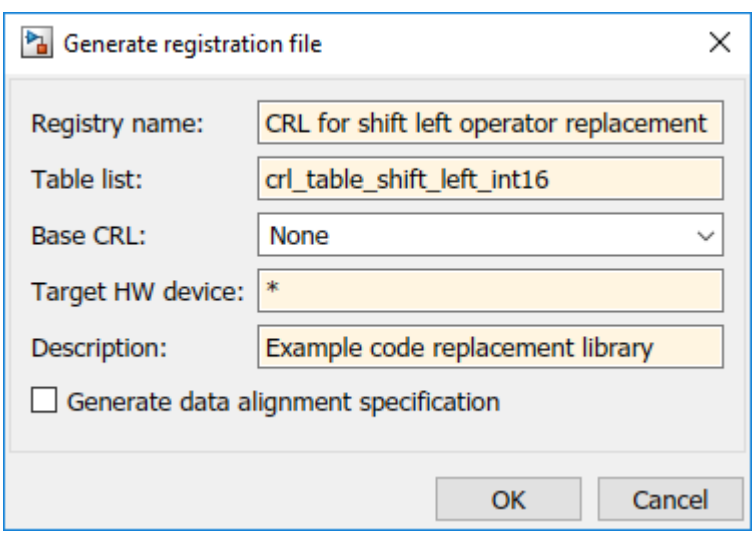
   Under **Function prototype**specify a **Name** for the replacement function.

7. Specify build information. Click the **Build Information** tab to open the build requirements pane. Specify the files (source, header, object) that the code generator requires for code replacement. For this example, you do not need to specify build information.

8. Validate and save the table. Click the **Mapping Information** tab and verify the fields are filled in as shown. Click **Validate entry**. In the crtool context menu, select **File** > **Save table** > **Save**.



9 Register a code replacement library. Registration creates a library composed of the tables that you specify. Select **File > Generate registration file**. In the Generate registration file dialog box, fill out these fields:



To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

Programatically Develop a Code Replacement Library

1 Open the programmatic interface from the **MATLAB** menu by selecting **New** > **Function**.

2 Create a table.

   a Create a function with the name of your code replacement library table that does not have arguments and returns a table object. You can use this function to call your code replacement library table.

   b Create a table object by calling `RTW.TflTable`.

```
function hTable = crl_table_shift_left_int16
% Create a function to call the code replacement library table

%% Create a table object
hTable = RTW.TflTable;
```

3 Create an entry. Because this example replaces a function, create a code replacement entry in your table by calling the entry function `RTW.TflCOperationEntry`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;
```

4 Create entry parameters. Because this example replaces a function, create entry parameters by calling the function `setTflCFunctionEntryParameters`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');
```

5 Create the conceptual representation. The conceptual representation describes the signature of the function that you want to replace. To explicitly specify argument properties, call the function `getTflArgFromString` and `addConceptualArg` .

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual representation
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', 'int16');
arg.CheckType = false;
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);
```

6 Create the implementation representation. The implementation representation describes the signature of the optimization function. To explicitly specify argument properties, call the function `getTflArgFromString`.

```
function hTable = crl_table_shift_left_int16()
% Create a code replacement library table

%% Create a table object
hTable = RTW.TflTable;

%% Create an entry
hEntry = RTW.TflCOperationEntry;

%% Create entry parameters
setTflCOperationEntryParameters(hEntry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

%% Create the conceptual and implementation representation
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEntry, arg);
hEntry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', 'int16');
arg.CheckType = false;
addConceptualArg(hEntry, arg);
hEntry.Implementation.addArgument(arg);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

7 Specify build information. In the entry parameters, specify files (header, source, object) that the code generator needs for code replacement. For this example, build information is not required.

8 Validate and save the customization file. From the **MATLAB** menu, save this customization file by selecting **File > Save**. From the command line, validate the code replacement library table by calling it:

```
>> hTable = crl_table_shift_left_int16
```

9 Register the code replacement library. Registration creates a code replacement library by defining the library name, code replacement tables, and other information. Create a registration file (a new function file) by using these specifications.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);
end

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'CRL for shift left operation’;
this(1).TableList = {'crl_table_shift_left_int16.m'}; % table created in this example
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example code replacement library';

end
```

To use your code replacement library, refresh your current MATLAB session with the command:

```
>>sl_refresh_customizations
```

10 Verify the code replacement library. From the MATLAB command line, open the library by using the **Code Replacement Viewer** and verify that the table and entry are correctly specified. For more information, see “Verify Code Replacement Library” on page 72-55. Configure your model to use the code replacement library, generate code, and verify that replacement occurs as expected. If unexpected behavior occurs, examine the hit and miss logs to troubleshoot the issues.

See Also

More About

- “Code You Can Replace from MATLAB Code” on page 73-5
- “Define Code Replacement Library Optimizations” on page 73-23
- “Develop a Code Replacement Library” on page 73-13
- “Net Slope Scaling Code Replacement” on page 73-151

Optimize Generated Code by Developing and Using Code Replacement Libraries - MATLAB

Develop and use code replacement libraries to replace functions and operators in generated code. Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware
- Integration with existing application code
- Compliance with a standard, such as AUTOSAR
- Modification of code behavior, such as enabling or disabling nonfinite or inline support
- Application- or project-specific code requirements, such as use of BLAS or elimination of `math.h`, system header files, or calls to `memcpy` or `memset`.

You can configure the code generator to use a code replacement library that MathWorks® provides. If you have an Embedded Coder® license, you can develop your own code replacement library interactively with the Code Replacement Tool or programmatically.

This example provides MATLAB® code that shows a variety of ways you can define code replacement mappings. With each example MATLAB function, the example provides MATLAB files that illustrate how to develop function and operator code replacements programmatically.

For more information, see “What Is Code Replacement Customization?” on page 73-3 and “Develop a Code Replacement Library” on page 73-13.

Create a Registration File

Create and save a registration file. Because an `rtwTargetInfo.m` file may already exist on the path, this example creates a new file from a file with a different name.

```
copyfile myRtwTargetInfoCRL.txt rtwTargetInfo.m
```

Steps for Developing a Code Replacement Library

1. Identify your code replacement requirements with respect to function or operating mappings, build information, and registration information.
2. Prepare for code replacement library development (for example, identify or develop models to test your library).
3. Define code replacement mappings.
4. Specify build information for replacement code.
5. Register code replacement mappings.
6. Verify code replacements.
7. Deploy the library.

For more information, see “Develop a Code Replacement Library” on page 73-13.

Math Function Replacement

This example defines and registers code replacement mappings for math functions. You can define code replacement mappings for a variety of functions. For details, see “Code You Can Replace from MATLAB Code” on page 73-5.

1. The `replace_math_fcns.m` file contains this code:

```
function [y1, y2] = replace_math_fcns(u1, u2) %#codegen

%   Copyright 2008-2010 The MathWorks, Inc.

y1 = single(atan2(u1,u2));
y2 = double(cos(double(u1)) + sin(double(u2)));
```

2. Configure the code generator to use the code replacement library **Function Replacement Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'Function Replacement Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the function input type.

```
t = single(2);
```

4. Open and explore the code replacement table definition file, `crl_table_functions.m`.

5. Compile the MATLAB program into a C source file by using the configuration parameters that point to the code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('replace_math_fcns', '-config', cfg, '-args', {t, t});
```

```
Code generation successful.
```

6. The `replace_math_fcns.c` file contains this generated code:

```
file = fullfile('codegen','lib','replace_math_fcns','replace_math_fcns.c');
rtwdemodbtype(file,'#include "replace_math_fcns.h','* File trailer for replace_math_fcns.c',1,0)
```

```
#include "replace_math_fcns.h"
#include "MyMath.h"

/* Function Definitions */
/*
 * Copyright 2008 The MathWorks, Inc.
 *
 * Arguments    : float u1
 *                float u2
 *                float *b_y1
 *                double *y2
 * Return Type  : void
 */
void replace_math_fcns(float u1, float u2, float *b_y1, double *y2)
{
```

```
  *b_y1 = atan2_single(u1, u2);
  *y2 = cos_double(u1) + sin_double(u2);
}

/*
```

For more information on math function replacement, see “Math Function Code Replacement” on page 73-51.

Addition and Subtraction Operator Replacement

This example shows how to define and register code replacement mappings for addition (+) and subtraction (-) operations. When defining entries for addition and subtraction operations, you can specify which of the following algorithms (`EntryInfoAlgorithm`) your library functions implement:

- Cast-before-operation (CBO) (`RTW_CAST_BEFORE_OP`), the default
- Cast-after-operation (CAO) (`RTW_CAST_AFTER_OP`)

1. The `addsub_two_int16.m` file contains this code:

```
function [y1, y2] = addsub_two_int16(u1, u2) %#codegen

%   Copyright 2008-2010 The MathWorks, Inc.

y1 = int16(u1 + u2);
y2 = int16(u1 - u2);
```

Note, the code uses the default algorithm, CBO.

2. Configure the code generator to use a code replacement library **Addition & Subtraction Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'Addition & Subtraction Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the operation input type.

```
t = int16(2);
```

4. Open and explore the code replacement table definition file, `addsub_two_int16(u1, u2).m`.

5. Compile the MATLAB program into a C source file by using the configuration parameters that point to the desired code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('addsub_two_int16','-config', cfg, '-args', {t, t});
```

```
Code generation successful.
```

6. The `addsub_two_int16.c` file contains this generated code.

```
file = fullfile('codegen','lib','addsub_two_int16','addsub_two_int16.c');
rtwdemodbtype(file,'#include "addsub_two_int16.h','* File trailer for addsub_two_int16.c',1,0);
```

```
#include "addsub_two_int16.h"
#include "MyMath.h"
```

```
/* Function Definitions */
/*
 * Copyright 2008 The MathWorks, Inc.
 *
 * Arguments    : short u1
 *                short u2
 *                short *b_y1
 *                short *y2
 * Return Type  : void
 */
void addsub_two_int16(short u1, short u2, short *b_y1, short *y2)
{
  *b_y1 = s16_add_s16_s16(u1, u2);
  *y2 = s16_sub_s16_s16(u1, u2);
}

/*
```

For more information on addition and subtraction operator replacement, see “Scalar Operator Code Replacement” on page 73-99 and “Addition and Subtraction Operator Code Replacement” on page 73-104.

Matrix Operator Replacement

This example defines and registers code replacement mappings for matrix operations: addition, subtraction, multiplication, transposition, conjugate, and Hermitian.

Supported types include:

- `single`, `double`
- `int8`, `uint8`
- `int16`, `uint16`
- `int32`, `uint32`
- `csingle`, `cdouble`
- `cint8`, `cuint8`
- `cint16`, `cuint16`
- `cint32`, `cuint32`
- fixed-point integers
- mixed types (different type on each input)

1. The `replace_matrix_ops.m` file contains this code:

```
function [y1, y2, y3] = replace_matrix_ops(u1, u2) %#codegen
% This block supports the Embedded MATLAB subset.
% See the help menu for details.

%   Copyright 2010 The MathWorks, Inc.

y1 = u1 + u2;
y2 = u1 - u2;
y3 = y2';
```

2. Configure the code generator to use a code replacement library **Matrix Op Replacement Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'Matrix Op Replacement Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the operation input type.

```
t = [1.0 2.0; 3.0, 4.0];
```

4. Open and explore the code replacement table definition file, crl_table_matops.m.

5. Compile the MATLAB program using the configuration parameters that point to the desired code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('replace_matrix_ops', '-config', cfg, '-args', {t, t});
```

```
Code generation successful.
```

6. The `replace_matrix_ops.c` file contains this generated code:

```
file = fullfile('codegen','lib','replace_matrix_ops','replace_matrix_ops.c');
rtwdemodbtype(file,'#include "replace_matrix_ops.h','* File trailer for replace_matrix_ops.c',1,(
```

```
#include "replace_matrix_ops.h"
#include "MatrixMath.h"

/* Function Definitions */
/*
 * This block supports the Embedded MATLAB subset.
 *  See the help menu for details.
 *
 * Arguments    : const double u1[4]
 *                const double u2[4]
 *                double b_y1[4]
 *                double y2[4]
 *                double y3[4]
 * Return Type  : void
 */
void replace_matrix_ops(const double u1[4], const double u2[4], double b_y1[4],
                        double y2[4], double y3[4])
{
  /*    Copyright 2015 The MathWorks, Inc. */
  matrix_sum_2x2_double((double *)&u1[0], (double *)&u2[0], &b_y1[0]);
  matrix_sub_2x2_double((double *)&u1[0], (double *)&u2[0], &y2[0]);
  y3[0] = y2[0];
  y3[1] = y2[1];
  y3[2] = y2[2];
  y3[3] = y2[3];
}

/*
```

For more information on matrix operator replacement, see “Small Matrix Operation to Processor Code Replacement” on page 73-111.

Matrix Multiplication Replacement for BLAS

This example defines and registers code replacement mappings for Basic Linear Algebra Subroutines (BLAS) subroutines `xGEMM` and `xGEMV`. You can map the following operations to a BLAS subroutine:

- Matrix multiplication
- Matrix multiplication with transpose on single or both inputs
- Matrix multiplication with Hermitian operation on single or both inputs

1. The `replace_matrix_ops_blas.m` file contains this code:

```
function [y1, y2] = replace_matrix_ops_blas(u1, u2) %#codegen
% This block supports the MATLAB Coder subset.
% See the help menu for details.

%   Copyright 2010 The MathWorks, Inc.

y1 = u1 * u2;
y2 = u1 * u2;
```

2. Configure the code generator to use a code replacement library **BLAS Replacement Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'BLAS Replacement Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the function input type.

```
t = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0];
```

4. Open and explore the code replacement table definition file, `crl_table_blas.m` file.

5. Compile the MATLAB program using the configuration parameters that point to the desired code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('replace_matrix_ops_blas', '-config', cfg, '-args', {t, t});
```

```
Code generation successful.
```

6. The `replace_matrix_ops_blas.c` file contains this generated code.

```
file = fullfile('codegen','lib','replace_matrix_ops_blas','replace_matrix_ops_blas.c');
rtwdemodbtype(file,'#include "replace_matrix_ops_blas.h','* File trailer for replace_matrix_ops_b
```

```
#include "replace_matrix_ops_blas.h"
#include "blascompat32.h"
#include <string.h>

/* Function Definitions */
/*
 * This block supports the MATLAB Coder subset.
 *  See the help menu for details.
 *
 * Arguments    : const double u1[9]
 *                const double u2[9]
```

```
 *                double b_y1[9]
 *                double y2[9]
 * Return Type  : void
 */
void replace_matrix_ops_blas(const double u1[9], const double u2[9],
                             double b_y1[9], double y2[9])
{
  double ALPHA;
  double BETA;
  int K;
  int LDA;
  int LDB;
  int LDC;
  int M;
  int N;
  char TRANSA;
  char TRANSB;
  /*    Copyright 2015 The MathWorks, Inc. */
  TRANSA = 'N';
  TRANSB = 'N';
  M = 3;
  N = 3;
  K = 3;
  ALPHA = 1.0;
  LDA = 3;
  LDB = 3;
  BETA = 0.0;
  LDC = 3;
  dgemm32(&TRANSA, &TRANSB, &M, &N, &K, &ALPHA, &u1[0], &LDA, &u2[0], &LDB,
          &BETA, &b_y1[0], &LDC);
  memcpy(&y2[0], &b_y1[0], 9U * sizeof(double));
}

/*
```

For more information on matrix multiplication replacement for BLAS, see “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 73-117.

Replacements for Row-Major and Column-Major Matrix Operations

This example defines and registers code replacement mappings for row-major and column-major matrix operations.

1. The `replace_mixed_row_column_ops.m` file contains this code:

```
function [y1, y2 , y3, y4] = replace_mixed_row_column_ops( u1,u2, u3)
    coder.rowMajor;
    [y1,y2] = Child_Op(u1,u2,u3);
    y3 = u1*u2;
    y4 = u1+u3;
end

function [b1, b2] = Child_Op( a1,a2,a3)
    coder.columnMajor;
    coder.inline('never');
    b1 = a1*a2;
    b2 = a1+a3;
end
```

2. Configure the code generator to use a code replacement library **Row Major CRL Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'Row Major CRL Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the function input type.

```
t = [1 2; 3 4];
```

4. Open and explore the code replacement table definition files, `crl_table_row_major_op.m` and `crl_table_column_major_op.m`.

5. Compile the MATLAB program using the configuration parameters that point to the desired code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('replace_mixed_row_column_ops.m', '-config', cfg, '-args', {t, t, t});
```

```
Warning: C Compiler produced warnings. See the build log for further details.

Code generation successful (with warnings): To view the report, open('codegen\lib\replace_mixed_
```

6. The `replace_matrix_row_column_ops.c` file contains this generated code.

```
file = fullfile('codegen','lib','replace_mixed_row_column_ops','replace_mixed_row_column_ops.c')
rtwdemodbtype(file,'#include "replace_mixed_row_column_ops.h','* File trailer for replace_mixed_
```

```
#include "replace_mixed_row_column_ops.h"
#include "CRL_Multiplication.h"

/* Function Declarations */
static void Child_Op(const double a1[4], const double a2[4], const double a3[4],
                     double b1[4], double b2[4]);

/* Function Definitions */
/*
 * Arguments    : const double a1[4]
 *                const double a2[4]
 *                const double a3[4]
 *                double b1[4]
 *                double b2[4]
 * Return Type  : void
 */
static void Child_Op(const double a1[4], const double a2[4], const double a3[4],
                     double b1[4], double b2[4])
{
  MyMul_CUSTOM_COL((double *)&a1[0], (double *)&a2[0], &b1[0], 2U, 2U, 2U, 2U);
  MyAdd_CUSTOM_COL((double *)&a1[0], (double *)&a3[0], &b2[0], 2U, 2U, 2U, 2U);
}

/*
 * Arguments    : const double u1[4]
 *                const double u2[4]
 *                const double u3[4]
 *                double b_y1[4]
```

```
 *                double y2[4]
 *                double y3[4]
 *                double y4[4]
 * Return Type  : void
 */
void replace_mixed_row_column_ops(const double u1[4], const double u2[4],
                                  const double u3[4], double b_y1[4],
                                  double y2[4], double y3[4], double y4[4])
{
  double b_u1[4];
  double b_u2[4];
  double b_u3[4];
  double dv[4];
  double dv1[4];
  b_u1[0] = u1[0];
  b_u2[0] = u2[0];
  b_u3[0] = u3[0];
  b_u1[1] = u1[2];
  b_u2[1] = u2[2];
  b_u3[1] = u3[2];
  b_u1[2] = u1[1];
  b_u2[2] = u2[1];
  b_u3[2] = u3[1];
  b_u1[3] = u1[3];
  b_u2[3] = u2[3];
  b_u3[3] = u3[3];
  Child_Op(b_u1, b_u2, b_u3, dv, dv1);
  y2[0] = dv1[0];
  b_y1[0] = dv[0];
  y2[1] = dv1[2];
  b_y1[1] = dv[2];
  y2[2] = dv1[1];
  b_y1[2] = dv[1];
  y2[3] = dv1[3];
  b_y1[3] = dv[3];
  MyMul_CUSTOM_ROW((double *)&u1[0], (double *)&u2[0], &y3[0], 2U, 2U, 2U, 2U);
  MyAdd_CUSTOM_ROW((double *)&u1[0], (double *)&u3[0], &y4[0], 2U, 2U, 2U, 2U);
}

/*
```

MATLAB Function Replacement Using Coder.Replace

Code replacement libraries support the replacement of MATLAB functions with scalar and matrix inputs and outputs for built-in, complex, and fixed-point data types.

1. The coder_replace_fcn.m file contains this code:

```
function [y1, y2, y3] = coder_replace_fcn(u1, u2) %#codegen
%   Copyright 2012 The MathWorks, Inc.

y1 = myScalarFunction(u1(1),u1(2));
y2 = myMatrixFunction(u1, u2);
complexValue = u1 + 1i;
y3 = myComplexFunction(complexValue, u2(1));
end
```

```
%----------------------------------------
% Lookup this function for CRL replacement
% Function: myScalarFunction
% Inputs:  u1: double scalar
%          u2: double scalar
% Outputs: y1: double scalar
%----------------------------------------
function y1 = myScalarFunction(u1, u2)

% Trigger CRL lookup
coder.replace('-errorifnoreplacement');
y1 = sin(u1) + cos(u2);
end

%----------------------------------------
% Lookup this function for CRL replacement
% Function: myMatrixFunction
% Inputs:  u1: 2x2 double matrix
%          u2: 2x2 double matrix
% Outputs: y1: 2x2 double matrix
%----------------------------------------
function y1 = myMatrixFunction(u1, u2)

% Trigger CRL lookup
coder.replace('-errorifnoreplacement');
y1 = u1 + u2;
end


%----------------------------------------
% Lookup this function for CRL replacement
% Function: myComplexFunction
% Inputs:  u1: 2x2 complex double matrix
%          u2: double scalar
% Outputs: y1: 2x2 complex double matrix
%----------------------------------------
function y1 = myComplexFunction(u1, u2)

% Trigger CRL lookup
coder.replace('-errorifnoreplacement');
y1 = u1 + u2;
end
```

2. Configure the code generator to use a code replacement library **Coder Replace Examples**.

```
RTW.TargetRegistry.getInstance('reset');
cfg = coder.config('lib','ecoder',true);
cfg.CodeReplacementLibrary = 'Coder Replace Examples';
cfg.GenerateReport = false;
cfg.LaunchReport = false;
```

3. Set up the configuration parameters to build and define the function input type.

```
t = [1 2; 3 4];
```

4. Explore the code replacement table definition file, `crl_table_coder_replace.m`.

5. Compile the MATLAB program using the configuration parameters that point to the desired code replacement library and the example input class defined in step 3 as input parameters to the `codegen` command.

```
codegen('coder_replace_fcn', '-config', cfg, '-args', {t, t});
```

```
Code generation successful.
```

6. The `coder_replace_fcn.c` file contains this generated code.

```
file = fullfile('codegen','lib','coder_replace_fcn','coder_replace_fcn.c');
rtwdemodbtype(file,'#include "coder_replace_fcn.h','* File trailer for coder_replace_fcn.c',1,0)
```

```
#include "coder_replace_fcn.h"
#include "MyMath.h"

/* Function Definitions */
/*
 * Copyright 2012 The MathWorks, Inc.
 *
 * Arguments    : const double u1[4]
 *                const double u2[4]
 *                double *b_y1
 *                double y2[4]
 *                creal_T y3[4]
 * Return Type  : void
 */
void coder_replace_fcn(const double u1[4], const double u2[4], double *b_y1,
                       double y2[4], creal_T y3[4])
{
  creal_T dcv[4];
  *b_y1 = myScalarFunction_impl(u1[0], u1[1]);
  myMatrixFunction_impl(&y2[0], (double *)&u1[0], (double *)&u2[0]);
  dcv[0].re = u1[0];
  dcv[0].im = 1.0;
  dcv[1].re = u1[1];
  dcv[1].im = 1.0;
  dcv[2].re = u1[2];
  dcv[2].im = 1.0;
  dcv[3].re = u1[3];
  dcv[3].im = 1.0;
  myComplexFunction_impl(&y3[0], &dcv[0], u2[0]);
}

/*
```

For more information, see “Replace MATLAB Functions with Custom Code by Using the coder.replace Function” on page 73-79.

Build Information

For each entry in a code replacement table, you can specify build information such as the following, for replacement functions:

- Header file dependencies
- Source file dependencies
- Additional include paths

- Additional source paths
- Additional link flags

Additionally, you can specify `RTW.copyFileToBuildDir` to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. You can specify `RTW.copyFileToBuildDir` by setting it as the value of:

- Property `GenCallback` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`.
- Argument `genCallback` in a call to `registerCFunctionEntry`, `registerCOperationEntry`, or `registerCSemaphoreEntry`.

Note: The `.m` scripts in this example are configured for code generation only because the implementations for the replacement functions are not provided.

For more information on specifying build information for replacement code, see “Develop a Code Replacement Library” on page 73-13.

Cleanup

Refresh the library registration information within the current MATLAB session.

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

Embedded Coder Support Package for Linux Applications

- “Support Package Installation” on page 74-2
- “Setup Linux Target Computer” on page 74-3
- “Build Simulink Model and Deploy Application” on page 74-6
- “Start and Stop Deployed Applications” on page 74-9
- “Tune Parameters and Measure Signal Data for Deployed Applications” on page 74-11
- “Analyze Logged Messages” on page 74-13
- “External Mode Simulation of Deployed Applications” on page 74-15

Support Package Installation

To install the Embedded Coder Support Package for Linux Applications, follow these steps:

1. On the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons** > **Get Hardware Support Packages**.
2. In the Add-On Explorer window, search for Embedded Coder Support Package for Linux Applications. Click install.
3. Sign in using the MathWorks account.
4. Accept the license agreement `MathWorks Auxillary Software License Agreement`.

Installation for the support package starts. After the installation is completed, you can exit the process or continue to setup the target. To setup the target:

- Click the **Setup Now** button.
- In the **Hardware Setup** window, enter the details of target such as `Target Name`, `IP Address`, `Username`, `Password`, and `Deployment Location` and click **Submit**.
- After the target setup is completed, a message appears and then click **Finish** button to complete and exit from the installation procedure.

See Also

`Linux Runtime Manager` | `linux.Targets` | `connect` | `setDeploymentLocation`

Related Examples

- "Setup Linux Target Computer" on page 74-3
- "Build Simulink Model and Deploy Application" on page 74-6

Setup Linux Target Computer

In this section...

"Target Computer requirements" on page 74-3
"Steps to setup target" on page 74-3

Target Computer requirements

To deploy Service Oriented Architecture applications on a target computer using the support package, following requirements have to be met.

- Linux machine that supports MATLAB installation. For more information, see https://in.mathworks.com/support/requirements/matlab-linux.html.
- Docker software version 20.10 or later needs to be installed.
- Zip and Unzip commands needs to be installed.
- SSH server needs to be in running state.
- Target computer username needs to have privileges to run docker commands. For more information, see https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user.

Note By default, port numbers 5555 and 5556 on target are used to communicate with the host. To modify the port numbers, update them in the `<Support-Package-root>/toolbox/coder/linux/host/ContainerConfig.json` file. You can get the support package root using the command:

```
matlabshared.supportpkg.getSupportPackageRoot
```

Port number 22255 is reserved for use on target and hence the port is not configurable.

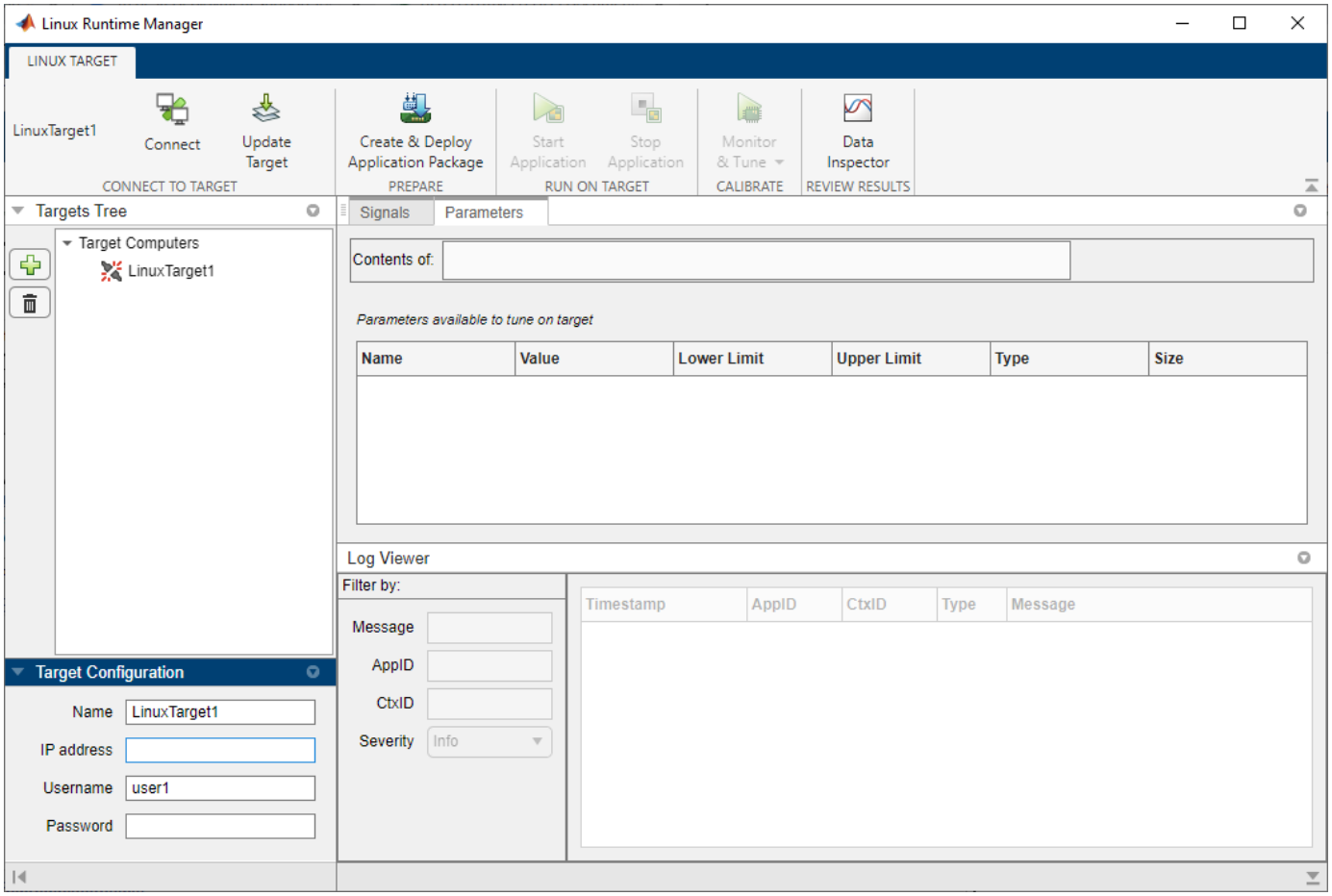
Steps to setup target

Use the Linux Runtime Manager application to connect, disconnect, add, remove, and update Linux target computers.

To connect to a target computer that is running the Linux operating system:

1. Open the Linux Runtime Manager.

   ```
   linux.RuntimeManager.open
   ```



2 In the **Targets Tree** pane, select **Target Computers** and click plus button to add a target computer. In **Target Configuration** section, enter details for the target computer.

programmatically:

```
tgs = linux.Targets();
tg = tgs.addTarget(<targetName>, <ipAddress>, <userName>, <password>);
```

3 For a newly added target computer, enter the full path of the location at which you want to deploy the application using this command:

```
tg.setDeploymentLocation(<Deployment location>);
```

4 To deploy DDS applications configured with `RTI Connext` vendor, set the value of environment variable **NDDSHOME_LINUX** on host to the `RTI Connext` software folder path on Linux target machine.

```
setenv('NDDSHOME_LINUX', '<Path for the software>')
```

For example:

```
setenv('NDDSHOME_LINUX', '/home/user/RTIDDS/glnxa64/rti_connext_dds-6.0.1_gcc7.3.0')
```

5 Select a target and click the **Linux Target** > **Connect To Target** > **Connect** button.

Programmatically:

```
tg.connect();
```

Note This step prompts you to provide a deployment location for the target computer if one is already not set.

6. To update software on the target computer, select the target computer and click **Linux Target** > **Connect To Target** > **Update Target**.

   Programmatically:

   ```
   tg.updateTarget();
   ```

7. To disconnect the target computer, select the target computer and click **Linux Target** > **Connect To Target** > **Disconnect**.

   Programmatically:

   ```
   tg.disconnect();
   ```

8. To completely remove the target computer files and its storage, use this command:

   ```
   tg.uninstallTarget();
   ```

9. To remove a target computer, select the target computer and click the delete button. This action removes the target computer from the **Targets Tree**, but the applications deployed to the target computer and its docker image still exist in storage.

   Programmatically:

   ```
   tgs.removeTarget(<Target Name>);
   ```

Using this application, you can connect to multiple target computers at the same time.

See Also

`linux.Targets` | `addTarget` | `connect` | `setDeploymentLocation`

Related Examples

- "Build Simulink Model and Deploy Application" on page 74-6

Build Simulink Model and Deploy Application

After connecting the development computer to the target computer, build the Simulink model and deploy the standalone Linux application to the Linux environment.

- For the Simulink model, select **Hardware Implementation** > **Hardware board** as `Embedded Coder Linux Docker Container`. This is mandatory for the model to deploy it using support package.

  Note Building a model with **Hardware board** set to `Embedded Coder Linux Docker Container` generates the `.mldatx` file.

- Click the **Linux Target** > **Prepare** > **Create & Deploy Application Package**.
- If you have an application package (`.mldatx`) file already created, select it. Or you can select the Simulink model.

  Programmatically:

  To create application package from Simulink model:

  ```
  linux.createApplicationPackage(<modelName>);
  ```

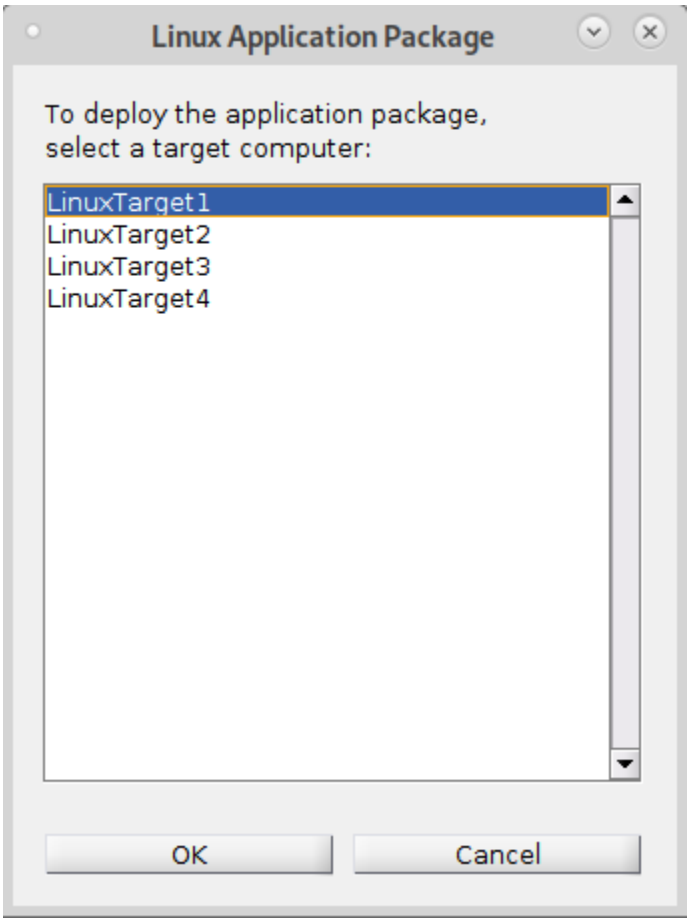
  To deploy the application package:

  ```
  tg.deployApplicationPackage(<NameOfSoftwarePackage>);
  ```

- The Linux Runtime Manager builds the model, creates the application package, and deploys the package.

If there is an application package (`.mldatx`) file available, you can also deploy it on target in the following way:

- Right-click the application package and select **Open**. The following window opens.



- Choose a target and click the **OK** button. The selected application gets deployed on the target.

The application name is displayed under the target computer. To view the list of measurable signals and tunable parameters, select the deployed application.

Alternatively, use these commands.

To view the available signals:

```
availableSignals = tg.getMeasurements(<applicationName>);
```

To view the tunable parameters:

```
availableParameters = tg.getParameters(<applicationName>);
```

Deploy DDS Blockset Models

To deploy DDS Blockset model setup target and follow the steps:

- To deploy DDS applications configured with `RTI Connext` vendor, set the value of environment variable **NDDSHOME_LINUX** on host to the `RTI Connext` software folder path on Linux target machine.

  ```
  setenv('NDDSHOME_LINUX', '<Path for the software>')
  ```

  For example:

```
setenv('NDDSHOME_LINUX', '/home/user/RTIDDS/glnxa64/rti_connext_dds-6.0.1_gcc7.3.0')
```

- On the Linux Runtime manager, click **Connect** button. If the target setup is not done, a pop-up opens to fill the deployment location.

  If the docker container is already present, delete it using this command:

  ```
  docker rm linux-container
  ```

- Click the **Linux Target** > **Prepare** > **Create & Deploy Application Package** and select the DDS Blockset model.

  Note Deployment of DDS Blockset models with referenced models is not supported.

  This will generate `MLDATX` file for the selected model and deploys it on the target.

- Select the application and click on **Start Application** to launch the application on target.

You can also deploy DDS Blockset models by converting them to Adaptive AUTOSAR models by using the function `linux.utils.migrateDds2Adaptive`.

See Also

`linux.utils.migrateDds2Adaptive` | `deployApplicationPackage`

Related Examples

- “Setup Linux Target Computer” on page 74-3
- “Start and Stop Deployed Applications” on page 74-9

Start and Stop Deployed Applications

The deployed applications are shown under the target computer name in the **Targets Tree**. Select an application and:

- To start the application, click **Linux Target** > **Run On Target** > **Start Application**.

  Programmatically:

  ```
  tg.startApplication(<applicationName>);
  ```

- To pause the application, click the **Linux Target** > **Run On Target** > **Pause Application**.

  Programmatically:

  ```
  tg.pauseApplication(<applicationName>);
  ```

- To resume the application, click the **Linux Target** > **Run On Target** > **Resume Application**.

  Programmatically:

  ```
  tg.resumeApplication(<applicationName>);
  ```

- To stop the application, click the **Linux Target** > **Run On Target** > **Stop Application**.

  Programmatically:

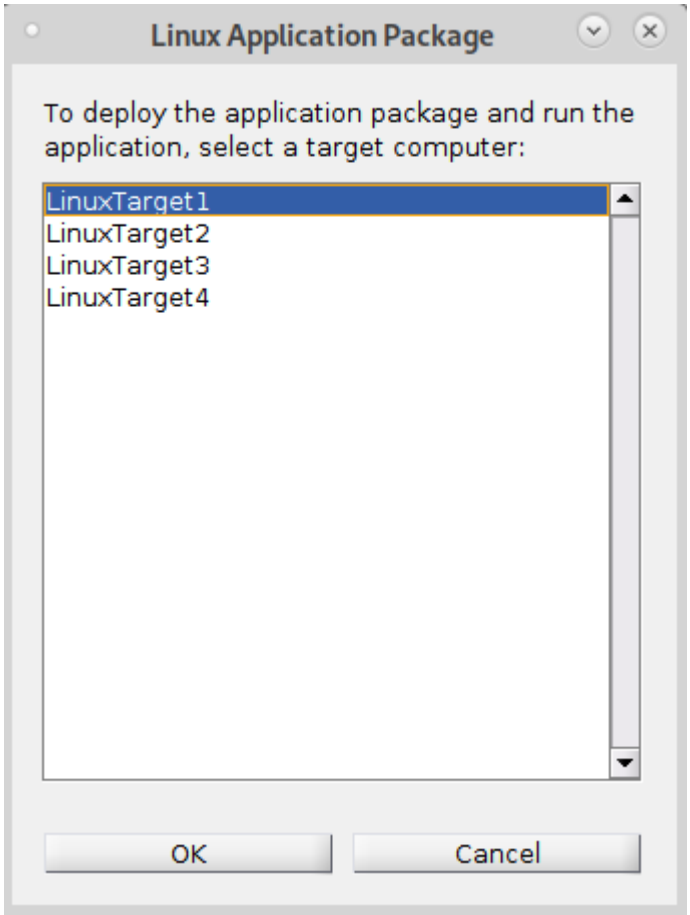
  ```
  tg.stopApplication(<applicationName>);
  ```

- Click the delete button to delete the application.

  Programmatically:

  ```
  tg.removeApplicationPackage(<applicationName>);
  ```

You can also start running the application by right-clicking the application package (`.mldatx`) file and selecting the **Run** option. This opens a window.



Choose a target to run the application and click **OK**.

See Also

`stopApplication` | `startApplication` | `removeApplicationPackage`

Related Examples

- "Tune Parameters and Measure Signal Data for Deployed Applications" on page 74-11

Tune Parameters and Measure Signal Data for Deployed Applications

After you deploy an application to a Linux target environment, you can view the list of measurable signals and tunable parameters in **Signals** and **Parameters** tabs respectively by selecting the application in the target tree. To set up the data:

- Select the signals to measure and move them from **Signals available on target** to **Signals to Instrument**.

  Programmatically:

  ```
  tg.selectSignalsToInstrument(<applicationName>, <SignalNamesArray>);
  ```
- Remove signals from **Signals to Instrument** by selecting them and clicking the delete button.

  Programmatically:

  ```
  tg.removeSignalsFromInstrument(<applicationName>, <SignalNamesArray>);
  ```
- If the application is in running state, stop it by clicking the **Linux Target** > **Run On Target** > **Stop Application**.
- Start tuning the parameters clicking the **Linux Target** > **Calibrate** > **Monitor & Tune**.

  Programmatically:

  ```
  tg.startMonitorAndTune(<applicationName>);
  ```

  Note If the application is in running state, you can start the calibration by clicking **Linux Target** > **Calibrate** > **Monitor & Tune** > **Connect**.

  Programmatically:

  ```
  tg.startMonitorAndTune(<applicationName>,false);
  ```

- To see a plot of the selected signals, open the `Simulation Data Inspector` by selecting **Linux Target** > **Review Results** > **Data Inspector**.

  Programmatically:

  ```
  Simulink.sdi.view
  ```
- You can add more signals or remove signals for measurement.
- To view an updated list of signals to appear on Simulink Data Inspector, click the **Restart Streaming** button.

  Programmatically:

  ```
  tg.restartStreaming(<applicationName>);
  ```
- On the **Parameters** tab, you can read parameter values.

  Programmatically:

  To read a specific parameter:

  ```
  tg.readParameter(<applicationName>, <paramName>);
  ```

To read the parameters available:

```
tg.readParameters(<applicationName>);
```

- To tune parameters, select **Parameters** tab and specify a value in the value field of the parameter of interest.

  Programmatically:

  ```
  tg.writeParameter(<applicationName>, <paramName>, <paramValue>);
  ```

  The `paramValue` accepts multiple data types to support various kinds of parameters.

  Note To enable calibration for AUTOSAR Adaptive models, configure the XCP communication interface by setting the **Code Generation** > **AUTOSAR Code Generation** > **XCP Slave Configuration** > **Transport layer** to `XCP On TCP/IP`.

  In Simulink Data Inspector, you can observe changes to signal data as you tune parameters.

- To stop streaming of signal data:

  ```
  tg.stopStreaming(<applicationName>);
  ```

- You can stop signal measurement and parameter tuning by clicking **Linux Target** > **Calibrate** > **Stop Monitor & Tune**.

  Programmatically:

  ```
  tg.stopMonitorAndTune(<applicationName>);
  ```

See Also

`startStreaming` | `stopStreaming` | `startMonitorAndTune` | `writeParameter`

Related Examples

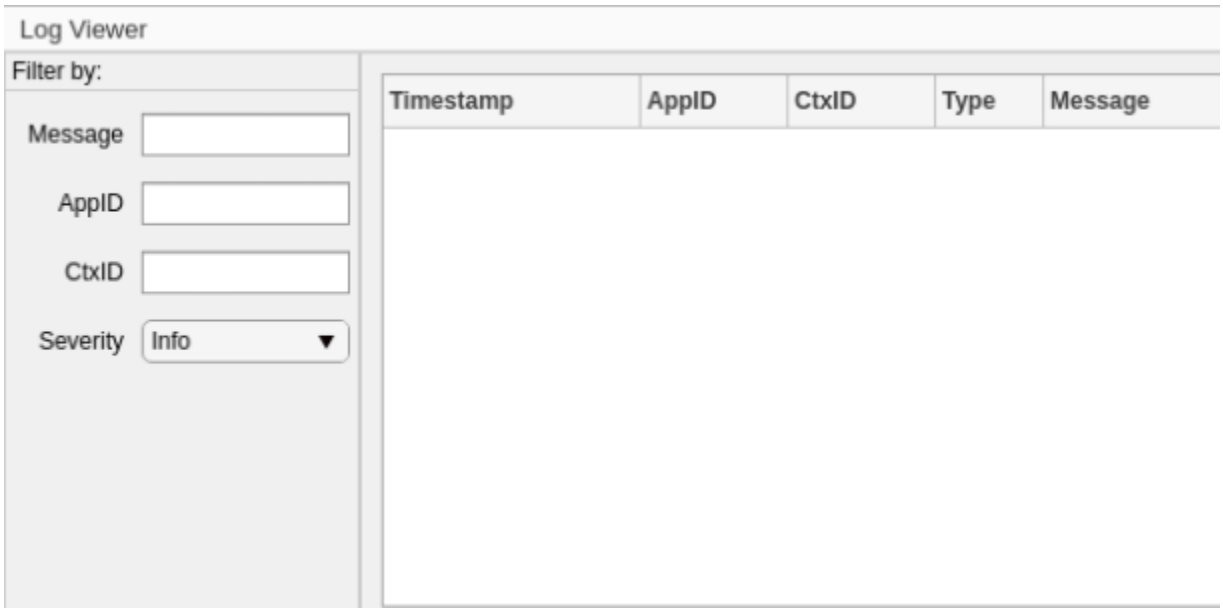
- "Start and Stop Deployed Applications" on page 74-9
- "Analyze Logged Messages" on page 74-13

Analyze Logged Messages

After you start an application, you can view the log messages of the application and custom log messages from the Linux target computer in the **Log Viewer** panel. Each logged message contains these fields:

- Time when message is logged - `Timestamp`
- Unique application id - `AppId`
- Context id of the message. An application can have multiple contexts - `CtxId`
- Type of the message. Possible types are `Error`, `Warn`, `Fatal`, `Info`, `Verbose`, and `Debug` - `Type`
- Message text - `Message`

To filter the messages to view on the application, use **Filter by:** column.



To create an object for the log messages from the command line, use this command:

```
logObj = linux.LogMessages(<targetName>);
```

From the log messages object, you can get the required messages by using this command:

```
msgs = logObj.get('MessageFilter', <message>, 'AppIdFilter',
<appId>, 'CtxIdFilter', <ctxId>, 'SeverityFilter', <severityLevel>)
```

To filter messages based on applications:

```
msgs_appName = logObj('AppIdFilter',<appId>);
```

To filter messages based on context:

```
msgs_ctx = logObj('AppIdFilter',<appId>,'CtxIdFilter',<ctxId>);
```

To filter messages based on severity:

```
msgs_severity = logObj('SeverityFilter',<SeverityLevel>);
```

See Also

`linux.LogMessages` | `linux.LogMessages.get`

Related Examples

- “Start and Stop Deployed Applications” on page 74-9

External Mode Simulation of Deployed Applications

You can use the **Linux Runtime Manager** app to deploy the models that you configure for external mode. The app also supports external mode simulation by establishing a communication channel between the model running on host computer and the application running in a docker container on Linux target.

To understand more about external mode simulation, see “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2. To know more about deploying a model on target computer, see “Build Simulink Model and Deploy Application” on page 74-6.

To perform external mode simulation of the deployed application follow these steps.

1. Open an AUTOSAR adaptive model.
2. Set the **Hardware board** configuration parameter in the **Hardware Implementation** to `Embedded Coder Linux Docker Container`.
3. Set the **Transport layer** parameter in the **XCP Slave Configuration** pane of the **Code Generation** > **AUTOSAR Code Generation** to `XCP On TCP/IP`.
4. Specify the IP address of the target machine and port number for XCP communication in the **Code Generation** > **AUTOSAR Code Generation** > **XCP Slave Configuration** section.
5. Enable the **External mode** option in the **Interface** pane of **Code Generation** section.
6. To log signals, select the signals you want to log and click the **Log Signals** in the **Prepare** pane of the **Simulation** section.
7. Open the **Linux Runtime Manager** from the Simulink apps gallery. Alternatively, enter this command at the MATLAB Command Window.

   ```
   linux.RuntimeManager.open
   ```
8. Deploy the model on the Linux target machine. See “Build Simulink Model and Deploy Application” on page 74-6 for more information.
9. Select the model in the **Linux Runtime Manager** and click **Monitor & Tune** button to start the application and an external mode simulation.
10. Tune the parameters using the Simulink model. You can use Dashboard blocks like Slider, Knob, Radio Button, Push Button etc. or directly changing block parameters. You can view the signals directly in Simulink model using Dashboard blocks like Gauge, Scope etc. You can also view the signal in Signal Data Inspector.

Select the application and click **Stop Monitor & Tune** to stop the external mode simulation. Or select the application and click **Stop Application** button to stop application.

You can also deploy a DDS Blockset model configured for external mode. For a DDS Blockset model, you can enter the target details in the **MEX-file arguments** section of the **External mode configuration** tab in the **Interface** section of **Code Generation** tab. For more information, see MEX-file arguments.

See Also

Linux Runtime Manager

Related Examples

- “External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling” on page 60-2
- “Build Simulink Model and Deploy Application” on page 74-6

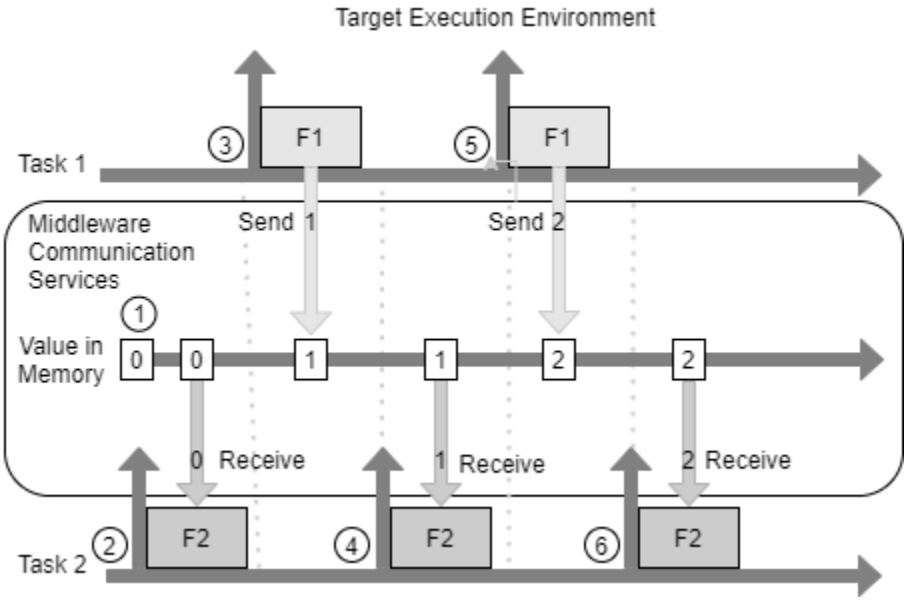
Data Communication Methods

Sample Target Environment Service Implementations of Data Communication Methods

Direct-Access Data Communication

To maximize performance of component code or if mutual exclusion is inherent in the component model design, use direct-access data communication. Generated function code communicates with other functions directly by using memory that platform services manage for the target execution environment. Platform services maintain memory persistence across time during one start and shutdown cycle.

This figure shows two generated callable functions that use direct-access data communication. The numeric callouts in the figure are keyed to the list that follows the figure.



1. At startup, the platform service allocates a buffer and initializes the buffer to 0. The buffer is persistent across time.
2. The service calls function F2 to receive data. The function reads value 0 from the buffer without safeguards.
3. The service calls function F1 to send data. The function writes the value 1 to the buffer. Direct access communication assumes that functions F1 and F2 do not access the buffer concurrently and a safeguard is not needed.
4. The service calls function F2. During its second execution, function F2 receives the new data. The function reads the value 1 from the buffer.
5. The service calls function F1. During its second execution, function F1 sends the value 2 to the buffer.
6. The service calls function F2. During its third execution, function F2 receives the value 2 by reading the new value from the buffer.

Functions F1 and F2 have direct access to memory for the duration of function execution. The platform service assumes that the functions do not access the buffer concurrently and applies no data concurrency safeguards.

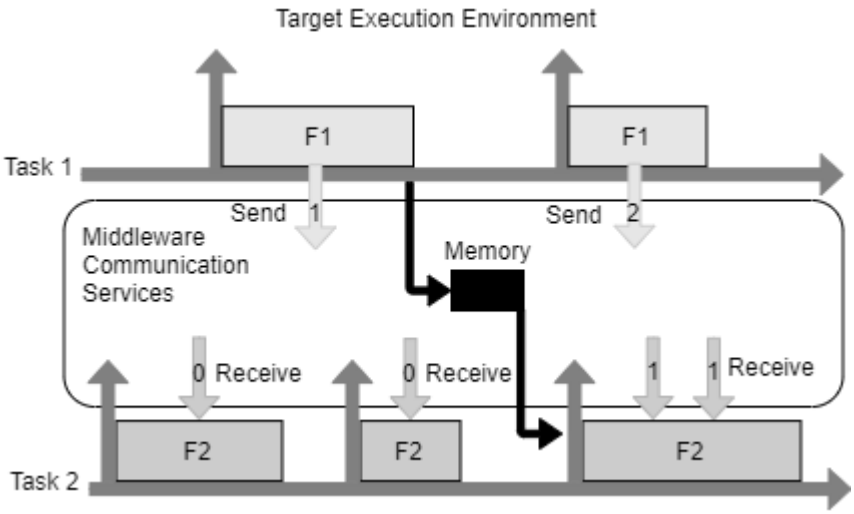
Outside-Execution Data Communication

To favor memory optimizations over data freshness or when the platform service code is autogenerated based on full knowledge of your application, use outside-execution data communication. The platform communicates data with other functions outside (before and after) function execution. As component model designer, you accept how the platform applies safeguards for concurrent data access. For the duration of its execution, a function can access memory.

The platform communication services:

- Allocate buffer and, at startup, initialize the buffer.
- Lock the buffer at the beginning of function execution.
- Unlock the buffer at the end of function execution.
- Switch buffers to maintain memory exclusivity for function execution.
- Maintain memory persistence during function execution within the context of a power cycle.

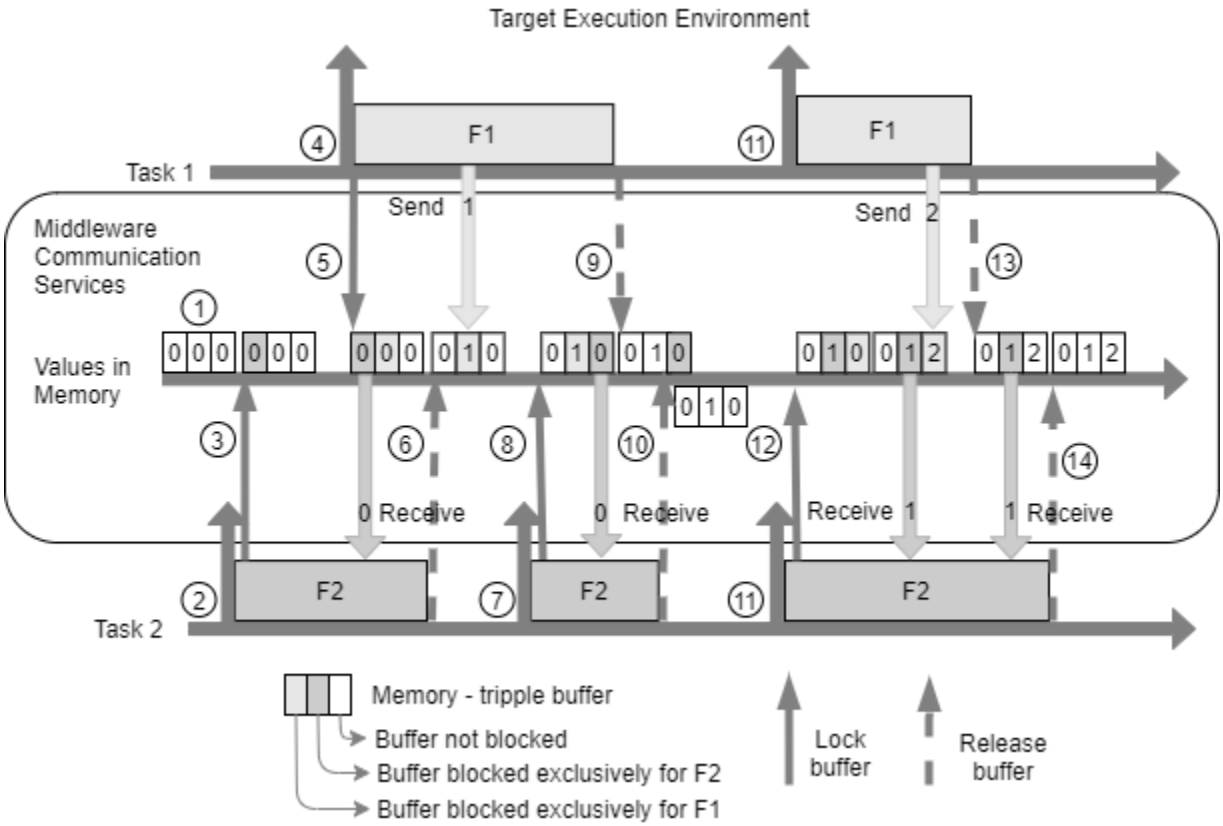
This figure shows platform communication services that use an outside-execution data communication algorithm.



Generated callable functions F1 and F2 send and receive data while executing. The platform services communicate the data before and after function execution.

- Data that function F1 sends becomes accessible to function F2 after F1 completes execution.
- The platform service locks the buffer and holds the data that function F2 receives constant while F2 executes.
- F2 can access the received data as needed until execution completes.

This figure shows how a platform service might implement outside-execution communication by using a triple-buffer algorithm. The numeric callouts in the figure are keyed to the list that follows the figure.



With this algorithm, a platform service:

1. Allocates three buffers and initializes the buffers with value 0.
2. Calls generated callable function F2.
3. Locks the first buffer for F2. F2 can read from the locked buffer as needed while executing. Because data has not been sent to the buffer, the function receives the value 0.
4. Calls generated callable function F1.
5. Locks the second buffer for F1. F1 can write to the locked buffer as needed while executing. The function writes the value 1 to the buffer.
6. When F2 completes execution, unlocks the first buffer.
7. Calls F2.
8. Locks the third buffer for F2. Because the second buffer is locked for F1, the value 1 in that buffer is not accessible to F2. F2 receives the value that is in the third buffer, which is 0.
9. When F1 completes execution, unlocks the second buffer.
10. When F2 completes execution, unlocks the third buffer.
11. Calls F1 and F2.
12. Locks the second and third buffers. The service locks the second buffer so that F2 can receive the freshest data, which is the value 1. The service locks the third buffer so that F1 can write a new value to that buffer. F1 writes the value 2 to the third buffer.
13. When F1 completes execution, unlocks the third buffer.

14 When F2 completes execution, unlocks the second buffer. Because the service retains the lock on the second buffer until F2 completes execution, F2 receives the value 1 a second time.

During-Execution Data Communication

To favor data freshness over memory usage or when there is no aggregate service generation phase, use during-execution data communication. The platform service communicates data with other functions immediately during function execution. As component model designer, you accept how the service applies safeguards for concurrent data access. For the duration of its execution, a function can access memory. A function maintains value coherence during execution by using a local buffer.

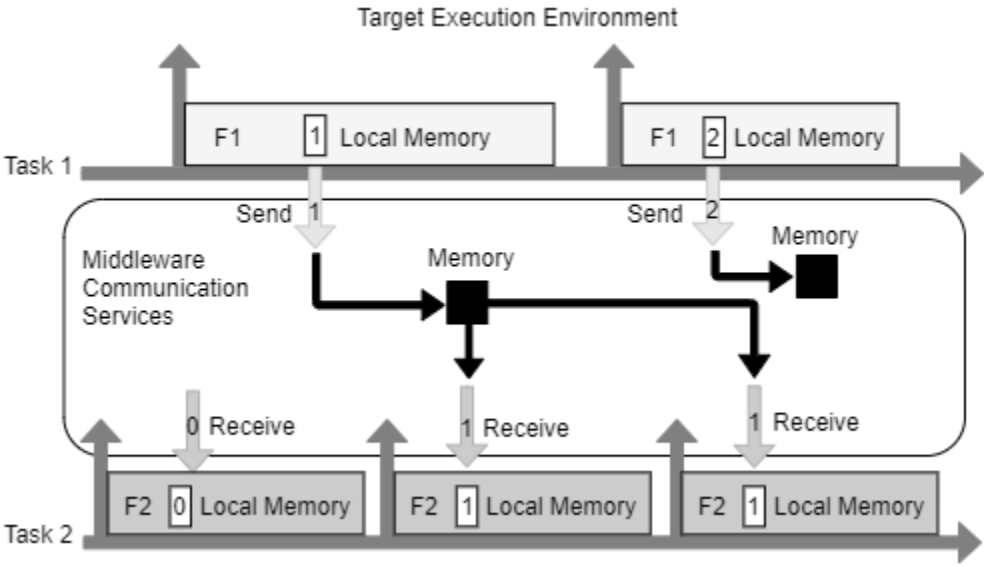
The platform communication services:

- Allocate a buffer and initializes the buffer to 0 at startup.
- Lock the buffer during a receive or send operation.
- Unlock the buffer immediately after reading a value from or writing a value to the buffer.
- Switch buffers to maintain memory exclusivity for function execution.
- Do not maintain memory persistence during a power cycle or during function execution.

During-execution communication can be nonblocking or blocking.

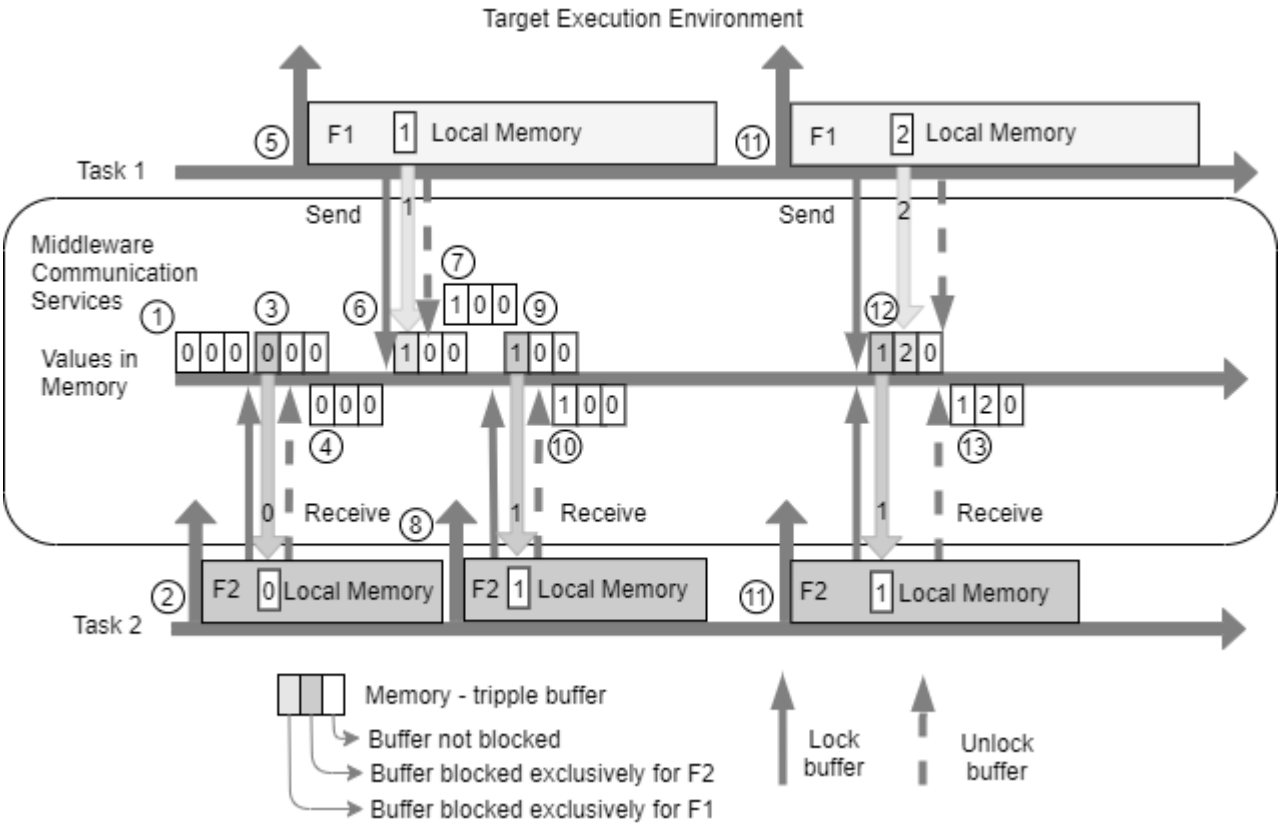
Nonblocking During-Execution Data Communication

This figure shows platform communication services that use a nonblocking during-execution data communication algorithm.



Data that generated callable function F1 sends becomes accessible to generated callable function F2 during the execution of F1. The receiver, function F2, uses local memory to hold the value constant during execution.

This figure shows how a platform service might implement nonblocking during-execution communication by using a triple-buffer approach. The numeric callouts in the figure are keyed to the list that follows the figure.

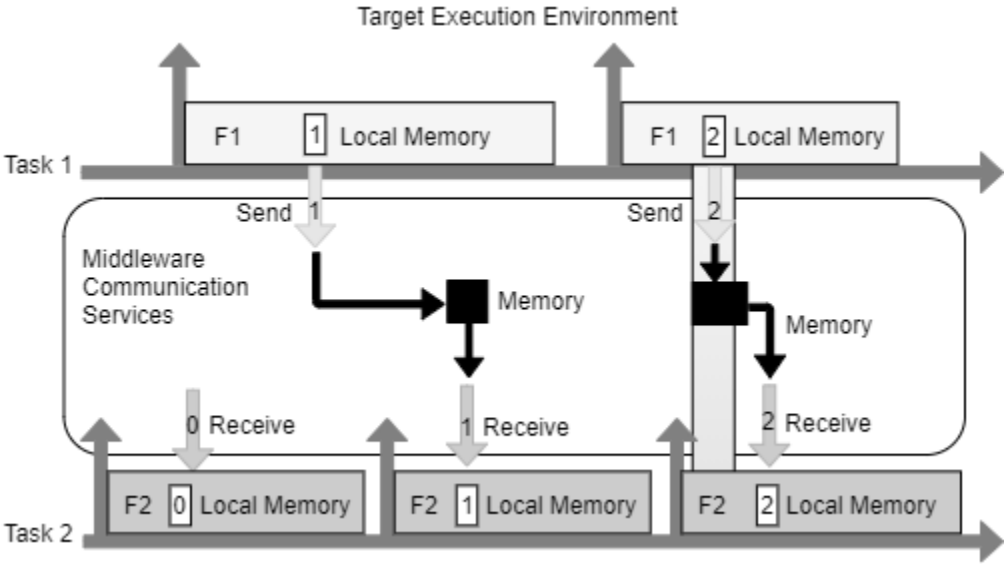


With this algorithm, a platform service:

1. Allocates three buffers and initializes the buffers with value 0.
2. Calls generated callable function F2.
3. Locks the first buffer for F2. F2 reads data from the buffer immediately.
4. When the read operation is complete, unlocks the buffer immediately. F2 cannot access the buffer again during execution. For additional read operations, F2 must preserve the value in local memory.
5. Calls generated callable function F1.
6. Locks the first buffer for F1. F1 writes the value 1 to the buffer immediately.
7. When the write operating is complete, unlocks the buffer immediately.
8. Calls F2.
9. Locks the buffer that contains the freshest data. The first buffer contains the freshest data, which is the value 1.
10. When the write operation is complete, unlocks the buffer immediately.
11. Calls F1 and F2.
12. For F2, locks the buffer that contains the freshest data. The first buffer contains the freshest data, which is the value 1. For F1, locks the buffer that contains the stalest data. The second buffer contains the stalest data, which is the value 2.
13. When the read and write operations are complete, unlocks the buffers immediately.

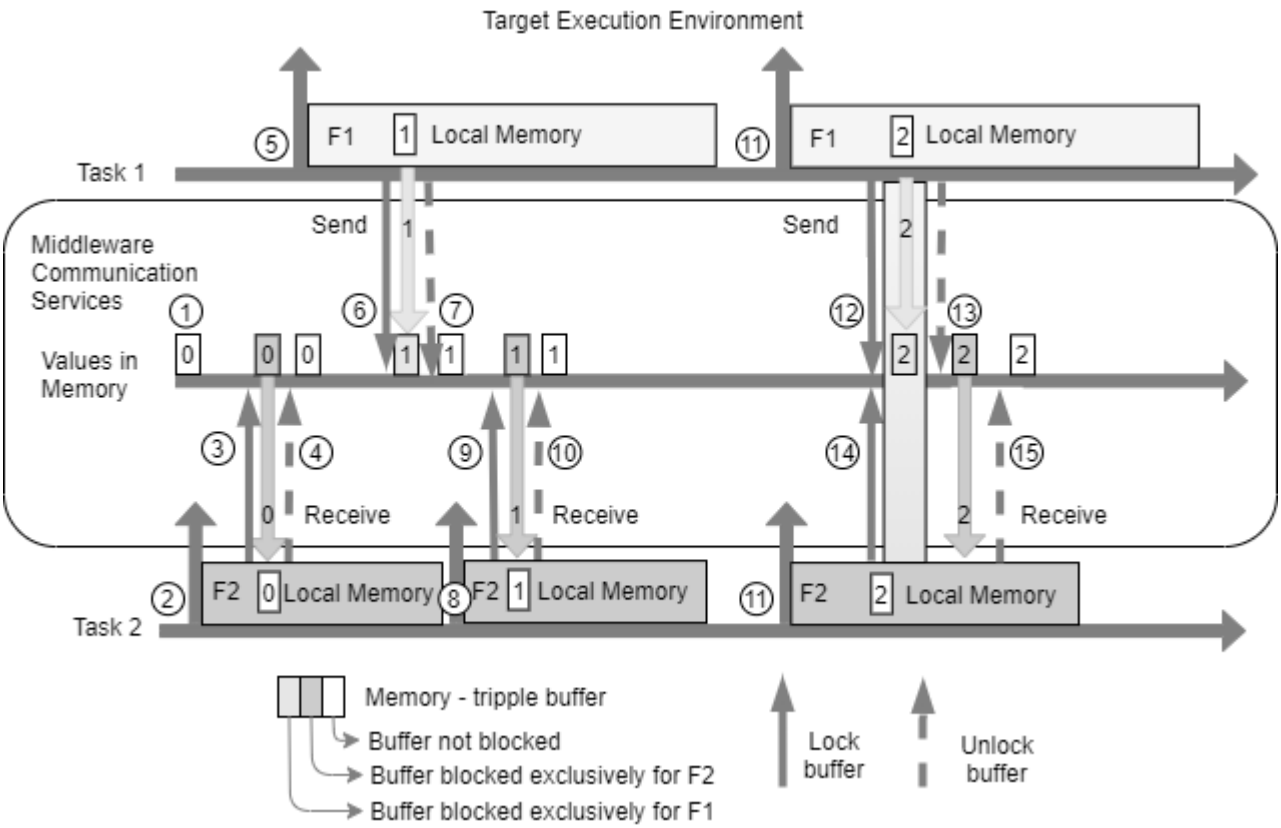
Blocking During-Execution Data Communication

This figure shows platform communication services that use a blocking during-execution data communication algorithm. A blocking algorithm must avoid live-lock and dead-lock concurrency issues.



Data that the generated callable function F1 sends becomes accessible to generated callable function F2 during the execution of F1. The sender, in this case F1, might block the receiver. The receiver, function F2, uses local memory to hold the value constant during execution. The service might maintain memory persistence for a power cycle.

This figure shows how service might implement blocking during-execution communication by using a single-buffer approach. The numeric callouts in the figure are keyed to the list that follows the figure.



With this algorithm, the a service:

1. Allocates a buffer and initializes the buffer with value 0.
2. Calls generated callable function F2.
3. Locks the buffer for F2. F2 reads the value in the buffer, 0, until a send occurs. For multiple read operations, F2 must preserve the value in local memory.
4. When a read operation is complete, unlocks the buffer immediately.
5. Calls generated callable function F1.
6. Locks the buffer for F1. F1 writes the value 1 to the buffer immediately.
7. When the write operation is complete, unlocks the buffer immediately.
8. Calls F2.
9. Locks the buffer for F2. F2 reads the value 1 from the buffer.
10. When the read operation is complete, unlocks the buffer immediately.
11. Calls F1 and F2.
12. Blocks F2 and locks the buffer for F1. F1 writes the value 2 to the buffer.
13. When the write operation is complete, unlocks the buffer immediately.
14. Unblocks F2 and locks the buffer for F2. F2 reads the value 2 from the buffer.
15. When the read operation is complete, unlocks the buffer immediately.

Performance

Optimizations for Generated Code in Simulink Coder

- “Increase Code Generation Speed” on page 76-3
- “Control Compiler Optimizations” on page 76-6
- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Control Memory Allocation for Time Counters” on page 76-10
- “Execution Profiling for Generated Code” on page 76-11
- “Optimize Generated Code by Combining Multiple for Constructs” on page 76-13
- “Subnormal Number Execution Speed” on page 76-16
- “Remove Code for Out-of-Range Floating Point to Integer Conversions” on page 76-20
- “Remove Code That Maps NaN to Integer Zero” on page 76-23
- “Disable Nonfinite Checks or Inlining for Math Functions” on page 76-26
- “Fold Expressions” on page 76-31
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35
- “Inline Invariant Signals” on page 76-38
- “Inline Numeric Values of Block Parameters” on page 76-41
- “Configure Loop Unrolling Threshold” on page 76-46
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 76-48
- “Generate Target Optimizations Within Algorithm Code” on page 76-51
- “Remove Code for Blocks That Have No Effect on Computational Results” on page 76-52
- “Eliminate Dead Code Paths in Generated Code” on page 76-56
- “Floating-Point Multiplication to Handle a Net Slope Correction” on page 76-59
- “Use Conditional Input Branch Execution” on page 76-62
- “Optimize Generated Code for Complex Signals” on page 76-66
- “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” on page 76-68
- “Speed Up Matrix Operations in Code Generated from a MATLAB Function Block” on page 76-71
- “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block” on page 76-75
- “Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block” on page 76-78
- “Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” on page 76-82
- “Optimize Memory Usage for Time Counters” on page 76-84
- “Optimize Generated Code Using Boolean Data for Logical Signals” on page 76-89
- “Reduce Memory Usage for Boolean and State Configuration Variables” on page 76-92
- “Customize Stack Space Allocation” on page 76-93
- “Optimize Generated Code Using memset Function” on page 76-95

- “Vector Operation Optimization” on page 76-99
- “Enable and Reuse Local Block Outputs in Generated Code” on page 76-102
- “Generate SIMD Code from Simulink Blocks” on page 76-105
- “Optimize Code for Reduction Operations by Using SIMD” on page 76-110

Increase Code Generation Speed

In this section...

"Build a Model in Increments" on page 76-3
"Build Large Model Reference Hierarchies in Parallel" on page 76-3
"Minimize Memory Requirements During Code Generation" on page 76-4
"Generate Only Code" on page 76-4
"Suppress Creation of a Code Generation Report" on page 76-4

The amount of time it takes to generate code for a model depends on the size and configuration settings of the model. For instance, if you are working with a large model, it can take awhile to generate code. To decrease the amount of time for code generation of a model, try one or more of the following methods:

- Build a model in increments
- Build large model reference hierarchies in parallel
- Minimize memory requirements during code generation
- Generate only code
- Disable the creation of a code generation report

Build a Model in Increments

You can use the `slbuild` command to build a model and generate code. By default, when rebuilding a model, `slbuild` provides an incremental model build, which only rebuilds a model or submodels that have changed since the most recent model build. Incremental model build saves code generation time. Use the **Rebuild** parameter on the **Model Referencing** pane to change the method that Simulink uses to determine when to rebuild code for referenced models. For more information on the **Rebuild** parameter, see Rebuild.

Build Large Model Reference Hierarchies in Parallel

In a parallel computing environment, whenever conditions allow, you can increase the speed of code generation and compilation by building models containing large model reference hierarchies in parallel. For example, if you have Parallel Computing Toolbox software, you can distribute code generation and compilation for each referenced model across the cores of a multicore host computer. If you have MATLAB Parallel Server software, you can distribute code generation and compilation for each referenced model across remote workers in your MATLAB Parallel Server configuration.

The execution speed improvement realized by using parallel builds for referenced models depends on several factors, including:

- How many models can be built in parallel for a given model referencing hierarchy
- The size of the referenced models
- Parallel computing resources such as the number of local and remote workers available
- The hardware attributes of the local and remote machines (amount of RAM, number of cores, and so on)

For more information, see "Reduce Build Time for Referenced Models by Using Parallel Builds".

Minimize Memory Requirements During Code Generation

Models that have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow code generation. The code generator copies this data to the *model*.rtw file. The *model*.rtw file is a partial representation of the model that the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language (for example, C). The Target Language Compiler (TLC) is an integral part of the code generator. The code generator copies parameters and data into *model*.rtw, whether they originate in the model or come from variables or objects in a workspace.

You can improve code generation speed by specifying the maximum number of elements that data vectors can have for the code generator to copy this data to *model*.rtw. When a data vector exceeds the specified size, the code generator places a reference key in *model*.rtw. The TLC uses this key to access the data from Simulink and format it into the generated code. Reference keys result in maintaining only one copy of large data vectors in memory.

The default value above which the code generator uses reference keys in place of actual data values is 10 elements. You can verify this value. In the Command Window, type the following command:

```
get_param(0, 'RTWDataReferencesMinSize')
```

To set the threshold to a different value, in the Command Window, type the following set_param function:

```
set_param(0, 'RTWDataReferencesMinSize', <size>)
```

Provide an integer value for size that specifies the number of data elements above which the code generator uses reference keys in place of actual data values.

Generate Only Code

You can increase code generation speed by specifying that the build process generate code and a makefile, but not invoke the make command. When the code generator invokes the make command, the build process takes longer because the code generator generates code, compiles code, and creates an executable file.

On the **Code Generation** pane in the Model Configuration Parameters dialog box, you can specify that the build process generate only code by selecting the **Generate code only** parameter. You can specify that the code generation process build a makefile by selecting the **Configuration Parameters** > **Code Generation** > **Build process** > **Makefile configuration** > **Generate makefile** parameter.

Suppress Creation of a Code Generation Report

You can speed up code generation by not generating a code generation report as a part of the build process. To disable the creation of a code generation report, on the **Code Generation** > **Report** pane, clear the **Create code generation report** parameter. To generate a code generation report after the build process, see “Generate Code Generation Report” on page 54-5.

See Also

Related Examples

- Enable parallel model reference builds
- MATLAB worker initialization for builds
- “Reduce Build Time for Referenced Models by Using Parallel Builds” on page 59-51

Control Compiler Optimizations

To control compiler optimizations for a makefile build at the GUI level, use the **Compiler optimization level** parameter. The **Compiler optimization level** parameter provides

- Target-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)`, which allow you to easily toggle compiler optimizations on and off during code development
- The value `Custom` for entering custom compiler optimization flags at the Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to build process make commands

The default setting is `Optimizations off (faster builds)`. Selecting the value `Custom` enables the **Custom compiler optimization flags** field, in which you can enter custom compiler optimization flags (for example, `-O2`).

If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

For a rapid simulation that uses the MinGW compiler:

- If `RTWCompilerOptimization` is set to `'on'`, the build process customizes compiler optimization to minimize run time. The build process ignores the `BuildConfiguration` and `CustomToolchainOptions` settings.
- If `RTWCompilerOptimization` is set to `'custom'` or `'off'`, the build process uses `BuildConfiguration` and `CustomToolchainOptions` settings. The build process ignores the `RTWCompilerOptimization` setting.

See Also

Related Examples

- “Template Makefiles and Make Options” on page 59-36
- “Configure a System Target File” on page 50-2
- “Support Compiler Optimization Level Control” on page 94-65

Design Techniques to Optimize Models for Efficient Code Generation

Use the Model Advisor to Optimize a Model

You can use the Model Advisor to analyze a model for code generation and identify aspects of your model that impede production deployment or limit code efficiency. You can select from a set of checks to run on a model's current configuration. The Model Advisor analyzes the model and generates check results providing suggestions for improvements in each area. Most Model Advisor diagnostics do not require the model to be in a compiled state; those that do are noted.

Before running the Model Advisor, select the target you plan to use for code generation. The Model Advisor works most effectively with ERT and ERT-based system target files.

Use the following examples to investigate optimizing models for code generation using the Model Advisor:

- `rtwdemo_advisor1`
- `rtwdemo_advisor2`
- `rtwdemo_advisor3`

Note Example models `rtwdemo_advisor2` and `rtwdemo_advisor3` require Stateflow and Fixed-Point Designer software.

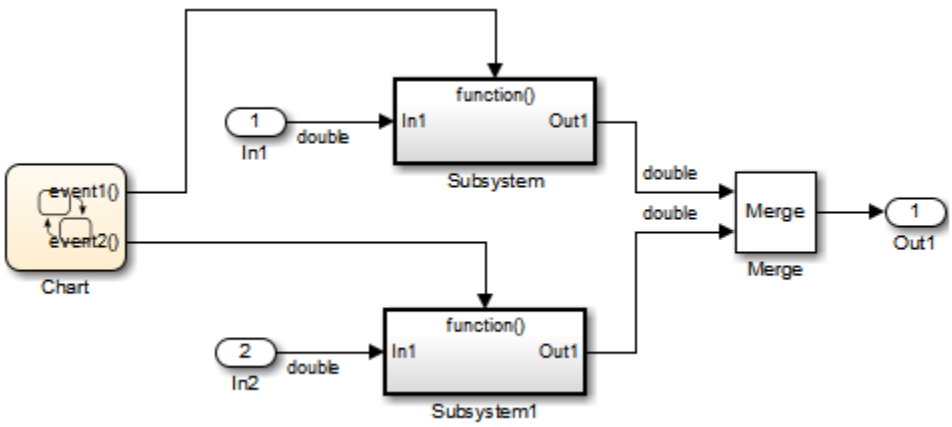
For more information on using the Model Advisor, see “Run Model Advisor Checks”. For more information about the checks, see “Model Advisor Checks”.

Design Techniques to Optimize a Model for Code Generation

You can apply the following techniques to optimize a model for code generation:

- For Embedded Coder users, if your application uses only integer arithmetic, clear the **Support floating-point numbers** parameter in the **Software environment** section of the **Interface** pane so that the generated code does not contain floating-point data or operations. When this parameter is cleared, an error is raised if noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.
- Disable the **Configuration Parameters** > **Code Generation** > **Interface** > **Advanced parameters** > **MAT-file logging** parameter. Deselecting this parameter eliminates extra code and memory usage for initializing, updating, and cleaning-up logging variables. In addition, the code generated to support MAT-file logging invokes `malloc`, which can be undesirable for your application.
- Use the Upgrade Advisor to upgrade older models (saved by prior versions or the current version) to use current features. For details, see “Model Upgrades”.
- Before building, set optimization flags for the compiler (for example, `-O2` for `gcc`, `-Ot` for the Microsoft Visual C++ compiler).

- Directly inline C/C++ S-functions into the generated code by writing a TLC file for the S-function. For more information, see “Use S-Function Target for Model or Subsystem” and see “Inline C MEX S-Functions”.
- Use a Simulink data type other than `double` when possible. The available data types are `Boolean`, signed and unsigned 8-, 16-, and 32-bit integers, and 32- and 64-bit floats (a `double` is a 64-bit float). For more information, see “About Data Types in Simulink”. For a block-by-block summary, click `showblockdatatypetable` or type the command in the Command Window.
- For tunable block parameters that you configure to store in memory in the generated code, you can match parameter data types with signal data types to eliminate unnecessary typecasts and C shifts. Where possible, store parameter values in small integer data types. See “Parameter Data Types in the Generated Code” on page 36-131.
- Remove repeated values in lookup table data.
- Use the Merge block to merge the output of signals wherever possible. This block is particularly helpful when you need to control the execution of function-call subsystems with a Stateflow chart. The following model shows an example of how to use the Merge block.



When more than one signal connected to a Merge block has a non-`Auto` storage class, all non-`Auto` signals connected to that block must *be identically labeled* and *have the same storage class*. When Merge blocks connect directly to one another, these rules apply to the signals connected to any of the Merge blocks in the group.

Design Techniques to Optimize Stateflow Objects for Code Generation

Be Explicit About the Inline Option of a Graphical Function

When you use a graphical function in a Stateflow chart, select `Inline` or `Function` for the property **Function Inline Option**. Otherwise, the code generated for a graphical function may not appear as you want. For more information, see “Specify Properties of Graphical Functions” (Stateflow).

Avoid Using Multiple Edge-Triggered Events in Stateflow Charts

If you use more than one trigger, you generate multiple code statements to handle rising or falling edge detections. If multiple triggers are required, use function-call events instead. For more information, see “Activate a Stateflow Chart by Sending Input Events” (Stateflow).

Combine Input Signals of a Chart Into a Single Bus Object

When you use a bus object, you reduce the number of parameters in the parameter list of a generated function. This guideline also applies to output signals of a chart. For more information, see “Define Stateflow Structures” (Stateflow).

Use Charts with Discrete Sample Times

The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of Simulink provided time. This allows for more efficient code generation in terms of overhead and memory, as well as enabling this code for use in software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

See Also

Related Examples

- “Increase Code Generation Speed” on page 76-3
- “Execution Profiling for Generated Code” on page 76-11
- “Model Configuration Parameters: Code Generation Optimization”
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

Control Memory Allocation for Time Counters

The **Application lifespan (days)** parameter lets you control the allocation of memory for absolute and elapsed time counters. Such counters exist in the code for blocks that use absolute or elapsed time. For a list of such blocks, see “Absolute Time Limitations”.

The size of the time counters in generated code is 8, 16, 32, or 64 bits. The size is set automatically to the minimum that can accommodate the duration value specified by **Application lifespan (days)** given the step size specified in the Configuration Parameters **Solver** pane. To minimize the amount of RAM used by time counters, specify the smallest lifespan possible and the largest step size possible.

An application runs to its specified lifespan. It may be able to run longer. For example, running a model with a step size of one millisecond (0.001 seconds) for one day requires a 32-bit timer, which could continue running without overflow for 49 days more.

To maximize application lifespan, specify **Application lifespan (days)** as `inf`. This value allocates 64 bits (two `uint32` words) for each timer. Using 64 bits to store timing data would allow a model with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. 64-bit counters do not violate the usual code generator length limitation of 32 bits because the value of a time counter does not provide the value of a signal, state, or parameter.

See Also

“Application lifespan (days)”

Related Examples

- “Absolute and Elapsed Time Computation”
- “Timers in Asynchronous Tasks”

Execution Profiling for Generated Code

Use code execution profiling to:

- Determine whether the generated code meets execution time requirements for real-time deployment on your target hardware.
- Identify code sections that require execution speed improvements.

The following tasks represent a general workflow that uses code execution profiling:

1. With the Simulink model, design and optimize your algorithm.
2. Configure the model for code execution profiling, and generate code.
3. Execute generated code on target hardware. For example, you can:
   - Run a software-in-the-loop (SIL) simulation on your development computer.
   - Run a processor-in-the-loop (PIL) simulation using a target support package or custom PIL target.
   - Run an XCP-based external mode simulation on your development computer or on custom hardware.
   - Perform real-time execution with Simulink Real-Time or a target support package.
4. Analyze execution speed through code execution profiling plots and reports. For example, check that the algorithm code satisfies execution time requirements for real-time deployment:
   - If the algorithm code easily meets the requirements, consider enhancing your algorithm to exploit available processing power.
   - If the code cannot be executed in real time, look for ways to reduce execution time.

     Identify the tasks that require the most time. For these tasks, investigate whether trade-offs between functionality and speed are possible.

     If your target is a multicore processor, distribute the execution of algorithm code across available cores.
5. If required, refine the model and return to step 2.

To find information about code execution profiling with Simulink products, use the following table.

Type of Execution	Target	Relevant Products	See
Software-in-the-loop (SIL)	Development computer	Embedded Coder	• “Create Execution-Time Profile for Generated Code” on page 81-2 • “View and Compare Code Execution Times” on page 81-6 • “Analyze Code Execution Data” on page 81-29

Type of Execution	Target	Relevant Products	See
Processor-in-the-loop (PIL)	Embedded hardware or instruction set simulator	Embedded Coder	• "Create Execution-Time Profile for Generated Code" on page 81-2 • "View and Compare Code Execution Times" on page 81-6 • "Analyze Code Execution Data" on page 81-29
XCP external mode	Development computer or custom hardware	Embedded Coder	• "Create Execution-Time Profile for Generated Code" on page 81-2 • "Analyze Code Execution Data" on page 81-29
Real-time execution	Simulink Real-Time	Simulink Coder, Simulink Real-Time	• Execution Profiling for Real-Time Applications (Simulink Real-Time)

See Also

Related Examples

- "Optimize and Deploy on a Multicore Target"
- "Design Techniques to Optimize Models for Efficient Code Generation" on page 76-7
- "SIL and PIL Simulations" on page 88-2
- "External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling" on page 60-2

Optimize Generated Code by Combining Multiple for Constructs

This example shows how the code generator combines `for` loops. The generated code uses `for` constructs to represent a variety of modeling patterns, such as a matrix signal or Iterator blocks. Using data dependency analysis, the code generator combines `for` constructs to reduce static code size and runtime branching.

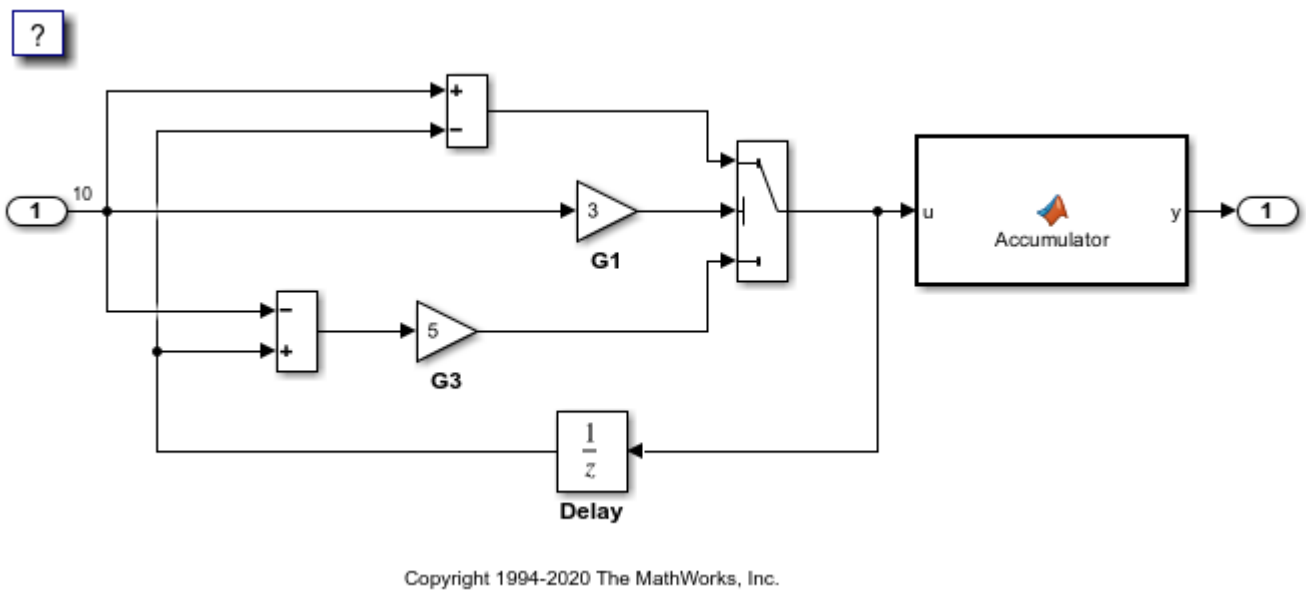
The benefits of optimizing `for` loops are:

- Reducing ROM and RAM consumption.
- Increasing execution speed.

`for` Loop Modeling Patterns

Open the example model `ForLoopConstruct`. The Switch block and MATLAB Function block represent `for` constructs. In the `In1` Block Parameters dialog box, the **Port dimensions** parameter is set to `10` .

```
model = 'ForLoopConstruct';
open_system(model);
```



Generate Code

The model does not contain data dependencies across the `for` loop iterations. Therefore, the code generator combines `for` loops into one loop.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: ForLoopConstruct
### Successful completion of build procedure for: ForLoopConstruct
```

```
Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
========================================================================================
ForLoopConstruct  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.915s
```

View the optimized code in `ForLoopConstruct.c`.

```
cfile = fullfile('ForLoopConstruct_grt_rtw','ForLoopConstruct.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void ForLoopConstruct_step(void)
{
  real_T In1;
  int32_T k;

  /* MATLAB Function: '<Root>/Accum' */
  /* MATLAB Function 'Accum': '<S1>:1' */
  /* '<S1>:1:3' */
  /* '<S1>:1:4' */
  ForLoopConstruct_Y.Out1 = 0.0;

  /* '<S1>:1:5' */
  for (k = 0; k < 10; k++) {
    /* Gain: '<Root>/G1' incorporates:
     *  Inport: '<Root>/In1'
     */
    In1 = ForLoopConstruct_U.In1[k];

    /* Switch: '<Root>/Switch' incorporates:
     *  Gain: '<Root>/G1'
     *  Gain: '<Root>/G3'
     *  Sum: '<Root>/Sum1'
     *  Sum: '<Root>/Sum2'
     *  UnitDelay: '<Root>/Delay'
     */
    if (3.0 * In1 >= 0.0) {
      In1 -= ForLoopConstruct_DW.Delay_DSTATE[k];
    } else {
      In1 = (ForLoopConstruct_DW.Delay_DSTATE[k] - In1) * 5.0;
    }

    ForLoopConstruct_DW.Delay_DSTATE[k] = In1;

    /* End of Switch: '<Root>/Switch' */

    /* MATLAB Function: '<Root>/Accum' */
    /* '<S1>:1:6' */
    ForLoopConstruct_Y.Out1 += ((real_T)k + 1.0) + In1;
```

```
  }
}
```

The generated code contains code for the single `for` loop.

Close the model

```
bdclose(model)
rtwdemoclean;
```

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Configure Loop Unrolling Threshold” on page 76-46
- “For Loop” on page 26-31

Subnormal Number Execution Speed

Subnormal numbers, formerly known as denormal numbers in floating-point literature, fill the underflow gap around zero in floating-point arithmetic. Subnormal values are a special category of floating-point values that are too close to `0.0` to be represented as a normalized value. The leading significand (mantissa) of a subnormal number is zero. When adding and subtracting floating-point numbers, subnormal numbers prevent underflow.

Using subnormal numbers provides precision beyond the normal representation by using leading zeros in the significand to represent smaller values after the representation reaches the minimum exponent. As the value approaches `0.0`, you trade off precision for extended range. Subnormal numbers are useful if your application requires extra range.

However, in a real-time system, using subnormal numbers can dramatically increase execution latency, resulting in excessive design margins and real-time overruns. If the simulation or generated code performs calculations that produce or consume subnormal numbers, the execution of these calculations can be up to 50 times slower than similar calculations on normal numbers. The actual simulation or code execution time for subnormal number calculations depends on your computer operating environment. Typically, for desktop processors, the execution time for subnormal number calculations is five times slower than similar calculations on normal numbers.

To minimize the possibility of execution slowdowns or overruns due to subnormal number calculation latency, do one of the following:

- In your model, manually flush to zero any incoming or computed subnormal values at inputs and key operations, such as washouts and filters. For an example, see “Flush Subnormal Numbers to Zero” on page 76-18.

  To detect a subnormal value for a single precision, 32-bit floating-point number:

  1. Find the smallest normalized number on a MATLAB host. In the Command Window, type:

     ```
     >> SmallestNormalSingle = realmin('single')
     ```

     In the C language, `FLT_MIN` , defined in `float.h`, is equivalent to `realmin('single')`.
  2. Look for values in range:

     ```
      0 < fabsf(x) < SmallestNormalSingle
     ```

  To detect a subnormal value for a double precision, 64-bit floating-point number:

  1. Find the smallest normalized number on a MATLAB host. In the Command Window, type:

     ```
     >> SmallestNormalDouble = realmin('double')
     ```

     In the C language, `DBL_MIN` , defined in `float.h`, is equivalent to `realmin('double')`.
  2. To detect a subnormal value, look for values in this range:

     ```
     0 < fabs(x) < SmallestNormalDouble
     ```
- Set the **Simulation behavior for denormal numbers** parameter to `Flush to zero (FTZ)` to emulate flush-to-zero behavior for all denormal results from arithmetic operations. For more information, see “Simulation behavior for denormal numbers”.

- On your processor, set flush-to-zero mode or, with your compiler, specify an option to disable subnormal numbers. Flush-to-zero modes treat a subnormal number as 0 when it is an input to a floating-point operation. Underflow exceptions do not occur in flush-to-zero mode.

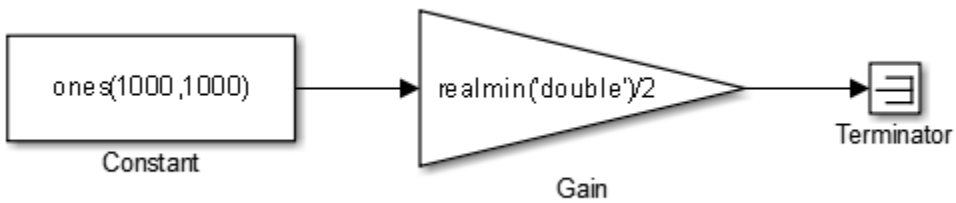
  For example, in Intel processors, the flush-to-zero (FTZ) and denormals-are-zero (DAZ) flags in the MXCSR register control floating-point calculations. For the gcc compiler on Linux, `-ffast-math` sets abrupt underflow (FTZ), flush-to-zero, while `–O3 -ffast-math` reverts to gradual underflow, using subnormal numbers.

For more information, see the IEEE Standard 754, *IEEE Standard for Floating-Point Arithmetic*.

Simulation Time with and Without Subnormal Numbers

This example shows how using subnormal numbers increases simulation time by ~5 times.

For this example, create a simple model `ex_subnormal` that has a Constant and a Gain block. The Gain is set to subnormal value `realmin('double')/2`.



To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe the elapsed times for simulation using subnormals, similar to the following:

```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 9.909326 seconds.
Elapsed time is 9.617966 seconds.
Elapsed time is 9.797183 seconds.
Elapsed time is 9.702397 seconds.
Elapsed time is 9.893946 seconds.
```

Set the Gain to a number, `2`, that is not a subnormal value:

```
>> set_param('ex_subnormal/Gain', 'Gain', '2');
```

To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe elapsed times for simulations that do not use subnormal values, similar to the following:

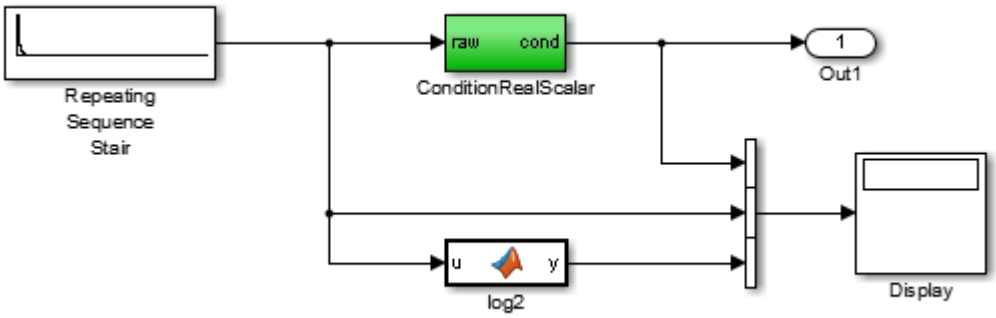
```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 2.045123 seconds.
Elapsed time is 1.796598 seconds.
Elapsed time is 1.758458 seconds.
```

```
Elapsed time is 1.721721 seconds.
Elapsed time is 1.780569 seconds.
```

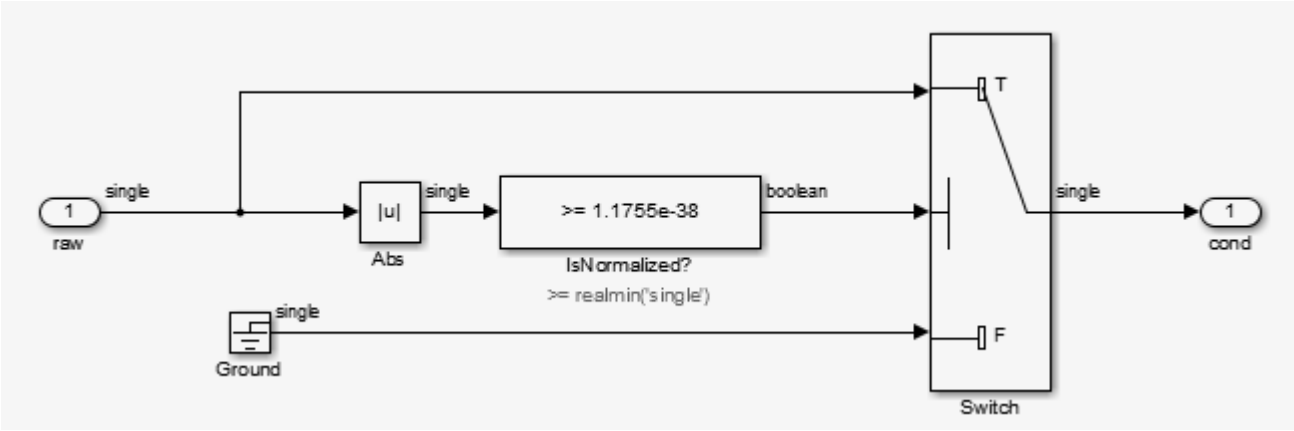
Flush Subnormal Numbers to Zero

This example shows how to flush single precision subnormal numbers to zero.

1 For this example, create a simple model `ex_flush_to_zero`.



- `Repeating Sequence Stair` generates a sequence of numbers from two raised to the power of 0 through two raised to the power of -165. The sequence approaches zero.
- `ConditionRealScalar` flushes subnormal single precision values that are less than `realmin('single')` to zero.



- MATLAB function block `log2` generates the base 2 logarithm of the `Repeating Sequence Stair` output. Specifically, `log2` generates the numbers 0 through -165.

```
function y = fcn(u)
%#codegen

y = log2(u);
end
```

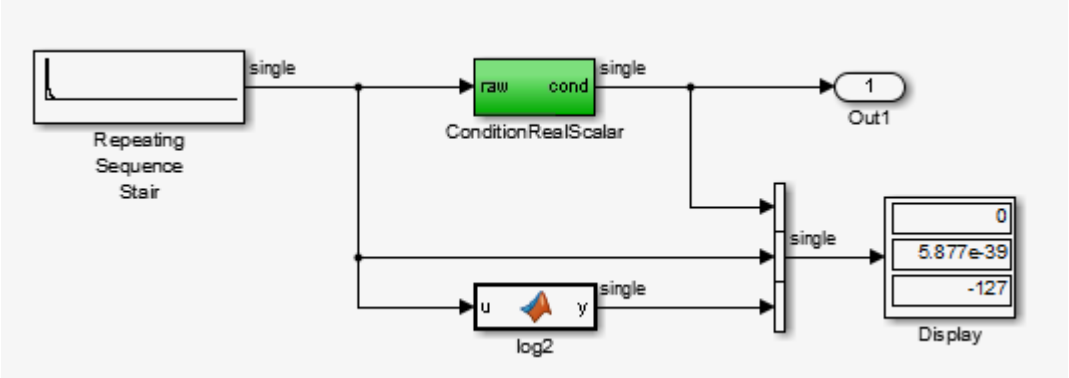
2 On the **Simulation** > **Step Back** > **Configure simulation stepping** pane:

- Select **Enable stepping back**.
- Select **Pause simulation when time reaches** and enter `121`.

3 In the model window, run the simulation. The simulation pauses at `T=121`. The displayed values:

- `ConditionRealScalar` output approaches zero.
- `Repeating Sequence Stair output` approaches zero.

4 Step the simulation forward to `T=127`. `ConditionRealScalar` flushes the subnormal value output from `Repeating Sequence Stair` to zero.



5 Continue stepping the simulation forward. `ConditionRealScalar` flushes the subnormal single precision values output from `Repeating Sequence Stair` to zero. When `T=150`, the output of `Repeating Squence Stair` is itself zero.

See Also

Related Examples

- “Data Types Supported by Simulink”
- “Numerical Consistency of Model and Generated Code Simulation Results”
- “Specify Single-Precision Data Type for Embedded Application” on page 36-104

Remove Code for Out-of-Range Floating Point to Integer Conversions

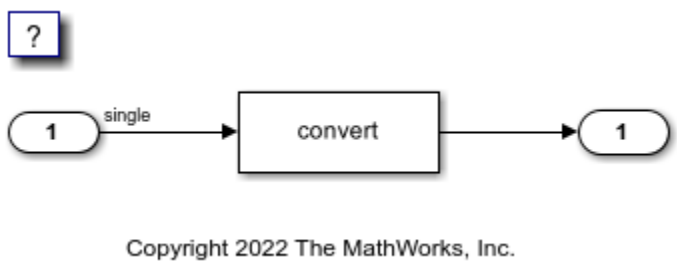
This example shows how to generate optimized code by removing code for out-of-range floating point to integer conversions. Standard C does not define the behavior of out-of-range floating point to integer conversions, while these conversions are well-defined during simulation. When you select the configuration parameter **Remove code from floating-point to integer conversions that wraps out-of-range values**, the code generator removes the wrapping code that handles out-of-range floating point to integer conversion and improves code efficiency. Without this code, there might be a mismatch between simulation and code generation results for out-of range values.

If the input values in your application are in the range of the output type, remove code for out-of-range floating-point to integer conversions. Removing this code reduces the size and increases the speed of the generated code.

Example Model

Open the example model `FloatMultiplicationNetSlope`.

```
model = 'FloatMultiplicationNetSlope';
open_system(model);
```



The model contains a Data Type Conversion block. To configure the model to convert the input signal from a `single` to a `uint8`, open the Data Type Conversion dialog box. For the **Output data type** parameter, select `uint8`. Alternatively, use this command-line API to set the **Output data type**:

```
set_param('FloatMultiplicationNetSlope/Data Type Conversion', 'OutDataTypeStr','uint8');
```

A uint8 can support values from 0 to 255. If the input signal has a value outside of this range, an out-of-range conversion occurs.

Generate Code Without Optimization

1. Open the Configuration Parameters dialog box. On the Optimization pane, clear the **Remove code from floating-point to integer conversions that wraps out-of-range values** parameter check box. Alternatively, use the command-line API to disable the optimization:

```
set_param(model, 'EfficientFloat2IntCast','off');
```

2. Build the model by using the `slbuild` function or by pressing **Ctrl+B**.

```
slbuild(model)
```

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.377s
```

3. Inspect the generated `FloatMultiplicationNetSlope_step` step function in the `FloatMultiplicationNetSlope.c`.

```
file = fullfile('FloatMultiplicationNetSlope_grt_rtw','FloatMultiplicationNetSlope.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  real32_T tmp;

  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/Input'
   */
  tmp = (real32_T)floor(FloatMultiplicationNetSlope_U.Input);
  if (rtIsNaNF(tmp) || rtIsInfF(tmp)) {
    tmp = 0.0F;
  } else {
    tmp = (real32_T)fmod(tmp, 256.0);
  }

  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   */
  FloatMultiplicationNetSlope_Y.Output = (uint8_T)(tmp < 0.0F ? (int32_T)
    (uint8_T)-(int8_T)(uint8_T)-tmp : (int32_T)(uint8_T)tmp);
}
```

The code generator applies the `fmod` function to handle out-of range-results. In this case, the generated code is larger but the results of the generated code match the simulation results.

Generate Code with Optimization

1. Open the Configuration Parameters dialog box. On the Optimization pane, select the **Remove code from floating-point to integer conversions that wraps out-of-range values** parameter. Alternatively, use the command-line API to enable the optimization:

```
set_param(model, 'EfficientFloat2IntCast','on')
```

2. Build the model. Inspect the generated `FloatMultiplicationNetSlope_step` step function with optimization in the `FloatMultiplicationNetSlope.c`

```
slbuild(model);
file = fullfile('FloatMultiplicationNetSlope_grt_rtw','FloatMultiplicationNetSlope.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:

Model                         Action                        Rebuild Reason
========================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.0399s

/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/Input'
   */
  FloatMultiplicationNetSlope_Y.Output = (uint8_T)
    FloatMultiplicationNetSlope_U.Input;
}
```

The generated code is more efficient without the protective code, but it is possible that the execution of the generated code does not produce the same results as simulation for values not in the range of 0 to 255.

Clean Up Example Folders and Files

Close the model and remove temporary folders and files.

```
bdclose(model);
rtwdemoclean;
```

See Also

Remove code from floating-point to integer conversions that wraps out-of-range values

Related Examples

- “hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values”
- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Remove Code That Maps NaN to Integer Zero” on page 76-23

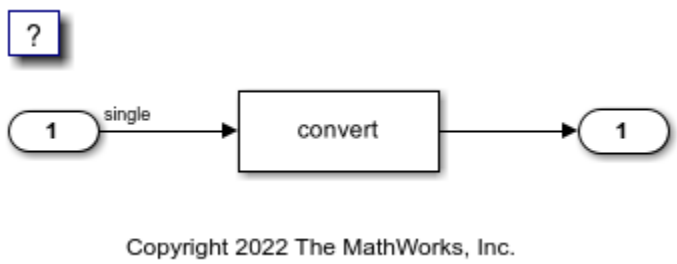
Remove Code That Maps NaN to Integer Zero

This example shows how to remove code that maps `NaN` to integer zero. For floating point to integer conversions involving saturation, Simulink® converts NaN to integer zero during simulation. If your model contains an input value of `NaN`, you can specify that the code generator produce code that maps NaN to zero. Without this code, there is a mismatch between simulation and code generation results because in Standard C, every condition involving `NaN` evaluates to false.

If input values of `NaN` do not exist in your application, you can remove code that maps NaN to integer zero. Removing this code reduces the size and increases the speed of the generated code.

Example Model

Open the example model `FloatMultiplicationNetSlope` by entering the model name in the Command Window.



The model contains a Data Type Conversion block. Configure the model to convert the input signal from a `single` to a `uint8`.

1. Open the Data Type Conversion dialog box. For the **Output data type** parameter, select `uint8`.

2. Select the **Saturate on integer overflow** parameter check box. Selecting this parameter specifies that an out-of-range signal value equals either the minimum or maximum value that the data type can represent.

3. Open the Configuration Parameters dialog box. Select the **Support: non-finite numbers** parameter check box. Selecting this parameter enables generating nonfinite data and operations on nonfinite data.

Generate Code Without Optimization

1. Open the Configuration Parameters dialog box. On the Optimization pane, clear the **Remove code from floating-point to integer conversions with saturation that maps NaN to zero** parameter check box.

2. Build the model.

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:
```

```
Model                        Action                        Rebuild Reason
==============================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.645s
```

3. Inspect the generated `FloatMultiplicationNetSlope_step` step function in the `FloatMultiplicationNetSlope.c`.

```
/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/Input'
   */
  if (FloatMultiplicationNetSlope_U.Input < 256.0F) {
    if (FloatMultiplicationNetSlope_U.Input >= 0.0F) {
      /* Outport: '<Root>/Output' */
      FloatMultiplicationNetSlope_Y.Output = (uint8_T)
        FloatMultiplicationNetSlope_U.Input;
    } else {
      /* Outport: '<Root>/Output' */
      FloatMultiplicationNetSlope_Y.Output = 0U;
    }
  } else if (FloatMultiplicationNetSlope_U.Input >= 256.0F) {
    /* Outport: '<Root>/Output' */
    FloatMultiplicationNetSlope_Y.Output = MAX_uint8_T;
  } else {
    /* Outport: '<Root>/Output' */
    FloatMultiplicationNetSlope_Y.Output = 0U;
  }

  /* End of DataTypeConversion: '<Root>/Data Type Conversion' */
}
```

The code generator applies condition to map `NaN` to interger zero. In this case, the generated code is larger but the results of the generated code match with simulation results.

Generate Code with Optimization

1. Open the Configuration Parameters dialog box.

2. On the Optimization pane, select the **Remove code from floating-point to integer conversions with saturation that maps NaN to zero** parameter.

3. Build the model.

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
```

```
==========================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.701s
```

4. Inspect the generated `FloatMultiplicationNetSlope_step` step function with optimization in the `FloatMultiplicationNetSlope.c`.

```
/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/Input'
   */
  if (FloatMultiplicationNetSlope_U.Input < 256.0F) {
    if (FloatMultiplicationNetSlope_U.Input >= 0.0F) {
      /* Outport: '<Root>/Output' */
      FloatMultiplicationNetSlope_Y.Output = (uint8_T)
        FloatMultiplicationNetSlope_U.Input;
    } else {
      /* Outport: '<Root>/Output' */
      FloatMultiplicationNetSlope_Y.Output = 0U;
    }
  } else {
    /* Outport: '<Root>/Output' */
    FloatMultiplicationNetSlope_Y.Output = MAX_uint8_T;
  }

  /* End of DataTypeConversion: '<Root>/Data Type Conversion' */
}
```

The generated code maps `NaN` to 255 and not integer zero. The generated code is more efficient without the extra code that maps NaN to integer zero. But it is possible that the execution of generated code does not produce the same results as the simulation for `NaN` values.

See Also

Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Remove Code for Out-of-Range Floating Point to Integer Conversions” on page 76-20

Disable Nonfinite Checks or Inlining for Math Functions

When the code generator produces code for math functions:

- If the model option **Support non-finite numbers** is selected, nonfinite number checking is generated uniformly for math functions, without the ability to specify that nonfinite number checking should be generated for some functions, but not for others.
- By default, inlining is applied uniformly for math functions, without the ability to specify that inlining should be generated for some functions, while invocations should be generated for others.

You can use code replacement library (CRL) customization entries to:

- Selectively disable nonfinite checks for math functions. This can improve the execution speed of the generated code.
- Selectively disable inlining of math functions. This can increase code readability and decrease code size.

The functions for which these customizations are supported include the following:

- Floating-point only: `atan2`, `copysign`, `fix`, `hypot`, `log`, `log10`, `round`, `sincos`, and `sqrt`
- Floating-point and integer: `abs`, `max`, `min`, `mod`, `rem`, `saturate`, and `sign`

The general workflow for disabling nonfinite number checking and/or inlining is as follows:

1. If you can disable nonfinite number checking for a particular math function, or if you want to disable inlining for a particular math function and instead generate a function invocation, you can copy the following MATLAB function code into a MATLAB file with an `.m` file name extension, for example, `crl_table_customization.m`.

```
function hTable = crl_table_customization

% Create an instance of the Code Replacement Library table for controlling
% function intrinsic inlining and nonfinite support

hTable = RTW.TflTable;

% Inline - true (if function needs to be inline)
%          false (if function should not be inlined)
% SNF (support nonfinite) - ENABLE (if non-finite checking should be performed)
%                           DISABLE (if non-finite checking should NOT be performed)
%                           UNSPECIFIED (Default behavior)

% registerCustomizationEntry(hTable, ...
%       Priority, numInputs, key, inType, outType, Inline, SNF);

registerCustomizationEntry(hTable, ...
        100, 2, 'atan2', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'atan2', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'sincos', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sincos', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int32',  'int32',  true, 'UNSPECIFIED');
```

```
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'integer','integer',true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint8',  'uint8',  true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'hypot', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'hypot', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'log', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'log', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'log10', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'log10', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int32',  'int32',  false, 'UNSPECIFIED');
```

```
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int16',  'int16',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int8',   'int8',   false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint8',  'uint8',  false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int32',  'int32',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int16',  'int16',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int8',   'int8',   false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint8',  'uint8',  false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'round', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'round', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'double', 'double',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'single', 'single',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int32',  'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int16',  'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int8',   'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint32', 'uint32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint16', 'uint16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint8',  'uint8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'integer','integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
```

```
        100, 1, 'sqrt', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sqrt', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'fix', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'fix', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'copysign', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'copysign', 'single', 'single', true, 'UNSPECIFIED');
```

2. To reduce the size of the file, you can delete the `registerCustomizationEntry` lines for functions for which the default nonfinite number checking and inlining behavior is acceptable.
3. For each remaining entry,
   - Set the `Inline` argument to `true` if the function should be inlined or `false` if it should not be inlined.
   - Set the `SNF` argument to `ENABLE` if nonfinite checking should be generated, `DISABLE` if nonfinite checking should not be generated, or `UNSPECIFIED` to accept the default behavior based on the model option settings.

   Save the file.
4. Optionally, perform a quick check of the syntactic validity of the customization table entries by invoking the table definition file at the MATLAB command line (`>> tbl = crl_table_customization`). Fix syntax errors that are flagged.
5. Optionally, view the customization table entries in the **Code Replacement Viewer** (`>> crviewer(crl_table_customization)`). For more information about viewing code replacement tables, see “Choose a Code Replacement Library”.
6. To register these changes and make them appear in the **Code replacement library** drop-down list located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, first copy the following MATLAB function code into an instance of the file `rtwTargetInfo.m`.

   Note For the example below, specify the argument `'RTW'` if a GRT target is selected for your model, otherwise omit the argument.

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

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

end % End of RTWTARGETINFO


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

  % Instantiate a CRL registry entry - specify 'RTW' for GRT
  thisCrl = RTW.TflRegistry('RTW');

  % Define the CRL properties
  thisCrl.Name = 'CRL Customization Example';
  thisCrl.Description = 'Example  of CRL Customization';
  thisCrl.TableList = {'crl_table_customization'};
  thisCrl.TargetHWDeviceType = {'*'};
```

```
end % End of LOCCRLREGFCN
```

You can edit the `Name` field to specify the library name that appears in the **Code replacement library** drop-down list. Also, the file name in the `TableList` field must match the name of the file you created in step 1.

To register your changes, with both of the MATLAB files you created present in the MATLAB path, enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

7. Create or open a model that generates function code corresponding to one of the math functions for which you specified a change in nonfinite number checking or inlining behavior.
8. Open the Configuration Parameters dialog box, go to the **Code Generation** > **Interface** pane, and use the **Code replacement library** drop-down list to select the code replacement entry you registered in step 6, for example, `CRL Customization Example`.
9. Generate code for the model and examine the generated code to verify that the math functions are generated as expected.

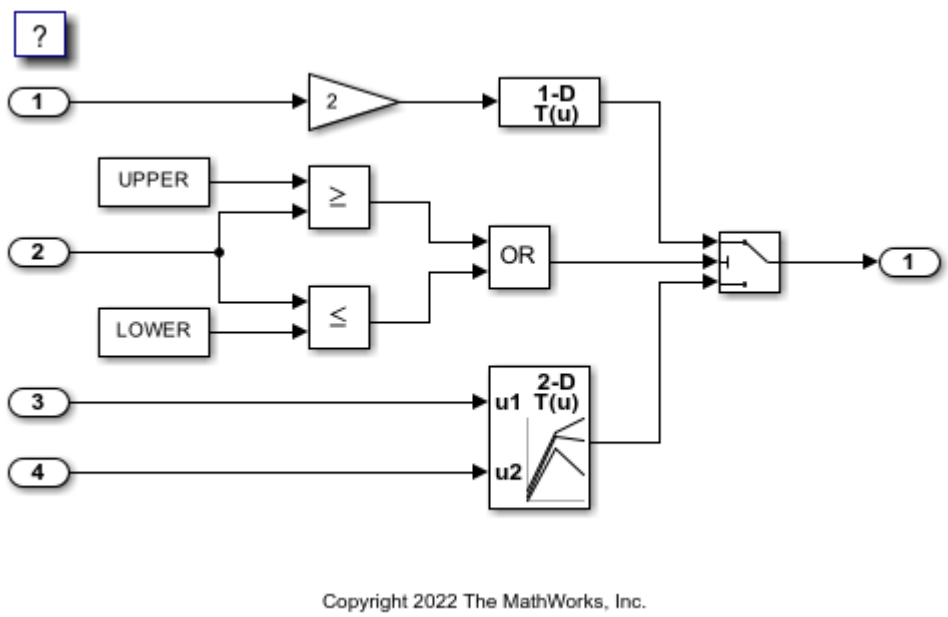
Fold Expressions

Expression folding optimizes code to minimize the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. When expression folding is on, the code generator collapses (folds) block computations into a single expression, instead of generating separate code statements and storage declarations for each block in a model. Most Simulink® blocks support expression folding.

Expression folding improves the efficiency of generated code, frequently achieving results that compare favorably to hand-optimized code. In many cases, entire groups of model computations fold into a single, highly optimized line of code.

Example Model

```
model = 'FoldBlockComputations';
open_system(model);
```



Generate Code

Expression folding is available only when the **Signal storage reuse** parameter is set to on because expression folding operates only on expressions involving local variables. The **Signal storage reuse** and **Eliminate superfluous local variables (expression folding)** parameters are on by default. Clear the **Eliminate superfluous local variables (expression folding)** parameter or enter the following command in the MATLAB® Command Window to turn the parameter off:

```
set_param(model, 'ExpressionFolding','off');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: FoldBlockComputations
### Successful completion of build procedure for: FoldBlockComputations
```

```
Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
FoldBlockComputations  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.593s
```

With expression folding off, in the `FoldBlockComputations.c` file, there are separate code statements before and in the Switch block operation.

```
cfile = fullfile('FoldBlockComputations_grt_rtw','FoldBlockComputations.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void FoldBlockComputations_step(void)
{
  real_T rtb_Gain;
  boolean_T rtb_LogicalOperator;
  boolean_T rtb_RelationalOperator;

  /* RelationalOperator: '<Root>/Relational Operator1' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In2'
   */
  rtb_LogicalOperator = (FoldBlockComputations_P.UPPER >=
    FoldBlockComputations_U.In2);

  /* RelationalOperator: '<Root>/Relational Operator' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In2'
   */
  rtb_RelationalOperator = (FoldBlockComputations_U.In2 <=
    FoldBlockComputations_P.LOWER);

  /* Logic: '<Root>/Logical Operator' */
  rtb_LogicalOperator = (rtb_LogicalOperator || rtb_RelationalOperator);

  /* Switch: '<Root>/Switch' */
  if (rtb_LogicalOperator) {
    /* Gain: '<Root>/Gain' incorporates:
     *  Inport: '<Root>/In1'
     */
    rtb_Gain = 2.0 * FoldBlockComputations_U.In1;

    /* Lookup_n-D: '<Root>/Look-Up Table' incorporates:
     *  Gain: '<Root>/Gain'
     */
    FoldBlockComputations_Y.Out1 = look1_binlx(rtb_Gain,
      FoldBlockComputations_P.T1Break, FoldBlockComputations_P.T1Data, 10U);
  } else {
    /* Lookup_n-D: '<Root>/Look-Up Table' incorporates:
     *  Inport: '<Root>/In3'
```

```
     *  Inport: '<Root>/In4'
     *  Lookup_n-D: '<Root>/Look-Up Table (2-D)'
     */
    FoldBlockComputations_Y.Out1 = look2_binlx(FoldBlockComputations_U.In3,
      FoldBlockComputations_U.In4, FoldBlockComputations_P.T2Break,
      FoldBlockComputations_P.T2Break, FoldBlockComputations_P.T2Data,
      rtCP_LookUpTable2D_maxIndex, 3U);
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Generate Code with Optimization

Enter the following command to turn expression folding on:

```
set_param(model, 'ExpressionFolding','on');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: FoldBlockComputations
### Successful completion of build procedure for: FoldBlockComputations

Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
FoldBlockComputations  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.251s
```

The following is a portion of `FoldBlockComputations.c`. In the optimized code, the code generator folds all computations into the Switch block operation.

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void FoldBlockComputations_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In2'
   *  Logic: '<Root>/Logical Operator'
   *  RelationalOperator: '<Root>/Relational Operator'
   *  RelationalOperator: '<Root>/Relational Operator1'
   */
  if ((FoldBlockComputations_P.UPPER >= FoldBlockComputations_U.In2) ||
      (FoldBlockComputations_U.In2 <= FoldBlockComputations_P.LOWER)) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Gain: '<Root>/Gain'
     *  Inport: '<Root>/In1'
     *  Lookup_n-D: '<Root>/Look-Up Table'
```

```
     */
    FoldBlockComputations_Y.Out1 = look1_binlx(2.0 * FoldBlockComputations_U.In1,
      FoldBlockComputations_P.T1Break, FoldBlockComputations_P.T1Data, 10U);
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In3'
     *  Inport: '<Root>/In4'
     *  Lookup_n-D: '<Root>/Look-Up Table (2-D)'
     */
    FoldBlockComputations_Y.Out1 = look2_binlx(FoldBlockComputations_U.In3,
      FoldBlockComputations_U.In4, FoldBlockComputations_P.T2Break,
      FoldBlockComputations_P.T2Break, FoldBlockComputations_P.T2Data,
      rtCP_LookUpTable2D_maxIndex, 3U);
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
```

Minimize Computations and Storage for Intermediate Results at Block Outputs

In this section...

"Expression Folding" on page 76-35
"Example Model" on page 76-35
"Generate Code" on page 76-35
"Enable Optimization" on page 76-36
"Generate Code with Optimization" on page 76-36

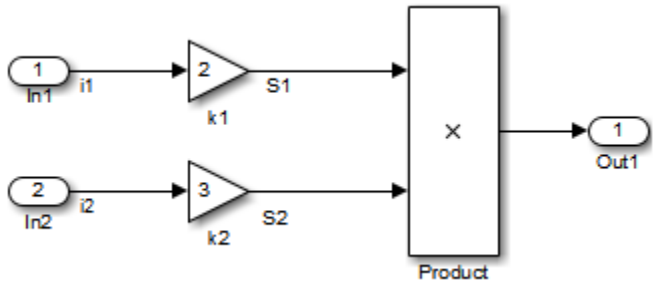
Expression Folding

Expression folding optimizes code to minimize the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. When expression folding is on, the code generator collapses (folds) block computations into a single expression, instead of generating separate code statements and storage declarations for each block in the model. Most Simulink blocks support expression folding.

Expression folding improves the efficiency of generated code, frequently achieving results that compare favorably to hand-optimized code. In many cases, entire groups of model computations fold into a single, highly optimized line of code.

You can use expression folding in your own inlined S-function blocks. For more information, see "S-Functions That Support Expression Folding".

Example Model



Generate Code

With expression folding off, in the `explfld.c` file, the code generator generates this code.

```
/* Model step function */
void exprfld_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
```

```
  exprfld_B.S1 = exprfld_P.Gain_Gain * exprfld_U.i1;

  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In2'
   */
  exprfld_B.S2 = exprfld_P.Gain1_Gain * exprfld_U.i2;

  /* Outport: '<Root>/Out1' incorporates:
   *  Product: '<Root>/Product'
   */
  exprfld_Y.Out1 = exprfld_B.S1 * exprfld_B.S2;
}
```

There are separate code statements for both Gain blocks. Before final output, these code statements compute temporary results for the Gain blocks.

Enable Optimization

Expression folding is on by default. To see if expression folding is on for an existing model:

1. Expression folding is available only when the **Configuration Parameters > Signal storage reuse** parameter is selected because expression folding operates only on expressions involving local variables. Enable the **Signal storage reuse** parameter.
2. When you select **Signal storage reuse**, the **Enable local block outputs**, **Reuse local block outputs**, and **Eliminate superfluous local variables (expression folding)** parameters are all on by default.

Generate Code with Optimization

With expression folding, the code generator generates a single-line output computation, as shown in the `expfld.c` file. The generated comments document the block parameters that appear in the expression.

```
/* Model step function */
void exprfld_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Gain: '<Root>/Gain1'
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   */
  exprfld_Y.Out1 =
     exprfld_P.Gain_Gain *
     exprfld_U.i1 *
     (exprfld_P.Gain1_Gain * exprfld_U.i2);
}
```

See Also

Signal storage reuse | Reuse local block outputs | Enable local block outputs | Eliminate superfluous local variables (Expression folding)

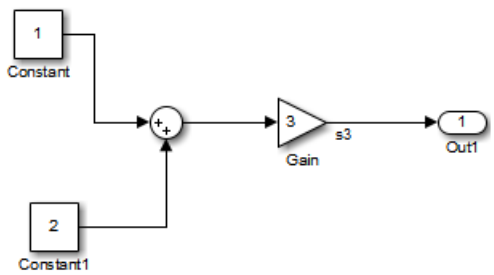
Related Examples

- “Fold Expressions”
- “Enable and Reuse Local Block Outputs in Generated Code”
- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “C Code Generation Configuration for Model Interface Elements” on page 33-2

Inline Invariant Signals

You can optimize the generated code by selecting **Inline invariant signals** on the **Optimization** pane. The generated code uses the numerical values of the invariant signals instead of their symbolic names.

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the signal `S3` is an invariant signal. An *invariant signal* is not the same as an *invariant constant*. The two constants (1 and 2) and the gain value of 3 are invariant constants. To inline invariant constants, set **Default parameter behavior** to `Inlined`.



Optimize Generated Code Using Inline Invariant Signals

This example shows how to use inline invariant signals to optimize the generated code. This optimization transforms symbolic names of invariant signals into constant values.

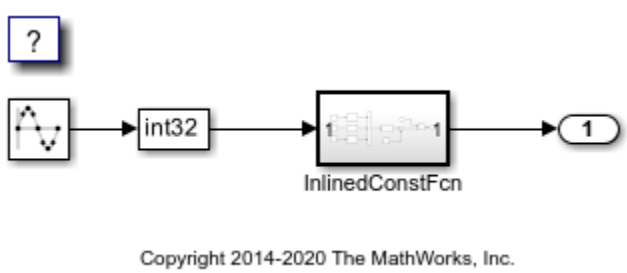
The `InlineInvariantSignals` optimization:

- Reduces ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model `InvariantSignalsInline`.

```
model = 'InvariantSignalsInline';
open_system(model);
```



Generate Code

Build the model using Simulink® Coder™.

```
slbuild(model)
```

```
### Starting build procedure for: InvariantSignalsInline
### Successful completion of build procedure for: InvariantSignalsInline

Build Summary

Top model targets built:

Model                    Action                        Rebuild Reason
==========================================================================================
InvariantSignalsInline  Code generated and compiled.  Code generation information file does not e

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.123s
```

View the generated code without the optimization. These lines of code are in `InvariantSignalsInline.c`.

```
cfile = fullfile('InvariantSignalsInline_grt_rtw',...
    'InvariantSignalsInline.c');
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
    '/* Model output', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void InvariantSignal_InlinedConstFcn(int32_T rtu_In1,
  B_InlinedConstFcn_InvariantSi_T *localB, const ConstB_InlinedConstFcn_Invari_T
  *localC)
{
  /* Product: '<S1>/Product' */
  localB->Product = rtu_In1 * localC->Sum_p;
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Inline Invariant Signals**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'InlineInvariantSignals', 'on');
```

Generate Code with Optimization

The generated code uses the numerical values of the folded constants instead of creating an additional structure.

Build the model using Simulink Coder.

```
slbuild(model)
```

```
### Starting build procedure for: InvariantSignalsInline
### Successful completion of build procedure for: InvariantSignalsInline

Build Summary

Top model targets built:
```

```
Model                   Action                        Rebuild Reason
========================================================================================
InvariantSignalsInline  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.9308s
```

View the generated code with the optimization in `InvariantSignalsInline.c`.

```
rtwdemodbtype(cfile,...
    '/* Output and update for atomic system', '/* Model output', 1, 0);


/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void InvariantSignal_InlinedConstFcn(int32_T rtu_In1,
  B_InlinedConstFcn_InvariantSi_T *localB)
{
  /* Product: '<S1>/Product' */
  localB->Product = rtu_In1 << 5;
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
```

See Also

Inline invariant signals

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Inline Numeric Values of Block Parameters” on page 76-41

Inline Numeric Values of Block Parameters

This example shows how to optimize the generated code by inlining the numeric values of block parameters. *Block parameters* include the **Gain** parameter of a Gain block and the table data and breakpoint sets of an n-D Lookup Table block.

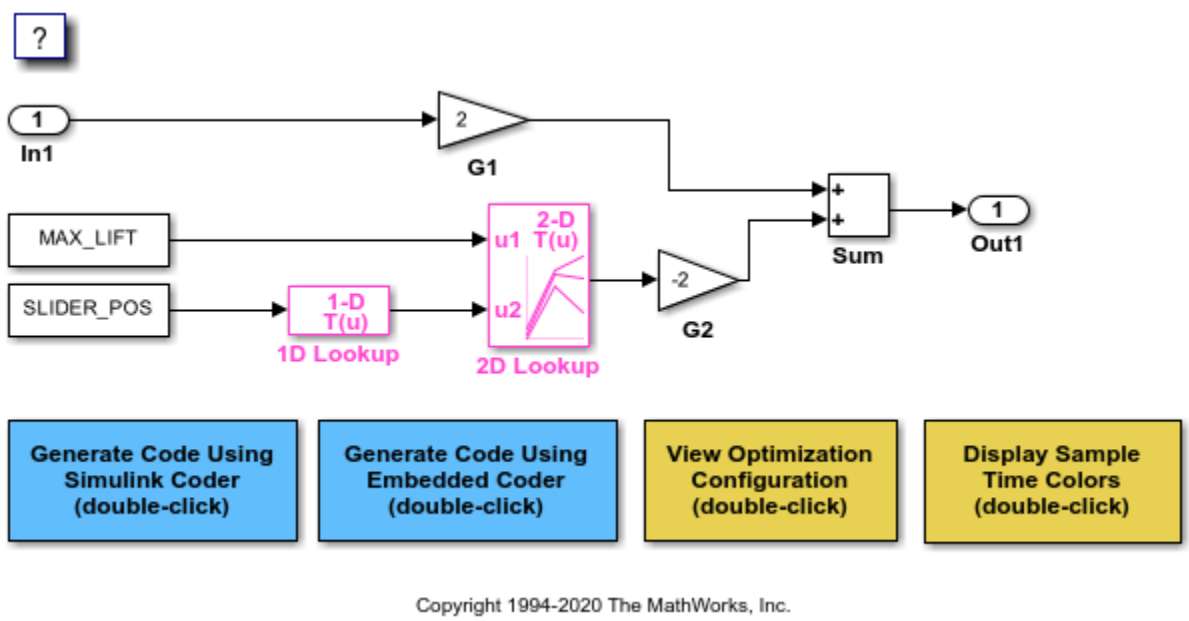
This optimization determines whether numeric block parameters occupy global memory in the generated code. The optimization can:

- Improve execution speed.
- Reduce RAM and ROM consumption.

Explore Example Model

Open the example model `rtwdemo_paraminline` and configure it to show the generated names of blocks.

```
load_system('rtwdemo_paraminline')
set_param('rtwdemo_paraminline','HideAutomaticNames','off')
open_system('rtwdemo_paraminline')
```



The model contains blocks that have these numeric parameters:

- The **Gain** parameters of the Gain blocks
- The **Constant value** parameters of the Constant blocks
- The table data and breakpoint sets of the n-D Lookup Table blocks

The output of the block G2, and the outputs of blocks upstream of G2, change only if you tune the values of the block parameters during simulation or during code execution. When you update the model diagram, these blocks and signal lines appear magenta in color.

Several blocks use `Simulink.Parameter` objects in the base workspace to set the values of their parameters. The parameter objects all use the storage class `Auto`, which means that you can configure the generated code to inline the parameter values.

Generate Code Without Optimization

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Disable the optimization by setting **Configuration Parameters > Default parameter behavior** to `Tunable`.

```
set_param('rtwdemo_paraminline','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
slbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for: rtwdemo_paraminline
### Successful completion of build procedure for: rtwdemo_paraminline

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
rtwdemo_paraminline  Code generated and compiled.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.351s
```

In the code generation report, view the source file `rtwdemo_paraminline_data.c`. The code defines a global structure that contains the block parameter values. Each block parameter in the model, such as a lookup table array, breakpoint set, or gain, appears as a field of the structure.

```
cfile = fullfile(...
    cgDir,'rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline_data.c');
rtwdemodbtype(cfile,'/* Block parameters (default storage) */', '};', 1, 1);
```

```
/* Block parameters (default storage) */
P_rtwdemo_paraminline_T rtwdemo_paraminline_P = {
  /* Variable: MAX_LIFT
   * Referenced by: '<Root>/Constant'
   */
  10.0,

  /* Variable: SLIDER_POS
   * Referenced by: '<Root>/Constant1'
   */
  0.0,

  /* Variable: T1Break
   * Referenced by: '<Root>/1D Lookup'
   */
```

```
  { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

  /* Variable: T1Data
   * Referenced by: '<Root>/1D Lookup'
   */
  { -1.0, -0.99, -0.98, -0.96, -0.76, 0.0, 0.76, 0.96, 0.98, 0.99, 1.0 },

  /* Variable: T2Break
   * Referenced by: '<Root>/2D Lookup'
   */
  { 1.0, 2.0, 3.0 },

  /* Variable: T2Data
   * Referenced by: '<Root>/2D Lookup'
   */
  { 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 },

  /* Expression: 2
   * Referenced by: '<Root>/G1'
   */
  2.0,

  /* Expression: -2
   * Referenced by: '<Root>/G2'
   */
  -2.0,

  /* Computed Parameter: uDLookup_maxIndex
   * Referenced by: '<Root>/2D Lookup'
   */
  { 2U, 2U }
};
```

You can tune the structure fields during code execution because they occupy global memory. However, at each step of the generated algorithm, the code must calculate the output of each block, including the outputs of the block G2 and the upstream blocks. View the algorithm in the model `step` function in the file `rtwdemo_paraminline.c`.

```
cfile = fullfile(cgDir,'rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(...
    cfile,'/* Model step function */','/* Model initialize function */',1,0);
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Constant: '<Root>/Constant'
   *  Constant: '<Root>/Constant1'
   *  Gain: '<Root>/G1'
   *  Gain: '<Root>/G2'
   *  Inport: '<Root>/In1'
   *  Lookup_n-D: '<Root>/1D Lookup'
   *  Lookup_n-D: '<Root>/2D Lookup'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = rtwdemo_paraminline_P.G1_Gain *
    rtwdemo_paraminline_U.In1 + rtwdemo_paraminline_P.G2_Gain * look2_binlx
```

```
    (rtwdemo_paraminline_P.MAX_LIFT, look1_binlx
     (rtwdemo_paraminline_P.SLIDER_POS, rtwdemo_paraminline_P.T1Break,
      rtwdemo_paraminline_P.T1Data, 10U), rtwdemo_paraminline_P.T2Break,
     rtwdemo_paraminline_P.T2Break, rtwdemo_paraminline_P.T2Data,
     rtwdemo_paraminline_P.uDLookup_maxIndex, 3U);
}
```

Generate Code with Optimization

Set **Default parameter behavior** to `Inlined`.

```
set_param('rtwdemo_paraminline','DefaultParameterBehavior','Inlined')
```

Generate code from the model.

```
slbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for: rtwdemo_paraminline
### Successful completion of build procedure for: rtwdemo_paraminline

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
rtwdemo_paraminline  Code generated and compiled.  Global variable MAX_LIFT changed.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.6319s
```

In the code generation report, view the algorithm in the file `rtwdemo_paraminline.c`.

```
rtwdemodbtype(...
    cfile,'/* Model step function */','/* Model initialize function */',1,0);
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/G1'
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = 2.0 * rtwdemo_paraminline_U.In1 + 150.0;
}
```

The code does not allocate memory for block parameters or for parameter objects that use the storage class `Auto`. Instead, the code generator uses the parameter values from the model, and from the parameter objects, to calculate and inline the constant output of the block G2, `150.0`. The generator also inlines the value of the **Gain** parameter of the Gain block G1, `2.0`.

With the optimization, the generated code leaves out computationally expensive algorithmic code for blocks such as the lookup tables. The optimized code calculates the output of a block only if the output can change during execution. For this model, only the outputs of the Inport block In1, the Gain block G1, and the Sum block can change.

Close the model and the code generation report.

```
bdclose('rtwdemo_paraminline')
rtwdemoclean;
cd(currentDir)
```

Preserve Block Parameter Tunability

When you set **Default parameter behavior** to `Inlined`, you can preserve block parameter tunability by creating `Simulink.Parameter` objects for individual parameters. You can configure each object to appear in the code as a tunable field of the global parameter structure or as an individual global variable. You can change parameter values during code execution and interface the generated code with your own handwritten code. For more information, see “Create Tunable Calibration Parameter in the Generated Code”.

Inline Invariant Signals

You can select the **Inline invariant signals** code generation option (which also places constant values in the generated code) only when you set **Default parameter behavior** to `Inlined`. See “Inline Invariant Signals”.

See Also

Default parameter behavior

Related Examples

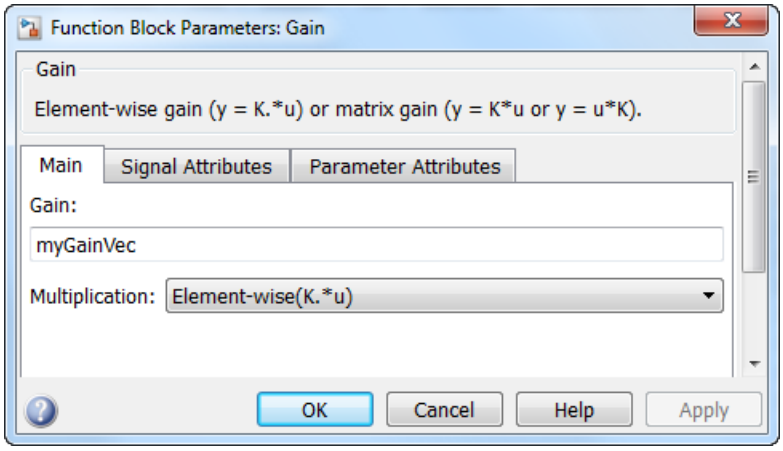
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

Configure Loop Unrolling Threshold

The **Loop unrolling threshold** parameter on the **Optimization** pane determines when a wide signal or parameter should be wrapped into a `for` loop and when it should be generated as a separate statement for each element of the signal. The default threshold value is 5.

For example, consider the model below:

The gain parameter of the Gain block is the vector `myGainVec`.



Assume that the loop unrolling threshold value is set to the default, 5.

If `myGainVec` is declared as

```
myGainVec = [1:10];
```

an array of 10 elements, `myGainVec_P.Gain_Gain[]`, is declared within the `Parameters_`*`model`* data structure. The size of the gain array exceeds the loop unrolling threshold. Therefore, the code generated for the Gain block iterates over the array in a `for` loop, as shown in the following code:

```
{
    int32_T i1;

    /* Gain: '<Root>/Gain' */
    for(i1=0; i1<10; i1++) {
      myGainVec_B.Gain_f[i1] = rtb_foo *
        myGainVec_P.Gain_Gain[i1];
    }
  }
```

If `myGainVec` is declared as

```
myGainVec = [1:3];
```

an array of three elements, `myGainVec_P.Gain_Gain[]`, is declared within the `Parameters` data structure. The size of the gain array is below the loop unrolling threshold. The generated code consists of inline references to each element of the array, as in the code below.

```
/* Gain: '<Root>/Gain' */
  myGainVec_B.Gain_f[0] = rtb_foo * myGainVec_P.Gain_Gain[0];
  myGainVec_B.Gain_f[1] = rtb_foo * myGainVec_P.Gain_Gain[1];
  myGainVec_B.Gain_f[2] = rtb_foo * myGainVec_P.Gain_Gain[2];
```

See “Explore Variable Names and Loop Rolling” for more information on loop rolling.

Note When a model includes Stateflow charts or MATLAB Function blocks, you can apply a set of Stateflow optimizations on the **Optimization** pane. The settings you select for the Stateflow options also apply to MATLAB Function blocks in the model. This is because the MATLAB Function blocks and Stateflow charts are built on top of the same technology and share a code base. You do not need a Stateflow license to use MATLAB Function blocks.

Note If your MATLAB Function block contains code with two nested `for`-loops:

- If the number of iterations of the inner loop is less than the threshold, the code generator first unrolls the inner loop. Subsequently, if the product of the number of iterations of the two loops is also less than the threshold, the code generator unrolls the outer loop. Otherwise the code generator produces the outer `for`-loop.
- If the number of iterations of the inner loop is equal to or greater than the threshold, the code generator produces both `for`-loops
- For an N-Dimensional array the code generator unrolls each contiguous loop until the product of loop iterations is less than the threshold value.

See Also

Loop unrolling threshold

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Optimize Generated Code by Combining Multiple for Constructs” on page 76-13
- “For Loop” on page 26-31

Use memcpy Function to Optimize Generated Code for Vector Assignments

In this section...

"Example Model" on page 76-48
"Generate Code" on page 76-49
"Generate Code with Optimization" on page 76-50

You can use the **Use memcpy for vector assignment** parameter to optimize generated code for vector assignments by replacing `for` loops with `memcpy` function calls. The `memcpy` function is more efficient than `for`-loop controlled element assignment for large data sets. This optimization improves execution speed.

Selecting the **Use memcpy for vector assignment** parameter enables the associated parameter **Memcpy threshold (bytes)**, which allows you to specify the minimum array size in bytes for which `memcpy` function calls should replace `for` loops in the generated code. For more information, see Use memcpy for vector assignment and Memcpy threshold (bytes). In considering whether to use this optimization,

- Verify that your target supports the `memcpy` function.
- Determine whether your model uses signal vector assignments (such as `Y=expression`) to move large amounts of data, for example, using the Selector block.

To apply this optimization,

1. Consider first generating code without this optimization and measuring its execution speed, to establish a baseline for evaluating the optimized assignment.
2. Select **Use memcpy for vector assignment** and examine the setting of **Memcpy threshold (bytes)**, which by default specifies 64 bytes as the minimum array size for which `memcpy` function calls replace `for` loops. Based on the array sizes used in your application's signal vector assignments, and target environment considerations that might bear on the threshold selection, accept the default or specify another array size.
3. Generate code, and measure its execution speed against your baseline or previous iterations. Iterate on steps 2 and 3 until you achieve an optimal result.

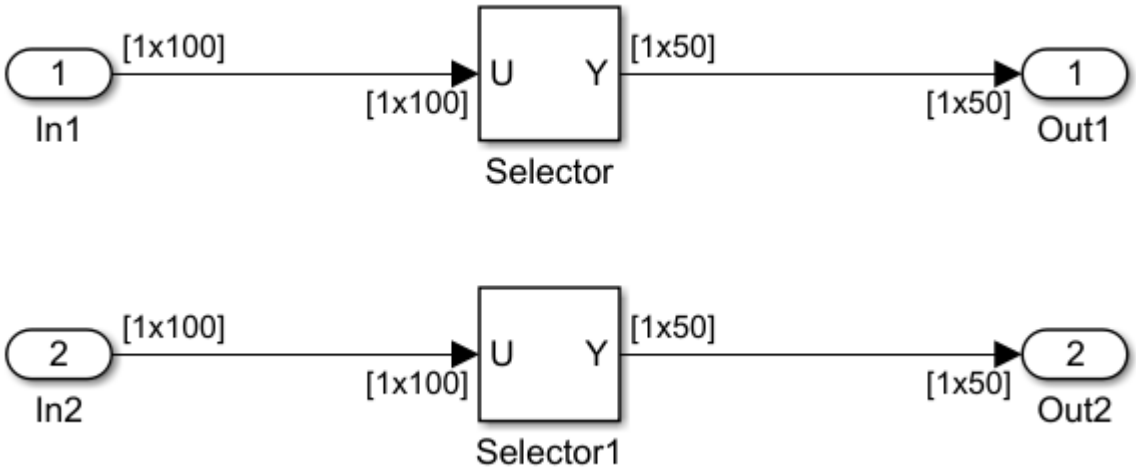
Note The `memcpy` optimization may not occur under certain conditions, including when other optimizations have a higher precedence than the `memcpy` optimization, or when the generated code is originating from Target Language Compiler (TLC) code, such as a TLC file associated with an S-function block.

Note If you are licensed for Embedded Coder software, you can use a code replacement library (CRL) to provide your own custom implementation of the `memcpy` function to be used in generated model code. For more information, see "Memory Function Code Replacement" on page 72-78.

Example Model

To examine the result of using the **Use memcpy for vector assignment** parameter on the generated vector assignment code, create a model that generates signal vector assignments. For example,

1. Use In, Out, and Selector blocks to create the following model.



2. Open Model Explorer and configure the **Signal Attributes** for the `In1` and `In2` source blocks. For each, set **Port dimensions** to `[1,100]`, and set **Data type** to `int32`. Apply the changes and save the model. In this example, the model has the name `vectorassign`.
3. For each Selector block, set the **Index** parameter to `1:50`. Set the **Input port size** parameter to `100`.

Generate Code

1. The **Use memcpy for vector assignment** parameter is on by default. To turn off the parameter, go to the **Optimization** pane and clear the **Use memcpy for vector assignment** parameter.
2. Go to the **Code Generation** > **Report** pane of the Configuration Parameters dialog box and select the **Create code generation report** parameter and the **Open report automatically** parameter. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model. When code generation completes, the HTML code generation report is displayed.
3. In the HTML code generation report, click the `vectorassign.c` section and inspect the model step function. Notice that the vector assignments are implemented using `for` loops.

```
/* Model step function */
void vectorassign_step(void)
{
  int32_T i;
  for (i = 0; i < 50; i++) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In1'
     */
    vectorassign_Y.Out1[i] = vectorassign_U.In1[i];

    /* Outport: '<Root>/Out2' incorporates:
     *  Inport: '<Root>/In2'
     */
    vectorassign_Y.Out2[i] = vectorassign_U.In2[i];
  }
}
```

Generate Code with Optimization

1. Go to the **Optimization** pane of the Configuration Parameters dialog box and select the **Use memcpy for vector assignment** option. Leave the **Memcpy threshold (bytes)** option at its default setting of `64`. Apply the changes and regenerate code for the model. When code generation completes, the HTML code generation report again is displayed.
2. In the HTML code generation report, click the `vectorassign.c` section and inspect the model output function. Notice that the vector assignments now are implemented using `memcpy` function calls.

```
/* Model step function */
void vectorassign_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  memcpy(&vectorassign_Y.Out1[0], &vectorassign_U.In1[0], 50U * sizeof(real_T));

  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In2'
   */
  memcpy(&vectorassign_Y.Out2[0], &vectorassign_U.In2[0], 50U * sizeof(real_T));
}
```

See Also

Use memcpy for vector assignment

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Vector Operation Optimization” on page 76-99
- “Convert Data Copies to Pointer Assignments” on page 80-18

Generate Target Optimizations Within Algorithm Code

Some application components are hardware-specific and cannot simulate on a host system. For example, consider a component that includes `pragmas` and assembly code for enabling hardware instructions for saturate on add operations or a Fast Fourier Transform (FFT) function.

The following table lists integration options to customize generated algorithm code with target-specific optimizations.

Note Solutions marked with *EC only* require an Embedded Coder license.

If...	Then...	For More Information, See
You want to optimize the execution speed and memory of the model code by replacing default math functions and operators with target-specific code	*EC only*—Implement function and operator replacements by using the **Code Replacement Tool**, code replacement library (CRL) API, and **Code Replacement Viewer** to create, examine, validate, and register hardware-specific replacement tables	"Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink" on page 79-31
You want to control how code generation technology declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	*EC only*—Design (create) and apply custom storage classes	• `rtwdemo_cscpredef` • `rtwdemo_importstruct` • `rtwdemo_configinterface` • "Organize Parameter Data into a Structure by Using Struct Storage Class" on page 33-67

Note To simulate an algorithm that includes target-specific elements in a host environment, you must create code that is equivalent to the target code and can run in the host environment.

Remove Code for Blocks That Have No Effect on Computational Results

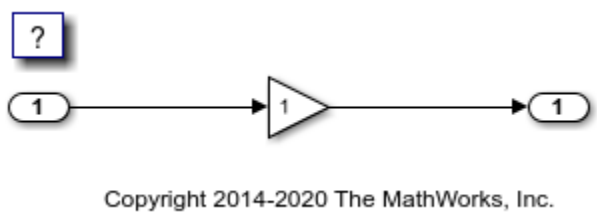
This example shows how the code generator optimizes generated code by removing code that has no effect on computational results. This optimization:

- Increases execution speed.
- Reduces ROM consumption.

Example

In the model rtwdemo_blockreduction, a Gain block of value `1.0` is in between Inport and Outport blocks.

```
model = 'rtwdemo_blockreduction';
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
```

Build the model.

```
set_param(model,'BlockReduction','off');
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_blockreduction
### Successful completion of build procedure for: rtwdemo_blockreduction

Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================================
rtwdemo_blockreduction  Code generated and compiled.  Code generation information file does not e

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.655s
```

This code is the code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir,'rtwdemo_blockreduction_ert_rtw','rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */',...
    '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void rtwdemo_blockreduction_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtwdemo_blockreduction_Y.Out1 = 1.0 * rtwdemo_blockreduction_U.In1;
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. If the **Block reduction** check box is not selected, select it.

Alternatively, use the command-line API to enable the optimization.

```
set_param(model,'BlockReduction','on');
```

Generate Code with Optimization

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_blockreduction
### Successful completion of build procedure for: rtwdemo_blockreduction

Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
rtwdemo_blockreduction  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.7152s
```

Here is the optimized code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir,'rtwdemo_blockreduction_ert_rtw','rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */',...
    '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void rtwdemo_blockreduction_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtwdemo_blockreduction_Y.Out1 = rtwdemo_blockreduction_U.In1;
}
```

Because multiplying the input signal by a value of `1.0` does not impact computational results, the code generator excludes the Gain block from the generated code. Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Limitations

The code generator cannot remove blocks under these circumstances:

- The block output is configured as a test point or has a storage class and the block input and output have different data types, complexities, or widths.
- The block has a nonconstant sample time, has an output connected to a root outport, and has a source that has a constant sample time.
- The block has multiple sources, some of which have constant sample times and some of which have nonconstant sample times.
- The block output has a storage class and has a source that is muxed, is a virtual bus, is connected to a root outport, is configured as a test point, or has a storage class.
- The block has an output that is connected to the block input.
- The block is inside an Initialize Function subsystem, Reset Function subsystem, or Terminate Function subsystem whose output is connected to an outport of the subsystem.
- The block has an output that is connected to a Subsystem Outport block that has an initial condition.
- The block has an output that is connected to an outport of an Initialize Function subsystem, Reset Function subsystem, or Terminate Function subsystem.
- The block has an output that is connected to an outport for which the parameter **Ensure outport is virtual** is selected.
- The block has an output that is connected to an outport that has an initial condition parameter.
- The block has an output that is connected to a Merge, a Vector Concatenate, or a Matrix Concatenate block.
- The block has tunable run-time parameters.
- The block is inside a reusable subsystem whose output is connected to a block outside the reusable subsystem.
- The block has a continuous state.
- The block has a discrete state and the states are logged or have a nonauto storage class.
- The block is an S-Function, unless you use `ssSetBlockReduction` to enable block reduction.
- The block output is configured as a test point or has a storage class and its source is a block inside a different conditional subsystem.

See Also

"Block reduction"

Related Examples

- "Design Techniques to Optimize Models for Efficient Code Generation" on page 76-7
- "Inline Numeric Values of Block Parameters" on page 76-41

- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35

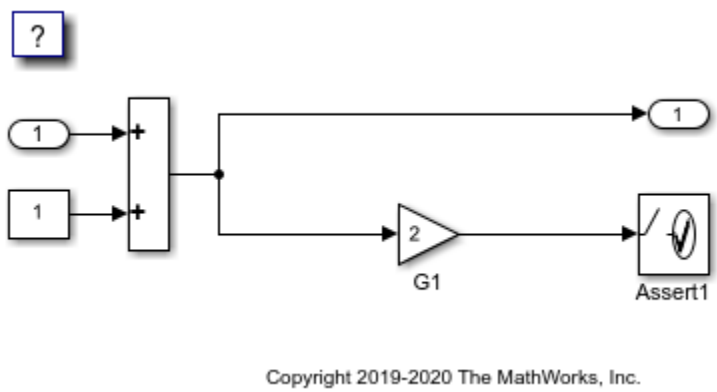
Eliminate Dead Code Paths in Generated Code

This example shows how the code generator eliminates dead (that is, unused) code paths from generated code. This optimization increases execution speed and conserves ROM and RAM consumption.

Example

In the `DeadPathElimination` model, the signal leaving the Sum block divides into two separate code paths. The top path is not a dead code path. If the user disables the Assertion block, the bottom path becomes a dead code path.

```
model = 'DeadPathElimination';
open_system(model);
```



Generate Code with an Enabled Assertion Block

1. For the Assertion block, open the block parameters dialog box.
2. Select the **Enable assertion** box. Alternatively, use the command-line API to enable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'on');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: DeadPathElimination
### Successful completion of build procedure for: DeadPathElimination

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
DeadPathElimination  Code generated and compiled.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.78s
```

Because the Assertion block is enabled, these lines of `DeadPathElimination.c` include code for the Gain and Assertion blocks.

```
cfile = fullfile('DeadPathElimination_grt_rtw','DeadPathElimination.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 0, 1);
```

```
void DeadPathElimination_step(void)
{
  /* Sum: '<Root>/Sum1' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In1'
   */
  DeadPathElimination_Y.Out1 = DeadPathElimination_U.In1 + 1.0;

  /* Assertion: '<Root>/Assert1' incorporates:
   *  Gain: '<Root>/G1'
   */
  utAssert(2.0 * DeadPathElimination_Y.Out1 != 0.0);
}
```

Generate Code with a Disabled Assertion Block

Disable the Assertion block to generate a dead code path. The code generator detects the dead code path and eliminates it from the generated code.

1. For the Assertion block, open the Block Parameters dialog box.
2. Deselect the **Enable assertion** box.

Alternatively, use the command-line API to disable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'off');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: DeadPathElimination
### Successful completion of build procedure for: DeadPathElimination

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
DeadPathElimination  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.738s
```

Because the Assertion block is disabled, these lines of `DeadpathElim.c` do not include code for the Gain and Assertion blocks.

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 0, 1);
```

```
void DeadPathElimination_step(void)
```

```
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum1'
   */
  DeadPathElimination_Y.Out1 = DeadPathElimination_U.In1 + 1.0;
}
```

Close the Model and Clean Up

```
bdclose(model)
rtwdemoclean;
```

For another example of how the code generator eliminates dead code paths in the generated code, see rtwdemo_deadpath.

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Remove Code for Blocks That Have No Effect on Computational Results” on page 76-52

Floating-Point Multiplication to Handle a Net Slope Correction

This example shows how to use floating-point multiplication to handle a net slope correction. When converting floating-point data types to fixed-point data types in the generated code, a net slope correction is one method of scaling fixed-point data types. Scaling the fixed-point data types avoids overflow conditions and minimizes quantization errors.

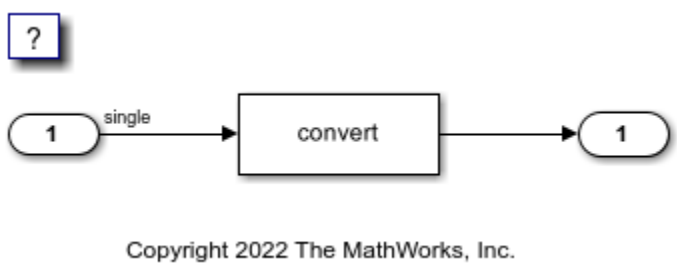
For processors that support efficient multiplication, using floating-point multiplication to handle a net slope correction improves code efficiency. If the net slope correction has a value that is not a power of two, using division improves precision.

Note: This example requires a Fixed-Point Designer™ license.

Example

In the model `FloatMultiplicationNetSlope`, a Convert block converts an input signal from a floating-point data type to a fixed-point data type. The net slope correction has a value of 3.

```
model = 'FloatMultiplicationNetSlope';
open_system(model);
```



Generate Code

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
==============================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.223s
```

In these lines of `FloatMultiplicationNetSlope.c` code, the code generator divides the input signal by `3.0F` .

```
cfile = fullfile('FloatMultiplicationNetSlope_grt_rtw',...
    'FloatMultiplicationNetSlope.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/Input'
   */
  FloatMultiplicationNetSlope_Y.Output = (int16_T)(real32_T)floor
    (FloatMultiplicationNetSlope_U.Input / 3.0F);
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Math and Data Types** pane, select **Use floating-point multiplication to handle net slope corrections**. This optimization is on by default.

Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'UseFloatMulNetSlope', 'on');
```

Generate Code with Optimization

```
slbuild(model)
```

```
### Starting build procedure for: FloatMultiplicationNetSlope
### Successful completion of build procedure for: FloatMultiplicationNetSlope

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
FloatMultiplicationNetSlope  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.6138s
```

In the optimized code, the code generator multiplies the input signal by the reciprocal of `3.0F` , that is `0.333333343F` .

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void FloatMultiplicationNetSlope_step(void)
{
  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/Input'
   */
  FloatMultiplicationNetSlope_Y.Output = (int16_T)(real32_T)floor
```

```
    (FloatMultiplicationNetSlope_U.Input * 0.333333343F);
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
```

See Also

“Use floating-point multiplication to handle net slope corrections”

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Remove Code for Out-of-Range Floating Point to Integer Conversions” on page 76-20
- “Subnormal Number Execution Speed” on page 76-16

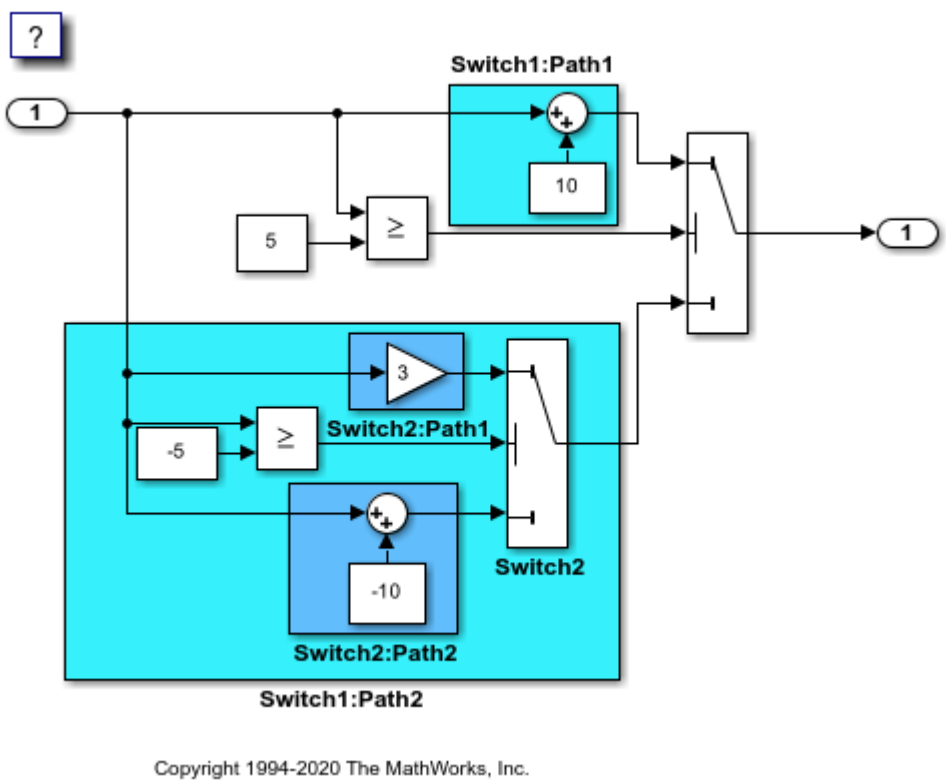
Use Conditional Input Branch Execution

This example shows how to optimize the generated code for a model that contains Switch and Multiport Switch blocks. When you select the model configuration parameter **Conditional input branch execution**, Simulink® executes only blocks that compute the control input and data input that the control input selects. This optimization improves execution speed.

Example Model

In this example, switch paths are conditionally executed. If `Switch1` control input is true, `Switch1` executes blocks grouped in the `Switch1:Path1` branch. If `Switch1` control input is false, `Switch1` executes blocks grouped in the `Switch1:Path2` branch. If `Switch1` executes blocks in the `Switch1:Path2` branch and `Switch2` control input is true, `Switch2` executes blocks in the `Switch2:Path1` branch. If `Switch2` control input is false, `Switch2` executes blocks in the `Switch2:Path2` branch. The pseudo code shows this logic.

```
model='ConditionalInput';
open_system(model);
```



Generate Code

The **Conditional input branch execution** parameter is on by default. Enter the following command-line API to turn off the parameter.

```
set_param(model, 'ConditionallyExecuteInputs', 'off');
```

Build the model

```
slbuild(model)
```

```
### Starting build procedure for: ConditionalInput
### Successful completion of build procedure for: ConditionalInput

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
========================================================================================================
ConditionalInput  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.486s
```

View the generated code without the optimization. These lines of code are in the `ConditionalInput.c` file.

```
cfile = fullfile('ConditionalInput_grt_rtw','ConditionalInput.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void ConditionalInput_step(void)
{
  /* Switch: '<Root>/ Switch2' incorporates:
   *  Constant: '<Root>/C_5'
   *  Inport: '<Root>/input'
   *  RelationalOperator: '<Root>/Relational Operator'
   */
  if (ConditionalInput_U.input >= -5.0) {
    /* Switch: '<Root>/Switch1' incorporates:
     *  Gain: '<Root>/  G3'
     */
    ConditionalInput_Y.output = 3.0 * ConditionalInput_U.input;
  } else {
    /* Switch: '<Root>/Switch1' incorporates:
     *  Sum: '<Root>/ Sum'
     */
    ConditionalInput_Y.output = ConditionalInput_U.input - 10.0;
  }

  /* End of Switch: '<Root>/ Switch2' */

  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/C5'
   *  Inport: '<Root>/input'
   *  RelationalOperator: '<Root>/Relational Operator1'
   */
  if (ConditionalInput_U.input >= 5.0) {
    /* Switch: '<Root>/Switch1' incorporates:
     *  Constant: '<Root>/  C10'
     *  Outport: '<Root>/output'
     *  Sum: '<Root>/ Sum1'
     */
```

```
    ConditionalInput_Y.output = ConditionalInput_U.input + 10.0;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

The generated code contains an `if-else` statement for the `Switch2` block and an `if` statement for the `Switch1` block. Therefore, the generated code for `Switch1:Path2` executes even if the `if` statement for `Switch1:Path1` evaluates to true.

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. Select the **Conditional input branch execution** parameter. Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'ConditionallyExecuteInputs','on');
```

Generate Code with Optimization

```
slbuild(model)
cfile = fullfile('ConditionalInput_grt_rtw','ConditionalInput.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
### Starting build procedure for: ConditionalInput
### Successful completion of build procedure for: ConditionalInput

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
================================================================================
ConditionalInput  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.6076s

/* Model step function */
void ConditionalInput_step(void)
{
  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/C5'
   *  Constant: '<Root>/C_5'
   *  Inport: '<Root>/input'
   *  RelationalOperator: '<Root>/Relational Operator'
   *  RelationalOperator: '<Root>/Relational Operator1'
   *  Switch: '<Root>/ Switch2'
   */
  if (ConditionalInput_U.input >= 5.0) {
    /* Outport: '<Root>/output' incorporates:
     *  Constant: '<Root>/  C10'
     *  Sum: '<Root>/ Sum1'
     */
    ConditionalInput_Y.output = ConditionalInput_U.input + 10.0;
  } else if (ConditionalInput_U.input >= -5.0) {
    /* Switch: '<Root>/ Switch2' incorporates:
     *  Gain: '<Root>/  G3'
```

```
     *  Outport: '<Root>/output'
     */
    ConditionalInput_Y.output = 3.0 * ConditionalInput_U.input;
  } else {
    /* Outport: '<Root>/output' incorporates:
     *  Sum: '<Root>/ Sum'
     *  Switch: '<Root>/ Switch2'
     */
    ConditionalInput_Y.output = ConditionalInput_U.input - 10.0;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

The generated code contains one `if` statement. The generated code for `Switch1:Path2` only executes if the `if` statement evaluates to false.

Close Model and Code Generation Report

```
bdclose(model)
rtwdemoclean;
```

See Also

“Conditional input branch execution” | Switch | Multiport Switch

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Eliminate Dead Code Paths in Generated Code” on page 76-56

Optimize Generated Code for Complex Signals

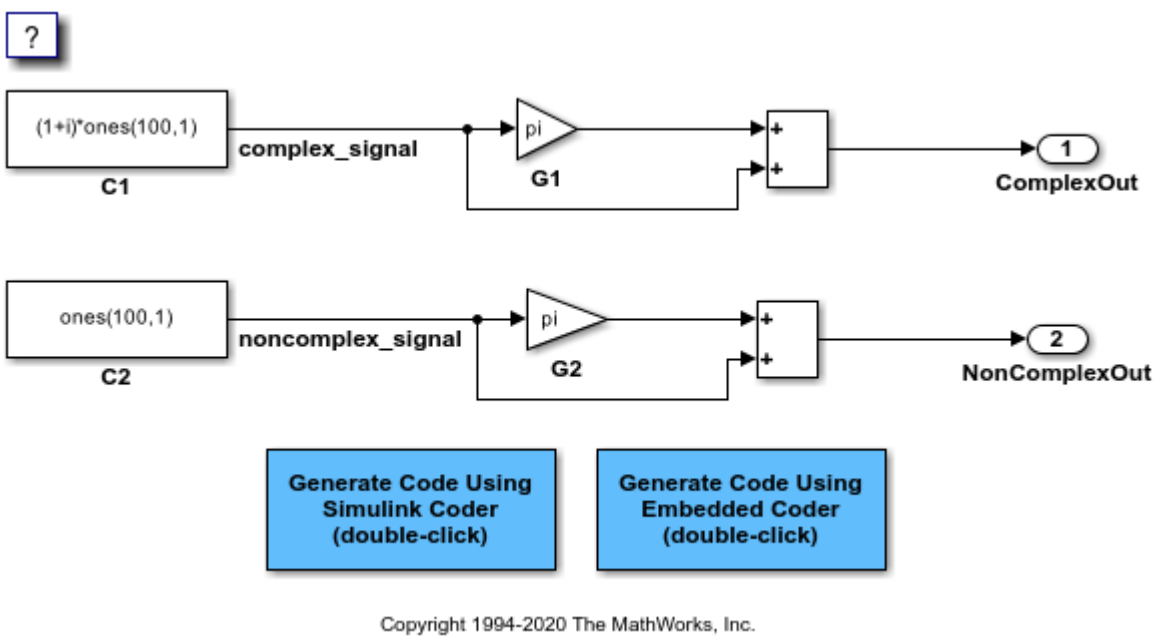
This example shows how Simulink® Coder™ handles complex signals efficiently. To view the data types of the signals, update the model. On the **Debug** menu, select **Update Model**. Complex signals are represented as structures in generated code. Simulink Coder performs various optimizations on these structures. For example:

- Expression Folding: Gain and Sum operations on the complex signal are folded into a single expression.
- For-loop fusion: Two separate for-loops, one for the complex signal and one for noncomplex signal, are combined into a single for-loop.
- Inlined block parameters: The value of Gain block "pi" is inlined in the expression of the complex Gain-Sum.

Because of optimizations such as these, the code generated for complex and noncomplex signals is equally efficient.

Example Model

```
model='rtwdemo_complex';
open_system(model);
```



See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35

- “Disable Nonfinite Checks or Inlining for Math Functions” on page 76-26

Speed Up Linear Algebra in Code Generated from a MATLAB Function Block

To improve the execution speed of code generated for certain linear algebra functions in a MATLAB Function block, specify that the code generator produce LAPACK calls. LAPACK is a software library for numerical linear algebra. The code generator uses the LAPACKE C interface to LAPACK. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet certain criteria, the code generator produces the LAPACK calls. Otherwise, the code generator produces code for the linear algebra functions.

The code generator uses the LAPACK library that you specify. Specify a LAPACK library that is optimized for your execution environment. See netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations.

Specify LAPACK Library

To generate LAPACK calls, you must have access to a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file for the LAPACK calls. To indicate that you want to generate LAPACK calls and that you want to use a specific LAPACK library, specify the name of the LAPACK callback class. In the Configuration Parameters dialog box, set **Custom LAPACK library callback** to the name of the callback class, for example, `useMyLAPACK`.

Write LAPACK Callback Class

To specify the locations of a particular LAPACK library and LAPACKE header file, write a LAPACK callback class. Share the callback class with others who want to use this LAPACK library for LAPACK calls in generated code.

The callback class must derive from the abstract class `coder.LAPACKCallback`. Use this example callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
            buildInfo.addDefines('LAPACK_ILP64');
        end
    end
end
```

You must provide the `getHeaderFilename` and `updateBuildInfo` methods. The `getHeaderFilename` method returns the LAPACKE header file name. In the example callback class, replace `mylapacke_custom.h` with the name of your LAPACKE header file. The `updateBuildInfo`

method provides the information required for the build process to link to the LAPACK library. Use code like the code in the template to specify the location of header files and the full path name of the LAPACK library. In the example callback class, replace `mylapack` with the name of your LAPACK library.

If your compiler supports only complex data types that are represented as structures, include these lines in the `updateBuildInfo` method.

```
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

You must specify the integer type that your LAPACK library uses. Not specifying this integer type can result in incorrect behaviors or crashes. Do one of the following:

- Include these lines in the `updateBuildInfo` method.

  ```
  buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
  buildInfo.addDefines('LAPACK_ILP64');
  ```

- Alternatively, you can directly specify the integer type that your LAPACK library uses. For example, if the integer type is `long long`, include this line in the `updateBuildInfo` method.

  ```
  buildInfo.addDefines('lapack_int=long long');
  ```

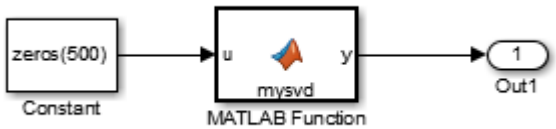
Generate LAPACK Calls by Specifying a LAPACK Callback Class

This example shows how to generate code that calls LAPACK functions in a specific LAPACK library. For this example, assume that the LAPACK callback class `useMyLAPACK` specifies the LAPACK library that you want.

1. Create a Simulink model.
2. Add a MATLAB Function block to the model.
3. In the MATLAB Function block, add code that calls a linear algebra function. For example, add the function `mysvd` that calls the MATLAB function `svd`.

   ```
   function s = mysvd(A)
       %#codegen
       s = svd(A);
   end
   ```

4. Add a Constant block to the left of the MATLAB Function block. Set the value to `zeros(500)`.
5. Add an Outport block to the right of the MATLAB Function block.
6. Connect the blocks.



7. Set the **Configuration Parameters** > **Code Generation** > **Advanced parameters** > **Custom LAPACK library callback** parameter to `useMyLAPACK`.

   The callback class must be on the MATLAB path.
8. Build the model.

If the input to `mysvd` is large enough, the code generator produces a LAPACK call for `svd`. An example of a call to the LAPACK library function for `svd` is:

```
info_t = LAPACKE_dgesvd( ...
     LAPACK_COL_MAJOR, 'N', 'N', (lapack_int)500, ...
     (lapack_int)500, &A[0], (lapack_int)500, &S[0], ...
     NULL, (lapack_int)1, NULL,(lapack_int)1, &superb[0]);
```

Locate LAPACK Library in Execution Environment

The LAPACK library must be available in your execution environment. If your LAPACK library is shared, use environment variables or linker options to specify the location of the LAPACK library.

- On a Windows platform, modify the PATH environment variable.
- On a Linux platform, modify the LD_LIBRARY_PATH environment variable or use the `rpath` linker option.
- On a macOS platform, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, you can use the build information `addLinkFlags` method in the `updateBuildInfo` method of your `coder.LAPACKCallback` class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

See Also

`coder.LAPACKCallback`

More About

- “LAPACK Calls for Linear Algebra in a MATLAB Function Block”

External Websites

- www.netlib.org/lapack
- www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations

Speed Up Matrix Operations in Code Generated from a MATLAB Function Block

To improve the execution speed of code generated for certain low-level vector and matrix operations (such as matrix multiplication) in a MATLAB Function block, specify that you want the code generator to produce BLAS calls. BLAS is a software library for low-level vector and matrix computations that has several highly optimized machine-specific implementations. The code generator uses the CBLAS C interface to BLAS. If you specify that you want to generate BLAS calls, and the input arrays for the matrix functions meet certain criteria, the code generator produces the BLAS calls. Otherwise, the code generator produces code for the matrix functions.

The code generator uses the BLAS library that you specify. Specify a BLAS library that is optimized for your execution environment.

Specify BLAS Library

To produce BLAS calls in generated code, you must have access to a BLAS callback class. A BLAS callback class specifies the BLAS library, the CBLAS header file, certain C data types the particular CBLAS interface uses and the compiler and linker options for the build process.

To indicate that you want to generate BLAS calls and use a specific BLAS library, specify the name of the BLAS callback class. In the Configuration Parameters dialog box, set **Custom BLAS library callback** to the name of the callback class.

Write BLAS Callback Class

To generate calls to a specific BLAS library in the generated code, write a BLAS callback class. Share the callback class with others who want to use this BLAS library for BLAS calls in standalone code.

The callback class must derive from the abstract class `coder.BLASCallback`. This example is an implementation of the callback class `mklcallback` for integration with the Intel MKL BLAS library on a Windows platform.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, ...
                                     libLinkOnly);
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin', ...
                                 'win64'), libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
            buildInfo.addDefines('-DMKL_ILP64');
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
            intTypeName = 'MKL_INT';
```

```
        end
    end
end
```

You must provide the `getHeaderFilename`, `getBLASIntTypeName`, and `updateBuildInfo` methods. The `getHeaderFilename` method returns the CBLAS header file name. If you are using a different BLAS library, replace `mkl_cblas.h` with the name of your CBLAS header file. The `getBLASIntTypeName` method returns the name of the integer data type that your CBLAS interface uses. If you are using a different BLAS library, replace `MKL_INT` with the name of the integer data type specific to your CBLAS interface. The `updateBuildInfo` method provides the information required for the build process to link to the BLAS library. Use code that is like the code in the example callback class to specify the location of header file, the full path name of the BLAS library, and the compiler and linker options. If you use the Intel MKL BLAS library, use the link line advisor to see which libraries and compiler options are recommended for your use case.

There are three other methods that are already implemented in `coder.BLASCallback`. These methods are `getBLASDoubleComplexTypeName`, `getBLASSingleComplexTypeName`, and `useEnumNameRatherThanTypedef`. By default, your callback class inherits these implementations from `coder.BLASCallback`. In certain situations, you must override these methods with your own definitions when you define your callback class.

The `getBLASDoubleComplexTypeName` method returns the type used for double-precision complex variables in the generated code. If your BLAS library takes a type other than `double*` and `void*` for double-precision complex array arguments, include this method in your callback class definition.

```
function doubleComplexTypeName = getBLASDoubleComplexTypeName()
doubleComplexTypeName = 'my_double_complex_type';
end
```

Replace `my_double_complex_type` with the type that your BLAS library takes for double-precision complex array arguments.

The `getBLASSingleComplexTypeName` method returns the type used for single-precision complex variables in the generated code. If your BLAS library takes a type other than `float*` and `void*` for single-precision complex array arguments, include this method in your callback class definition.

```
function singleComplexTypeName = getBLASSingleComplexTypeName()
doubleComplexTypeName = 'my_single_complex_type';
end
```

Replace `my_single_complex_type` with the type that your BLAS library takes for single-precision complex array arguments.

The `useEnumNameRatherThanTypedef` method returns `false` by default. If types for enumerations in your BLAS library include the `enum` keyword, redefine this method to return `true` in your callback class definition.

```
function p = useEnumNameRatherThanTypedef()
p = true;
end
```

An excerpt from generated C source code that includes the `enum` keyword is:

```
enum CBLAS_SIDE t;
enum CBLAS_UPLO b_t;
double temp;
```

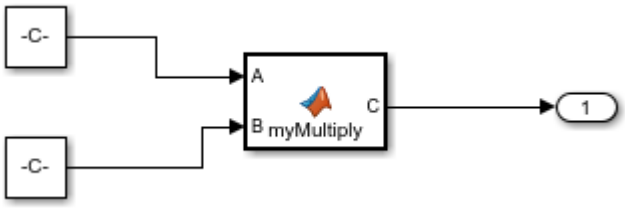
```
enum CBLAS_TRANSPOSE c_t;
enum CBLAS_DIAG d_t;
```

Generate BLAS Calls by Specifying a BLAS Callback Class

This example shows how to generate code that calls BLAS functions in a specific BLAS library. The BLAS callback class `useMyBLAS` specifies the BLAS library that you want to use in this example.

1. Create a Simulink model.
2. Add a MATLAB Function block to the model.
3. In the MATLAB Function block, add code that calls a function for a basic matrix operation. For example, add the function `myMultiply` that multiplies two matrices `A` and `B`.

   ```
   function C = myMultiply(A,B) %#codegen
   C = A*B;
   end
   ```
4. Add two Constant blocks to the left of the MATLAB Function block. Set their values to `zeros(1000)`.
5. Add an Outport block to the right of the MATLAB Function block.
6. Connect the blocks.



7. In the Configuration Parameters dialog box, set **Custom BLAS library callback** to the name of the callback class `useMyBLAS`.

   The callback class must be on the MATLAB path.
8. Build the model.

   If `A` and `B` are large enough, the code generator produces a BLAS call for the matrix multiplication function.

Locate BLAS Library in Execution Environment

The BLAS library must be available in your execution environment. If your BLAS library is shared, use environment variables or linker options to specify the location of the BLAS library.

- On a Windows platform, modify the PATH environment variable.
- On a Linux platform, modify the LD_LIBRARY_PATH environment variable or use the `rpath` linker option.
- On a macOS platform, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, use the build information `addLinkFlags` method in the `updateBuildInfo` method of your BLAS callback class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

Usage Notes and Limitations for OpenBLAS Library

If you generate code that includes calls to the OpenBLAS library functions, follow these guidelines and restrictions:

- If you generate C++ code that includes calls to OpenBLAS library functions, compiling it with the `-pedantic` option produces warnings. To disable the `-pedantic` compiler option, include these lines in the `updateBuildInfo` method:

  ```
  if ctx.getTargetLang() == 'C++'
      buildInfo.addCompileFlags('-Wno-pedantic');
  end
  ```

- OpenBLAS does not support the C89/C90 standard.

See Also

`coder.BLASCallback`

More About

- "BLAS Calls for Matrix Operations in a MATLAB Function Block"

External Websites

- https://netlib.org/blas/
- https://netlib.org/blas/faq.html#_5_a_id_are_optimized_blas_libraries_available_where_can_i_find_vendor_supplied_blas_a_are_optimized_blas_libraries_available_where_can_i_find_optimized_blas_libraries
- https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
- https://www.intel.com/content/www/us/en/develop/tools/oneapi/onemkl/link-line-advisor.html

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

When you simulate a model that includes a MATLAB Function block that calls MATLAB fast Fourier transform (FFT) functions, the simulation software uses the library that MATLAB uses for FFT algorithms. If you generate C/C++ code for this model, by default, the code generator produces code for the FFT algorithms instead of producing FFTW library calls. To increase the speed of fast Fourier transforms in generated code, specify that the code generator produce calls to the FFTW library. For more information about FFTW, see www.fftw.org.

To produce calls to the FFTW library instead of generating FFT algorithms, you can choose from these configuration parameters in **Configuration Parameters** > **Code Generation** under the **Advanced Parameters**:

- Select **Built-in FFTW library callback** to use the FFTW library included with MATLAB. This instructs the code generator to use the built-in callback for FFTW library calls in the generated code.
- Select **Custom FFT library callback** to use the specific installed FFTW library.

The code generator produces FFTW library calls when:

- A MATLAB Function block calls one of these MATLAB functions: `fft`, `fft2`, `fftn`, `ifft`, `ifft2`, or `ifftn`.
- You generate C/C++ code from a model that includes the MATLAB Function block.

Use Built-in FFTW Library Callback for FFTW Library Usage

If you select **Built-in FFTW library callback**, the code generator produces FFTW library calls using the FFTW library version 3.3.8 included with the MATLAB.

You don't need to perform the additional steps required in **Custom FFT library callback** if you select this parameter during code generation.

Use Custom FFT Library Callback for FFTW Library Usage

If you select **Custom FFT library callback**, the code generator produces FFTW library calls when all of these conditions are true:

- You have access to an FFTW library installation, version 3.2 or later.
- You specify the FFTW library installation in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.
- You set the **Custom FFT library callback** configuration parameter to the name of the callback class.

Install an FFTW Library

If you do not have access to an installed FFTW library, version 3.2 or later, then you must install one.

For a Linux platform or a Mac platform, consider using a package manager to install the FFTW library.

For a Windows platform, in addition to `.dll` files, you must have `.lib` import libraries, as described in the Windows installation notes on the FFTW website.

See the installation instructions for your platform on the FFTW website.

Write FFT Callback Class

To specify your installation of the FFTW library, write an FFT callback class. Share the callback class with others who want to use this FFTW library for FFTW calls.

The callback class must derive from the abstract class `coder.fftw.StandaloneFFTW3Interface`. Use this example callback class as a template.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

Implement the `updateBuildInfo` and `getNumThreads` methods. In the `updateBuildInfo` method, set `fftwLocation` to the full path for your installation of the library. Set `includePath` to the full path of the folder that contains `fftw3.h`. Set `libPath` to the full path of the folder that contains the library files. If your FFTW installation uses multiple threads, modify the `getNumThreads` method to return the number of threads that you want to use.

Optionally, you can implement these methods:

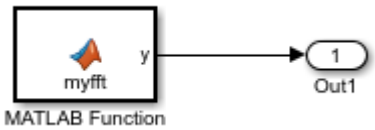
- `getPlanMethod` to specify the FFTW planning method. See `coder.fftw.StandaloneFFTW3Interface`.
- `lock` and `unlock` to synchronize multithreaded access to the FFTW planning process. See “Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block”.

Generate FFTW Calls by Specifying an FFT Callback Class

1. Create a Simulink model.
2. Add a MATLAB Function block to the model.
3. In the MATLAB Function block, add code that calls a MATLAB FFT function. For example, add the function `myfft` that calls the MATLAB function `fft`.

```
function y = myfft()
t = 0:1/50:10-1/50;
x = sin(2*pi*15*t) + sin(2*pi*20*t);
y = fft(x);
end
```

4 Connect the blocks.



5 Indicate that the code generator produce calls to the FFTW library specified in the FFT library callback class `useMyFFTW`. In the Configuration Parameters dialog box, set **Custom FFT library callback** to `useMyFFTW`.

   The callback class must be on the MATLAB path.

6 Build the model.

Locate FFTW Library in Execution Environment

The FFTW library must be available in your execution environment. If the FFTW library is shared, use environment variables or linker options to specify the location of the library.

- On a Windows platform, modify the PATH environment variable.
- On a Linux platform, modify the LD_LIBRARY_PATH environment variable or use the `rpath` linker option.
- On a macOS platform, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, you can use the build information `addLinkFlags` method in the `updateBuildInfo` method of your `coder.fftw.StandaloneFFTW3Interface` class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

See Also

`coder.fftw.StandaloneFFTW3Interface`

More About

- “Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block”

External Websites

- www.fftw.org

Synchronize Multithreaded FFTW Planning in Code Generated from a MATLAB Function Block

This example shows how to generate code that synchronizes multithreaded access to the FFTW planning process for FFTW library calls in code generated from a MATLAB Function block.

The code generator produces FFTW library calls when all of these conditions are true:

- A MATLAB Function block calls one of these functions:`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, or `ifftn`.
- You generate C/C++ code for a model that includes the MATLAB Function block.
- You have access to an FFTW library installation, version 3.2 or later.
- You specify the FFTW library installation in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.
- You set the **Custom FFT library callback** configuration parameter to the name of the callback class.

If you integrate the code that contains the FFTW calls with external code that runs on multiple threads, then you must prevent concurrent access to the FFTW planning process. In your FFT library callback class, implement the `lock` and `unlock` methods. You must also provide C code that manages a lock or mutex. Many libraries, such as OpenMP, pthreads, and the C++ standard library (C++ 11 and later), provide locks. This example shows how to implement the `lock` and `unlock` methods and provide supporting C code. To manage a lock, this example uses the OpenMP library.

Prerequisites

Before you start, for the basic workflow for generating FFTW library calls for fast Fourier transforms in a MATLAB Function block, see “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block”.

You must have:

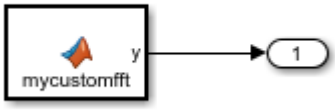
- Access to an installed FFTW library.
- A compiler that supports the OpenMP library. To use a different library, such as pthreads, modify the supporting C code accordingly.

Create a Model with a MATLAB Function Block That Calls an FFT Function

1. Create a Simulink model and add a MATLAB Function block to it.
2. Add this code to the MATLAB Function block.

```
function y  = mycustomfft()

t = 0:1/50:10-1/50;
x = sin(2*pi*15*t) + sin(2*pi*20*t);
y = fft(x);
for k = 1:100
    y = y + ifft(x+k);
end
```

3 Add an `outport` block and connect it to the MATLAB Function block.



Write Supporting C Code

Write C functions that initialize, set, and unset a lock. This example uses the OpenMP library to manage the lock. For a different library, modify the code accordingly.

- Create a file `mylock.c` that contains this C code:

```
#include "mylock.h"
#include "omp.h"

static omp_nest_lock_t lockVar;

void mylock_initialize(void)
{
    omp_init_nest_lock(&lockVar);
}

void mylock(void)
{
    omp_set_nest_lock(&lockVar);
}

void myunlock(void)
{
    omp_unset_nest_lock(&lockVar);
}
```

- Create a header file `mylock.h` that contains:

```
#ifndef MYLOCK_H
#define MYLOCK_H

 void mylock_initialize(void);
 void mylock(void);
 void myunlock(void);

#endif
```

Create an FFT Library Callback Class

Write an FFT callback class `myfftcb` that:

- Specifies the FFTW library.
- Implements `lock` and `unlock` methods that call the supporting C code to control access to the FFTW planning.

Use this class as a template. Replace `fftwLocation` with the location of your FFTW library installation.

```
classdef myfftcb < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function lock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('mylock');
        end

        function unlock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('myunlock');
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '\usr\lib\fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
            libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);

        end
    end
end
```

Configure Code Generation Parameters and Build the Model

1. Configure code generation to use the FFTW callback class and the C code called by the `lock` and `unlock` methods. Configure code generation to generate a call to `mylock_initialize` in the initialization code.

   In the Configuration Parameters dialog box:

   - Set **Custom FFT library callback** to `myfftcb`.
   - In **Code Generation > Custom Code**, under **Additional build information**, set **Source files** to `mylock.c`.

- In **Code Generation > Custom Code**, under **Insert custom C code in generated**, set **Initialize function** to `mylock_initialize();`.

2 Build the model.

See Also

`coder.fftw.StandaloneFFTW3Interface`

More About

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

External Websites

- www.fftw.org

Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block

Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. You can use dynamic memory allocation for arrays inside a MATLAB Function block.

You cannot use dynamic memory allocation for:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.
- Discrete state properties of System objects associated with a MATLAB System block.

Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

Dynamic memory allocation and the freeing of this memory can result in slower execution of the generated code. To control the use of dynamic memory allocation for variable-size arrays in a MATLAB Function block, you can:

- Provide upper bounds for variable-size arrays on page 76-82.
- Disable dynamic memory allocation for MATLAB Function blocks on page 76-82.
- Modify the dynamic memory allocation threshold on page 76-82.

Provide Upper Bounds for Variable-Size Arrays

For an unbounded variable-size array, the code generator allocates memory dynamically on the heap. For a bounded variable-size array, if the size, in bytes, is less than the dynamic memory allocation threshold, the code generator allocates memory statically on the stack. To avoid dynamic memory allocation, provide upper bounds for the array dimensions so that the size of the array, in bytes, is less than the dynamic memory allocation threshold. See “Specify Upper Bounds for Variable-Size Arrays”.

Disable Dynamic Memory Allocation for MATLAB Function Blocks

By default, dynamic memory allocation for MATLAB Function blocks is enabled for GRT-based targets and disabled for ERT-based targets. To change the setting, in the Configuration Parameters dialog box, clear or select **Dynamic memory allocation in MATLAB functions**.

If you disable dynamic memory allocation, you must provide upper bounds for variable-size arrays.

Modify the Dynamic Memory Allocation Threshold

Instead of disabling dynamic memory allocation for all variable-size arrays, you can use the dynamic memory allocation threshold to specify when the code generator uses dynamic memory allocation.

Use the dynamic memory allocation threshold to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. However, static memory allocation can lead to unused storage space. You can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

The default value of the dynamic memory allocation threshold is 64 kilobytes. To change the threshold, in the Configuration Parameters dialog box, set the **Dynamic memory allocation threshold in MATLAB functions** parameter.

To use dynamic memory allocation for all variable-size arrays, set the threshold to `0`.

See Also

MATLAB Function

Related Examples

- “Code Generation for Variable-Size Arrays”
- “Specify Upper Bounds for Variable-Size Arrays”
- “Use Dynamic Memory Allocation for Variable-Size Arrays in a MATLAB Function Block”

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

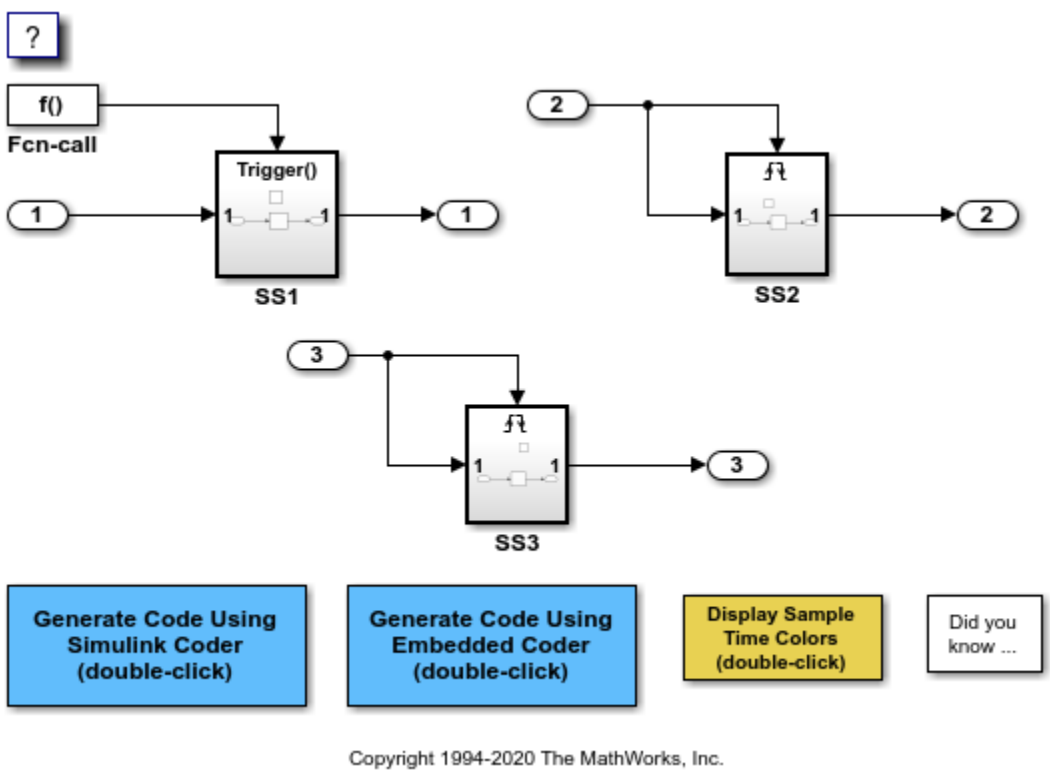
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



The model consists of three subsystems `SS1`, `SS2`, and `SS3`. Open the Model Configuration Parameters dialog box. On the **Math and Data Types** pane, the **Application lifespan (days)** parameter is set to the default, which is `inf`.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.

2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.

3. Build the model.

```
### Starting build procedure for: rtwdemo_abstime
### Successful completion of build procedure for: rtwdemo_abstime

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================
rtwdemo_abstime  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.921s
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.h`.

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
```

```
  struct {
    uint32_T clockTick1;
    uint32_T clockTickH1;
    uint32_T clockTick2;
    uint32_T clockTickH2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};

/* Block states (default storage) */
extern DW_rtwdemo_abstime_T rtwdemo_abstime_DW;

/* Zero-crossing (trigger) state */
extern PrevZCX_rtwdemo_abstime_T rtwdemo_abstime_PrevZCX;

/* External inputs (root inport signals with default storage) */
extern ExtU_rtwdemo_abstime_T rtwdemo_abstime_U;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY_rtwdemo_abstime_T rtwdemo_abstime_Y;

/* Model entry point functions */
extern void rtwdemo_abstime_initialize(void);
extern void rtwdemo_abstime_step0(void);
extern void rtwdemo_abstime_step1(void);
extern void rtwdemo_abstime_step2(void);
extern void rtwdemo_abstime_terminate(void);

/* Real-time Model object */
extern RT_MODEL_rtwdemo_abstime_T *const rtwdemo_abstime_M;

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model.  The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model.  For example,
 *
 * hilite_system('<S3>')    - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'rtwdemo_abstime'
 * '<S1>'   : 'rtwdemo_abstime/SS1'
 * '<S2>'   : 'rtwdemo_abstime/SS2'
 * '<S3>'   : 'rtwdemo_abstime/SS3'
 */
#endif                                /* RTW_HEADER_rtwdemo_abstime_h_ */
```

Four 32-bit unsigned integers, `clockTick1` , `clockTickH1` , `clockTick2` , and `clockTickH2` are counters for storing the elapsed time of subsystems `SS2` and `SS3`.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.

2. Build the model.

```
### Starting build procedure for: rtwdemo_abstime
### Successful completion of build procedure for: rtwdemo_abstime

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
================================================================================
rtwdemo_abstime  Code generated and compiled.  Incremental checksum changed.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.449s
```

Review the Regenerated Code

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint16_T clockTick2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};

/* Block states (default storage) */
extern DW_rtwdemo_abstime_T rtwdemo_abstime_DW;

/* Zero-crossing (trigger) state */
extern PrevZCX_rtwdemo_abstime_T rtwdemo_abstime_PrevZCX;

/* External inputs (root inport signals with default storage) */
extern ExtU_rtwdemo_abstime_T rtwdemo_abstime_U;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY_rtwdemo_abstime_T rtwdemo_abstime_Y;

/* Model entry point functions */
extern void rtwdemo_abstime_initialize(void);
extern void rtwdemo_abstime_step0(void);
extern void rtwdemo_abstime_step1(void);
extern void rtwdemo_abstime_step2(void);
```

```
extern void rtwdemo_abstime_terminate(void);

/* Real-time Model object */
extern RT_MODEL_rtwdemo_abstime_T *const rtwdemo_abstime_M;

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model.  The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model.  For example,
 *
 * hilite_system('<S3>')    - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'rtwdemo_abstime'
 * '<S1>'   : 'rtwdemo_abstime/SS1'
 * '<S2>'   : 'rtwdemo_abstime/SS2'
 * '<S3>'   : 'rtwdemo_abstime/SS3'
 */
#endif                                /* RTW_HEADER_rtwdemo_abstime_h_ */
```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for `SS2`
- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for `SS3`

Related Information

- “Model Configuration Parameters: Code Generation Optimization”
- “Timers in Asynchronous Tasks”
- “Time-Based Scheduling and Code Generation”

See Also

More About

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Control Memory Allocation for Time Counters” on page 76-10
- “Access Timers Programmatically”
- “Generate Code for an Elapsed Time Counter”
- “Absolute Time Limitations”

Optimize Generated Code Using Boolean Data for Logical Signals

Optimize generated code by storing logical signals as Boolean data. When you select the model configuration parameter **Implement logic signals as Boolean data (vs. double)**, blocks that generate logic signals output Boolean signals.

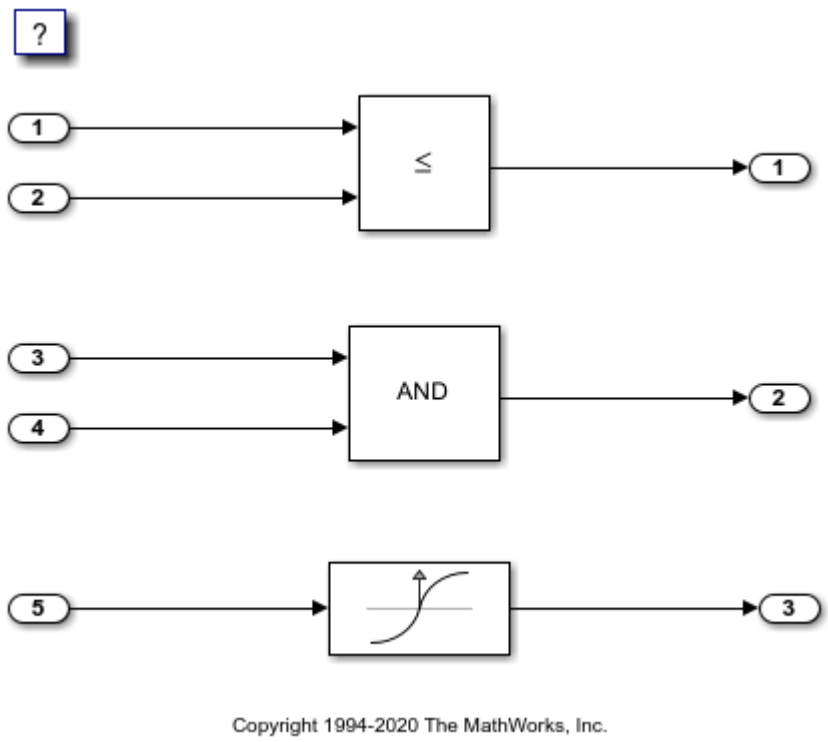
The optimization:

- Reduces the ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model `LogicalAsBoolean`. The outputs of the `Relational Operator`, `Logical Operator` and `HitCrossing` blocks are `double`, even though they represent logical data.

```
model = 'LogicalAsBoolean';
open_system(model);
```



Generate Code

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: LogicalAsBoolean
### Successful completion of build procedure for: LogicalAsBoolean
```

```
Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
==========================================================================================
LogicalAsBoolean  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.959s
```

View the generated code without the optimization. These lines of code are in `LogicalAsBoolean.h`.

```
hfile = fullfile('LogicalAsBoolean_grt_rtw',...
    'LogicalAsBoolean.h');
rtwdemodbtype(hfile,'/* External outputs','/* Parameters (default storage) */',1,0);
```

```
/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1;                          /* '<Root>/Out1' */
  real_T Out2;                          /* '<Root>/Out2' */
  real_T Out3;                          /* '<Root>/Out3' */
} ExtY_LogicalAsBoolean_T;
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. Select the **Implement logic signals as Boolean data (vs. double)** parameter.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'BooleanDataType','on');
```

Generate Code with Optimization

The generated code stores the logical signal output as Boolean data.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: LogicalAsBoolean
### Successful completion of build procedure for: LogicalAsBoolean

Build Summary

Top model targets built:

Model             Action                        Rebuild Reason
==================================================================================
LogicalAsBoolean  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.968s
```

View the generated code with the optimization. These lines of code are in `LogicalAsBoolean.h`.

```
rtwdemodbtype(hfile,'/* External outputs','/* Parameters (default storage) */',1,0);
```

```
/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  boolean_T Out1;                      /* '<Root>/Out1' */
  boolean_T Out2;                      /* '<Root>/Out2' */
  boolean_T Out3;                      /* '<Root>/Out3' */
} ExtY_LogicalAsBoolean_T;
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
```

See Also

"Implement logic signals as Boolean data (vs. double)"

Related Examples

- "Design Techniques to Optimize Models for Efficient Code Generation" on page 76-7
- "Data Types Supported by Simulink"
- "Use Conditional Input Branch Execution" on page 76-62
- "Bitfields" on page 26-74

Reduce Memory Usage for Boolean and State Configuration Variables

1. Open the Model Configuration Parameters dialog box.
2. In the Model Configuration Parameters dialog box, select the **Optimization** pane.
3. Choose from these options:
   - **Use bitsets for storing state configuration** — Reduces the amount of memory that stores state configuration variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.
   - **Use bitsets for storing Boolean data** — Reduces the amount of memory that stores Boolean variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.

   Note You cannot use bitsets when you generate code for these cases:

   - An external mode simulation
   - A target that specifies an explicit structure alignment

See Also

Use bitsets for storing state configuration | Use bitsets for storing Boolean data

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Optimize Generated Code Using Boolean Data for Logical Signals” on page 76-89
- “Bitfields” on page 26-74

Customize Stack Space Allocation

Your application might be constrained by limited memory. Controlling the maximum allowable size for the stack is one way to modify whether data is defined as local or global in the generated code. You can limit the use of stack space by specifying a positive integer value for the Maximum stack size (bytes) parameter, on the **Optimization** pane of the Configuration parameter dialog box. Specifying the maximum allowable stack size provides control over the number of local and global variables in the generated code. Specifically, lowering the maximum stack size might generate more variables into global structures. The number of local and global variables help determine the required amount of stack space for execution of the generated code.

If your model contains a variable that is larger than 4096 bytes, the code generator implements it in global memory by default. You can increase the size of variables that the code generator places in local memory by changing the value of the TLC variable `MaxStackVariableSize`. You can change this value by typing the following command in MATLAB Command Window:

```
set_param(modelName,'TLCOptions','-aMaxStackVariableSize=N')
```

The default setting for Maximum stack size (bytes) is `Inherit from target`. In this case, the value of the maximum stack size is the smaller value of the following: the default value set by the code generator (200,000 bytes) or the value of the TLC variable `MaxStackSize` found in the system target file (`ert.tlc`).

To specify a smaller stack size for your application, select the `Specify a value` option of the **Maximum stack size (bytes)** parameter and enter a positive integer value. To specify a smaller stack size at the command line, use:

```
set_param(model_name, 'MaxStackSize', 65000);
```

Note For overall executable stack usage metrics, you might want to do a target-specific measurement, such as using runtime (empirical) analysis or static (code path) analysis with object code.

It is recommended that you use the **Maximum stack size (bytes)** parameter to control stack space allocation instead of modifying the TLC variable, `MaxStackSize`, in the system target file. However, a target author might want to set the TLC variable, `MaxStackSize`, for a target. To set `MaxStackSize`, use `assign` statements in the system target file (`ert.tlc`), as in the following example.

```
%assign MaxStackSize = 4096
```

Write your `%assign` statements in the `Configure RTW code generation settings` section of the system target file. The `%assign` statement is described in “Target Language Compiler”.

See Also

Maximum stack size (bytes)

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Enable and Reuse Local Block Outputs in Generated Code” on page 76-102

- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35
- “Inline Numeric Values of Block Parameters” on page 76-41

Optimize Generated Code Using memset Function

This example shows how to optimize the generated code by using the `memset` function to clear the internal storage. When you select the model configuration parameter **Use memset to initialize floats and doubles to 0.0**, the `memset` function clears internal storage, to the integer bit pattern `0` (that is, all bits are off).

If your compiler and target CPU both represent floating-point zero with the integer bit pattern `0`, consider setting this parameter to gain execution and ROM efficiency.

NOTE: The command-line values are the reverse of the settings values. 'on' in the command line corresponds to clearing the setting. `'off'` in the command line corresponds to selecting the setting.

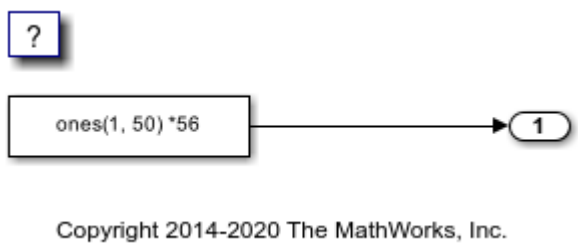
This optimization:

- Reduces ROM consumption.
- Improves execution speed.

Example Model

Consider the model matlab:rtwdemo_memset.

```
model = 'rtwdemo_memset';
open_system(model);
```



Generate Code

The code generator uses a loop to initialize the `Constant` block values.

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_memset
### Successful completion of build procedure for: rtwdemo_memset

Build Summary

Top model targets built:
```

```
Model           Action                        Rebuild Reason
==========================================================================================
rtwdemo_memset  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.456s
```

View the generated code without the optimization. These lines of code are in `rtwdemo_memset.c`.

```
cfile = fullfile(cgDir,'rtwdemo_memset_grt_rtw','rtwdemo_memset.c');
rtwdemodbtype(cfile,'/* Model initialize function */',...
    '/* Model terminate function */',1,0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

  /* external outputs */
  {
    int32_T i;
    for (i = 0; i < 50; i++) {
      rtwdemo_memset_Y.Out1[i] = 0.0;
    }
  }

  {
    int32_T i;

    /* ConstCode for Outport: '<Root>/Out1' */
    for (i = 0; i < 50; i++) {
      rtwdemo_memset_Y.Out1[i] = 56.0;
    }

    /* End of ConstCode for Outport: '<Root>/Out1' */
  }
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. In the Configuration Parameter dialog box select the **Use memset to initialize floats and doubles to 0.0** parameter. Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'InitFltsAndDblsToZero','off');
```

```
% # Open the Configuration Parameters dialog box.
```

Generate Code with Optimization

The code generator uses the `memset` function to initialize the `Constant` block values.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_memset
### Successful completion of build procedure for: rtwdemo_memset

Build Summary

Top model targets built:

Model           Action                        Rebuild Reason
================================================================================
rtwdemo_memset  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.593s
```

View the generated code with the optimization. These lines of code are in `rtwdemo_memset.c`.

```
rtwdemodbtype(cfile,'/* Model initialize function */',...
    '/* Model terminate function */',1,0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

  /* external outputs */
  (void)memset(&rtwdemo_memset_Y, 0, sizeof(ExtY_rtwdemo_memset_T));

  {
    int32_T i;

    /* ConstCode for Outport: '<Root>/Out1' */
    for (i = 0; i < 50; i++) {
      rtwdemo_memset_Y.Out1[i] = 56.0;
    }

    /* End of ConstCode for Outport: '<Root>/Out1' */
  }
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Use memset to initialize floats and doubles to 0.0

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 76-48
- “Vector Operation Optimization” on page 76-99
- “Remove Zero-Initialization Code” on page 79-4

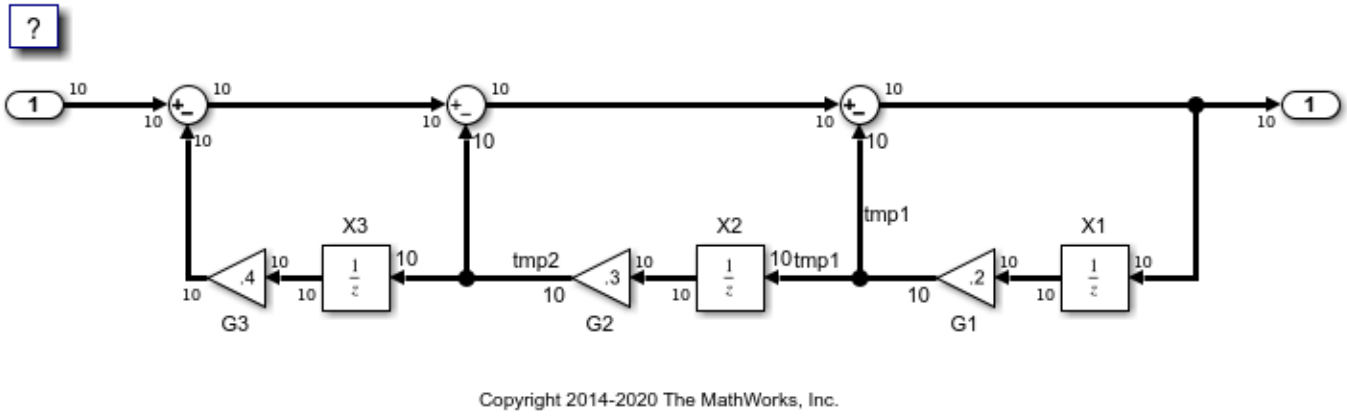
Vector Operation Optimization

This example shows how Simulink® Coder™ optimizes generated code by setting block output that generates vectors to scalars, for blocks such as the Mux, Sum, Gain, and Bus. This optimization reduces stack memory by replacing temporary local arrays with local variables.

Example Model

In the model, rtwdemo_VectorOptimization, the output of Gain blocks `G1` and `G2` are the vector signals `tmp1` and `tmp2`. These vectors have a width of 10.

```
model = 'rtwdemo_VectorOptimization';
open_system(model);
set_param(model, 'SimulationCommand', 'update')
```



Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_VectorOptimization
### Successful completion of build procedure for: rtwdemo_VectorOptimization

Build Summary

Top model targets built:

Model                       Action                        Rebuild Reason
========================================================================================
rtwdemo_VectorOptimization  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.217s
```

The optimized code is in `rtwdemo_VectorOptimization.c`. The signals `tmp1` and `tmp2` are the local variables `rtb_tmp1` and `rtb_tmp2`.

```
cfile = fullfile(cgDir,'rtwdemo_VectorOptimization_grt_rtw',...
    'rtwdemo_VectorOptimization.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_VectorOptimization_step(void)
{
  real_T rtb_Sum3_0;
  real_T rtb_tmp1_0;
  real_T rtb_tmp2_0;
  int32_T i;
  for (i = 0; i < 10; i++) {
    /* Gain: '<Root>/G2' incorporates:
     *  UnitDelay: '<Root>/X2'
     */
    rtb_tmp2_0 = 0.3 * rtwdemo_VectorOptimization_DW.X2_DSTATE[i];

    /* Gain: '<Root>/G1' incorporates:
     *  UnitDelay: '<Root>/X1'
     */
    rtb_tmp1_0 = 0.2 * rtwdemo_VectorOptimization_DW.X1_DSTATE[i];

    /* Sum: '<Root>/Sum3' incorporates:
     *  Gain: '<Root>/G3'
     *  Inport: '<Root>/In2'
     *  Sum: '<Root>/Sum1'
     *  Sum: '<Root>/Sum2'
     *  UnitDelay: '<Root>/X3'
     */
    rtb_Sum3_0 = ((rtwdemo_VectorOptimization_U.In2[i] - 0.4 *
                   rtwdemo_VectorOptimization_DW.X3_DSTATE[i]) - rtb_tmp2_0) -
      rtb_tmp1_0;

    /* Outport: '<Root>/Out2' */
    rtwdemo_VectorOptimization_Y.Out2[i] = rtb_Sum3_0;

    /* Update for UnitDelay: '<Root>/X3' */
    rtwdemo_VectorOptimization_DW.X3_DSTATE[i] = rtb_tmp2_0;

    /* Update for UnitDelay: '<Root>/X2' */
    rtwdemo_VectorOptimization_DW.X2_DSTATE[i] = rtb_tmp1_0;

    /* Update for UnitDelay: '<Root>/X1' */
    rtwdemo_VectorOptimization_DW.X1_DSTATE[i] = rtb_Sum3_0;
  }
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 76-48

Enable and Reuse Local Block Outputs in Generated Code

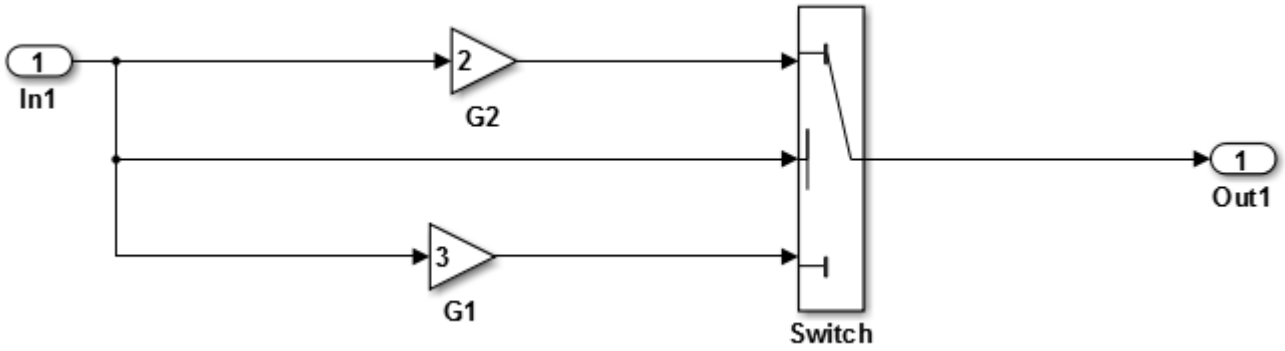
In this section...

"Example Model" on page 76-102
"Generate Code Without Optimization" on page 76-102
"Enable Local Block Outputs and Generate Code" on page 76-103
"Reuse Local Block Outputs and Generate Code" on page 76-104

This example shows how to specify block output as local variables. The code generator can potentially reuse these local variables in the generated code. Declaring block output as local variables conserves ROM consumption. Reusing local variables conserves RAM consumption, reduces data copies, and increases execution speed.

Example Model

1. Use Inport, Outport, Gain, and Switch blocks to create the following model. In this example, the model is named `local_variable_ex`.



2. For `G2`, open the Gain Block Parameters dialog box. Enter a value of `2`.
3. For `G1`, enter a value of `3`.
4. For the Switch block, open the Block Parameters dialog box. For the **Criteria for passing first input** parameter, select `u2>=Threshold`.

Generate Code Without Optimization

1. Open the Model Configuration Parameters dialog box. Select the **Solver** pane. For the **Type** parameter, select `Fixed-step`.
2. Clear the **Configuration Parameters** > **Signal storage reuse** parameter.
3. Select the **Code Generation > Report** pane and select **Create code generation report**.
4. Select the **Code Generation** pane. Select **Generate code only**, and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report appears.
5. In the code generation report, select the `local_variable_ex.c` section and view the model step function. The Gain block outputs are the global variables `local_variable_ex_B.G2` and `local_variable_ex_B.G1`.

```
/* Model step function */
void local_variable_ex_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    local_variable_ex_B.G2 = 2.0 * local_variable_ex_U.In1;

    /* Outport: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = local_variable_ex_B.G2;
  } else {
    /* Gain: '<Root>/G1' */
    local_variable_ex_B.G1 = 3.0 * local_variable_ex_U.In1;

    /* Outport: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = local_variable_ex_B.G1;
  }

  /* End of Switch: '<Root>/Switch' */
```

Enable Local Block Outputs and Generate Code

1. Select the **Configuration Parameters > Signal Storage Reuse** parameter. The **Signal Storage Reuse** enables the following optimization parameters:
   - **Enable local block outputs**
   - **Reuse local block outputs**
   - **Eliminate superfluous local variables (expression folding)**
2. Clear **Reuse local block outputs** and **Eliminate superfluous local variables (expression folding)**.
3. Generate code and view the model step function. There are three local variables in the model step function because you selected the optimization parameter **Enable Local Block Outputs**. The local variables `rtb_G2` and `rtb_G1` hold the outputs of the Gain blocks. The local variable `rtb_Switch` holds the output of the Switch block.

```
/* Model step function */
void local_variable_ex_step(void)
{
  real_T rtb_Switch;
  real_T rtb_G2;
  real_T rtb_G1;

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    rtb_G2 = 2.0 * local_variable_ex_U.In1;
    rtb_Switch = rtb_G2;
  } else {
    /* Gain: '<Root>/G1' */
    rtb_G1 = 3.0 * local_variable_ex_U.In1;
    rtb_Switch = rtb_G1;
```

```
  }

  /* End of Switch: '<Root>/Switch' */

  /* Outport: '<Root>/Out1' */
  local_variable_ex_Y.Out1 = rtb_Switch;
```

Reuse Local Block Outputs and Generate Code

1. Select the **Configuration Parameters** > **Reuse local block outputs** parameter.
2. Generate code. In the `local_variable_ex.c` section, view the model step function. There is one local variable, `rtb_G2`, that the code generator uses three times.

```
/* Model step function */
void local_variable_ex_step(void)
{
  real_T rtb_G2;

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    rtb_G2 = 2.0 * local_variable_ex_U.In1;
  } else {
    /* Gain: '<Root>/G1' */
    rtb_G2 = 3.0 * local_variable_ex_U.In1;
  }

  /* End of Switch: '<Root>/Switch' */

  /* Outport: '<Root>/Out1' */
  local_variable_ex_Y.Out1 = rtb_G2;
```

The extra temporary variable `rtb_Switch` and the associated data copy is not in the generated code.

See Also

Enable local block outputs

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Customize Stack Space Allocation” on page 76-93
- “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49

Generate SIMD Code from Simulink Blocks

You can generate single instruction, multiple data (SIMD) code from certain Simulink blocks by using Intel SSE and, if you have Embedded Coder, Intel AVX technology. SIMD is a computing paradigm in which a single instruction processes multiple data. Many modern processors have SIMD instructions that, for example, perform several additions or multiplications at once. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms.

Blocks That Support SIMD Code Generation

When certain conditions are met, you can generate SIMD code by using Intel SSE or Intel AVX technology. This table lists blocks that support SIMD code generation. The table also details the conditions under which the support is available.

Block	Conditions
Add	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32`, or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`.
Subtract	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32`, or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`.
Sum of Elements	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32`, or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`. • The **Optimize reductions** configuration parameter is set to `on`.

Block	Conditions
Product	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int16` or `int32`. • For AVX512F, the input signal has a data type of `single` or `double`. • Set **Multiplication** parameter to `Element-wise(.*)`
Product of Elements	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int16` or `int32`. • For AVX512F, the input signal has a data type of `single` or `double`. • Set **Multiplication** parameter to `Element-wise(.*)` • Set the **Optimize reductions** configuration parameter to `on`.
Gain	• For AVX, and SSE, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int16`, or `int32`. • For Intel AVX512F, the input signal has a data type of `single` or `double`. • Set **Multiplication** parameter to `Element-wise(.*)`
Divide	The input signal has a data type of `single` or `double`.
Sqrt	The input signal has a data type of `single` or `double`.
Ceil	• For AVX2, AVX, SSE4.1, SSE2, and SSE, the input signal has a data type of `single` or `double`. • AVX512F is not supported.
Floor	• For AVX2, AVX, SSE4.1, SSE2, and SSE, the input signal has a data type of `single` or `double`. • AVX512F is not supported.

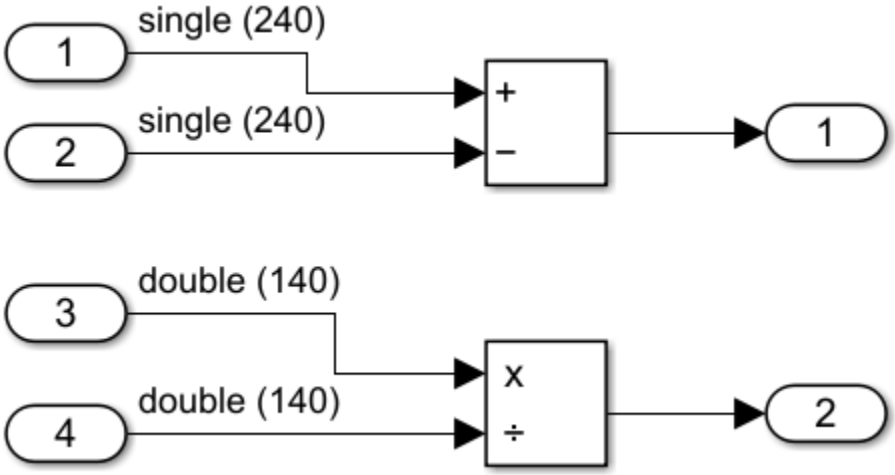
Block	Conditions
MinMax	• The input signal has a data type of `single` or `double`. • The value of the **Support: non-finite numbers** configuration parameter is set to `off`.
MinMax of Elements	• The input signal has a data type of `single` or `double`. • The value of the **Support: non-finite numbers** configuration parameter is set to `off`. • The **Optimize reductions** configuration parameter is set to `on`.
MATLAB Function	MATLAB code meets the conditions specified in this topic: “Generate SIMD Code for MATLAB Functions” on page 97-27.
For Each Subsystem	• The For Each Subsystem block contains a block listed in this table that meets the specified conditions. • The value of the **Partition Dimension** block parameter must be above the value of the **Loop unrolling threshold** configuration parameter.
Bitwise Operator	• The value of the **Operator** block parameter must be `AND`, `OR`, or `XOR`. • For SSE2, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`. • For AVX2, and AVX512F, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`.
Shift Arithmetic	• The input signal has a data type of `int32`.

If you have DSP System Toolbox, you can also generate SIMD code from certain DSP System Toolbox blocks. For more information, see “Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation” (DSP System Toolbox).

If you have the Embedded Coder Support Package for ARM Cortex®-A Processors, you can generate SIMD code optimized for ARM Cortex A processors by setting the Code replacement libraries parameter to `GCC ARM Cortex-A`. You will also need to set the **Device vendor** parameter to `ARM Compatible` and the **Device type** parameter to `ARM Cortex-A`.

Generate SIMD Code Compared to Plain C Code

For this example, create a simple model `simdDemo` that has a Subtract block and a Divide block. The Subtract block has an input signal that has a dimension of 240 and an input data type of `single`. The Divide block has an input signal that has a dimension of 140 and an input data type of `double`.



The plain generated C code for this model is:

```
void simdDemo_step(void)
{
  int32_T i;
  for (i = 0; i < 240; i++) {
    simdDemo_Y.Out1[i] = simdDemo_U.In1[i] - simdDemo_U.In2[i];
  }

  for (i = 0; i < 140; i++) {
    simdDemo_Y.Out2[i] = simdDemo_U.In3[i] / simdDemo_U.In4[i];
  }
}
```

In the plain (non-SIMD) C code, each loop iteration produces one result.

To generate SIMD code:

1. Open the Simulink Coder app or the Embedded Coder app.
2. Click **Settings** > **Hardware Implementation**.
3. Set the **Device vendor** parameter to `Intel` or `AMD`.
4. Set the **Device type** parameter to `x86-64(Windows 64)` or `x86-64(Linux 64)`.
5. On the **Optimization** pane, for the **Leverage target hardware instruction set extensions** parameter, select the instruction set extension that your processor supports. For example, select `SSE2`. If you use Embedded Coder, you can also select from the instruction sets `SSE`, `SSE4.1`, `AVX`, `AVX2`, `FMA`, and `AVX512F`. For more information, see https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.
6. Optionally, select the **Optimize reductions** parameter to generate SIMD code for reduction operations.
7. Generate code from the model.

```
void simdDemo_step(void)
{
  int32_T i;
  for (i = 0; i <= 236; i += 4) {
    _mm_storeu_ps(&simdDemo_Y.Out1[i], _mm_sub_ps(_mm_loadu_ps(&simdDemo_U.In1[i]),
      _mm_loadu_ps(&simdDemo_U.In2[i])));
```

```
  }

  for (i = 0; i <= 138; i += 2) {
    _mm_storeu_pd(&simdDemo_Y.Out2[i], _mm_div_pd(_mm_loadu_pd(&simdDemo_U.In3[i]),
      _mm_loadu_pd(&simdDemo_U.In4[i])));
  }
}
```

This code is for the `SSE2` instruction set extension. The SIMD instructions are the intrinsic functions that start with the identifier `_mm`. These functions process multiple data in a single iteration of the loop because the loop increments by four for single data types and by two for double data types. For models that process more data and are computationally more intensive than this one, the presence of SIMD instructions can significantly speed up the code execution time.

For a list of a Intel intrinsic functions for supported Simulink blocks, see https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Limitations

The generated code is not optimized through SIMD if:

- The code in a MATLAB Function block contains scalar data types outside the body of loops. For instance, if `a`,`b`, and `c` are scalars, the generated code does not optimize an operation such as `c=a+b`.
- The code in a MATLAB Function block contains indirectly indexed arrays or matrices. For instance if `A`,`B`,`C`, and `D` are vectors, the generated code is not vectorized for an operation such as `D(A)=C(A)+B(A)`.
- The blocks within a reusable subsystem might not be optimized.
- If the code in a MATLAB Function block contains parallel for-Loops (`parfor`), the `parfor` loop is not optimized with SIMD code, but loops within the body of the `parfor` loop can be optimized with SIMD code.
- Polyspace does not support analysis of generated code that includes SIMD instructions. Disable SIMD code generation by setting the **Leverage target hardware instruction set extensions** parameter to `None`.
- Reduction operations that operate on floating-point values when support for nonfinite numbers is enabled. To generate SIMD code for reduction operations that operate on floating-point values, set the configuration parameter **Support: non-finite numbers** to `Off`.

See Also

More About

- “Generate SIMD Code for MATLAB Functions” on page 97-27
- “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” (DSP System Toolbox)
- “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” (DSP System Toolbox)

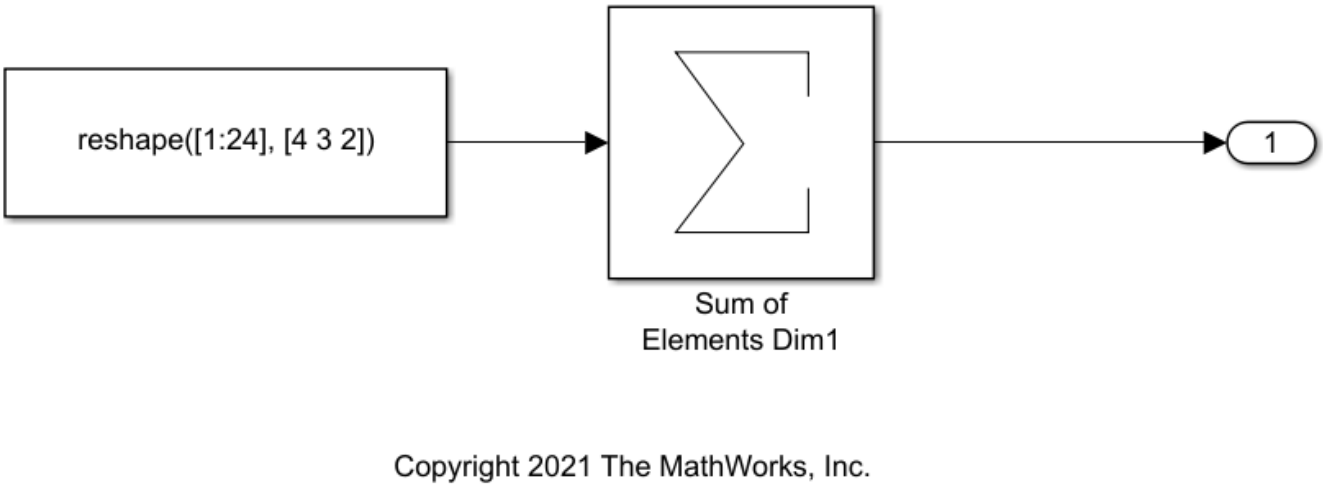
Optimize Code for Reduction Operations by Using SIMD

This example shows how to generate optimized single instruction, multiple data (SIMD) code that performs reduction operations. A reduction is an operation that reduces a set of elements, such as an array, to a single value using an associative binary operator. For example, calculating the sum of the elements in an array is a reduction operation that uses the addition operator. For processors that support SIMD instructions, you can optimize reduction operations by generating parallel code that uses the SIMD instructions for the operation.

Example Model

Open the example model `ex_rowmajor_algorithm`. The model calculates the sum of the elements in a 4 by 3 by 2 array.

```
model = 'ex_rowmajor_algorithm';
open_system(model)
```



Generate SIMD Code Without Reduction Optimization

Open the configuration parameters for the model. On the **Modeling** tab, click **Settings**. On the **Code Generation** pane, set **System target file** to `grt.tlc`. On the **Optimization** pane, set the **Leverage target hardware instruction set extensions** to `SSE2`. Alternatively, use the command-line.

```
set_param(model, 'SystemTargetFile', 'grt.tlc');
set_param(model, 'SaveOutput', 'off');
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'InstructionSetExtensions', 'SSE2');
```

De-select the parameter **Optimize reductions**.

```
set_param(model, 'OptimizeReductions', 'off');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: ex_rowmajor_algorithm
### Successful completion of build procedure for: ex_rowmajor_algorithm

Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
ex_rowmajor_algorithm  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.88s
```

Inspect the generated `ex_rowmajor_algorithm_step` function.

```
file = fullfile('ex_rowmajor_algorithm_grt_rtw','ex_rowmajor_algorithm.c');
rtwdemodbtype(file,'/* Model step function */','/* End of Sum',1,1);
```

```
/* Model step function */
void ex_rowmajor_algorithm_step(void)
{
  int32_T i;

  /* Sum: '<Root>/Sum of Elements Dim1' incorporates:
   *  Constant: '<Root>/Constant1'
   */
  ex_rowmajor_algorithm_Y.Out2 = -0.0F;
  for (i = 0; i < 24; i++) {
    ex_rowmajor_algorithm_Y.Out2 +=
      ex_rowmajor_algorithm_ConstP.Constant1_Value[i];
  }
```

The generated code contains a non-vectorized loop that adds each array element individually.

Generate SIMD Code With Reduction Optimization

Open the configuration parameters for the model. On the **Modeling** tab, click **Settings**. On the **Optimization** pane, set the **Leverage target hardware instruction set extensions** to SSE2. Alternatively, use the command-line.

```
set_param(model, 'InstructionSetExtensions', 'SSE2');
```

Select the parameter **Optimize reductions**. If you want to optimize reduction operations that operate on floating-point values, set **Support: Non-finite numbers** to `Off`. Reduction optimization does not support operations on floating-point values when non-finite numbers are supported.

```
set_param(model, 'OptimizeReductions', 'on');
set_param(model, 'SupportNonFinite', 'off');
set_param(model, 'MatFileLogging', 'off');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: ex_rowmajor_algorithm
### Successful completion of build procedure for: ex_rowmajor_algorithm
```

```
Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
ex_rowmajor_algorithm  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 9.3657s
```

Inspect the generated `ex_rowmajor_algorithm_step` function.

```
file = fullfile('ex_rowmajor_algorithm_grt_rtw','ex_rowmajor_algorithm.c');
rtwdemodbtype(file,'/* Model step function */','/* Matfile',1,1);
```

```
/* Model step function */
void ex_rowmajor_algorithm_step(void)
{
  __m128 tmp;
  int32_T i;
  real32_T tmp_0[4];
  tmp = _mm_set1_ps(0.0F);
  for (i = 0; i <= 20; i += 4) {
    /* Sum: '<Root>/Sum of Elements Dim1' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    tmp = _mm_add_ps(tmp, _mm_loadu_ps
                     (&ex_rowmajor_algorithm_ConstP.Constant1_Value[i]));
  }

  _mm_storeu_ps(&tmp_0[0], tmp);

  /* Outport: '<Root>/Out2' */
  ex_rowmajor_algorithm_Y.Out2 = (tmp_0[0] + tmp_0[1]) + (tmp_0[2] + tmp_0[3]);
}

/* Model initialize function */
void ex_rowmajor_algorithm_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(ex_rowmajor_algorithm_M, (NULL));

  /* external outputs */
  ex_rowmajor_algorithm_Y.Out2 = 0.0F;
}

/* Model terminate function */
void ex_rowmajor_algorithm_terminate(void)
{
  /* (no terminate code required) */
}
```

The vectorized loop uses the SSE2 function `mm_add_ps`, which adds four elements of the array to the sum in one step. This optimization improves the execution speed of the generated code.

Open the code generation report. On the **C Code** tab, click **Open Report**.

Additional Information

Code Generation Advisor	Not run
Optimize reductions	Reductions are optimized using SSE2 Instruction Set Extensions

On the report summary, in the **Additional Information** section, the report indicates that reduction operations are optimized using the SSE2 instruction set.

See Also

More About

- “Row-Major Algorithms for Row-Major Array Layout”

Configuration in Embedded Coder

Set Hardware Implementation Parameters

Specification of target hardware device characteristics (such as word sizes for `char`, `short`, `int`, and `long` data types, or desired rounding behaviors in integer operations) for generated code can be critical in embedded systems development. The **Hardware Implementation** category of parameters in a configuration set provides a way to control such characteristics in simulation and code generation.

By configuring the **Hardware Implementation** parameters of the active configuration set for a model to match the behaviors of your compiler and hardware, you can generate more efficient code. For example, if you specify the **Byte ordering** parameter, you can avoid generation of extra code that tests the byte ordering of the target CPU.

Before generating and deploying code, get familiar with the **Hardware Implementation** pane of the Configuration Parameters dialog box. By default, target hardware microprocessor device details are hidden. To view the details, click the **Device details** arrow. See “Hardware Implementation Pane” and “Configure Run-Time Environment Options” for more information.

Data Copy Reduction in Embedded Coder

- “Optimize Global Variable Usage” on page 78-2
- “Reuse Global Block Outputs in the Generated Code” on page 78-12
- “Virtualized Output Ports Optimization” on page 78-15
- “Specify Buffer Reuse for Signals in a Path” on page 78-17
- “Specify Buffer Reuse for MATLAB Function Blocks in a Path” on page 78-25
- “Remove Data Copies by Reordering Block Operations in the Generated Code” on page 78-27
- “Data Copy Reduction for Data Store Read and Data Store Write Blocks” on page 78-32
- “Reduce Data Copies for Bus Assignment Blocks in Generated Code” on page 78-36
- “Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse” on page 78-39

Optimize Global Variable Usage

In this section...

"Use Global to Hold Temporary Results" on page 78-2
"Minimize Global Data Access" on page 78-6

To tune your application and choose tradeoffs for execution speed and memory usage, you can choose a global variable reference optimization for the generated code.

On the Configuration Parameters dialog box, in the **Optimize global data access** drop-down list, three parameter options control global variable usage optimizations.

- `None`. Use default optimizations. This choice works well for most models. The code generator balances the use of local and global variables. It generates code which balances RAM and ROM consumption and execution speed.
- `Use global to hold temporary results`. Reusing global variables improves code efficiency and readability. This optimization reuses global variables, which results in the code generator defining fewer variables. It reduces RAM and ROM consumption and data copies.
- `Minimize global data access`. Using local variables to cache global data reduces ROM consumption by reducing code size. This optimization improves execution speed because the code uses fewer instructions for local variable references than for global variable references.

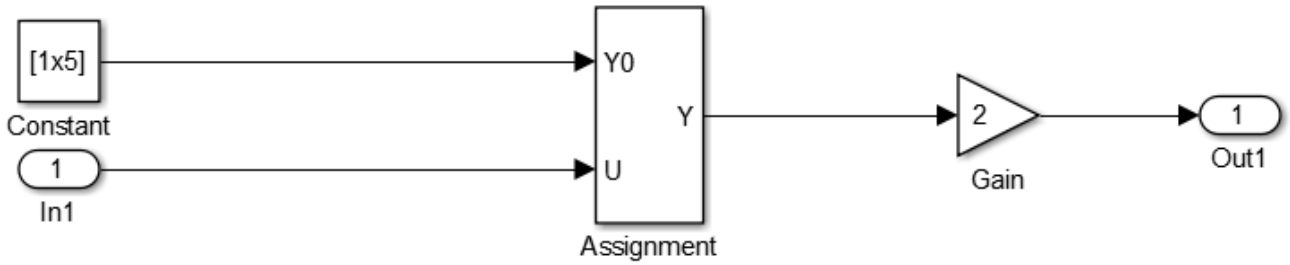
  Minimizing the use of global variables by using local variables interacts with stack usage control. For example, stack size can determine the number of local and global variables that the code generator can allocate in the generated code. For more information, see "Customize Stack Space Allocation".

Use Global to Hold Temporary Results

The code generator uses global and local variables when you select `None` versus when you select `Use global to hold temporary results`.

Example Model

In the model `UseGlobalsForTemporaryResults`, an Assignment block assigns values coming from the Inport and Constant blocks to an output signal. The output signal feeds into a Gain block.




```
model = 'UseGlobalsForTemporaryResults';
load_system('UseGlobalsForTemporaryResults')
```

Generate Code Without Optimization

1. In the Configuration Parameters dialog box, verify that the **Signal storage reuse** parameter is selected.
2. In the Configuration Parameters dialog box, for the **Optimize global data access** parameter, select `None` or enter the following command in the MATLAB® Command Window:

```
set_param('UseGlobalsForTemporaryResults','GlobalVariableUsage','None');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: UseGlobalsForTemporaryResults
### Successful completion of build procedure for: UseGlobalsForTemporaryResults

Build Summary

Top model targets built:

Model                          Action                        Rebuild Reason
========================================================================================
UseGlobalsForTemporaryResults  Code generated and compiled.  Code generation information file doe

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.194s
```

View the generated code without the optimization. Here is a portion of `UseGlobalsForTemporaryResults.c`.

```
cfile = fullfile(pwd,'UseGlobalsForTemporaryResults_ert_rtw',...
    'UseGlobalsForTemporaryResults.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void UseGlobalsForTemporaryResults_step(void)
{
  real_T rtb_Assignment[5];
  int32_T i;

  /* SignalConversion generated from: '<Root>/Assignment' incorporates:
   *  Constant: '<Root>/Constant'
   */
  for (i = 0; i < 5; i++) {
    rtb_Assignment[i] = rtCP_Constant_Value[i];
  }

  /* End of SignalConversion generated from: '<Root>/Assignment' */

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtb_Assignment[1] = rtU.In1;

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   */
  for (i = 0; i < 5; i++) {
```

```
    rtY.Out1[i] = 2.0 * rtb_Assignment[i];
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The code assigns values to the local vector `rtb_Assignment`. The last statement copies the values in the local vector `rtb_Assignment` to the global vector `rtY.Out1`. Fewer global variable references result in improved execution speed. The code uses more instructions for global variable references than for local variable references.

In the Static Code Metrics Report, examine the Global Variables section.

1. In the Code Generation Report window, select **Static Code Metrics Report**.
2. Scroll down to the `Global Variables` section.
3. Select the **[+]** sign before each variable to expand it.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	1	1
Out1	40	1	1
[-] rtU	8	1	1
In1	8	1	1
[-] rtM_	4	0*	0*
errorStatus	4	0	0
Total	**52**	**2**	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 2.

Generate Code with Optimization

In the Configuration Parameters dialog box, for the **Optimize global data access** parameter, select `Use global to hold temporary results`, or enter the following command in the MATLAB Command Window:

```
set_param('UseGlobalsForTemporaryResults',...
    'GlobalVariableUsage','Use global to hold temporary results');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: UseGlobalsForTemporaryResults
### Successful completion of build procedure for: UseGlobalsForTemporaryResults

Build Summary

Top model targets built:

Model                          Action                        Rebuild Reason
```

```
============================================================================================
UseGlobalsForTemporaryResults  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.912s
```

View the generated code with the optimization. Here is a portion of `UseGlobalsForTemporaryResults.c`.

```
cfile = fullfile(pwd,'UseGlobalsForTemporaryResults_ert_rtw',...
    'UseGlobalsForTemporaryResults.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void UseGlobalsForTemporaryResults_step(void)
{
  int32_T i;

  /* SignalConversion generated from: '<Root>/Assignment' incorporates:
   *  Assignment: '<Root>/Assignment'
   *  Constant: '<Root>/Constant'
   */
  for (i = 0; i < 5; i++) {
    rtY.Out1[i] = rtCP_Constant_Value[i];
  }

  /* End of SignalConversion generated from: '<Root>/Assignment' */

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtY.Out1[1] = rtU.In1;

  /* Outport: '<Root>/Out1' incorporates:
   *  Assignment: '<Root>/Assignment'
   *  Gain: '<Root>/Gain'
   */
  for (i = 0; i < 5; i++) {
    rtY.Out1[i] *= 2.0;
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The code assigns values to the global vector `rtY.Out1` without using a local variable. This assignment improves ROM and RAM consumption and reduces data copies. The code places the value in the destination variable for each assignment instead of copying the value at the end. In the Static Code Metrics Report, examine the Global Variables section.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	4	4
Out1	40	4	4
[-] rtU	8	1	1
In1	8	1	1
[-] rtM_	4	0*	0*
errorStatus	4	0	0
Total	**52**	**5**	

* The global variable is not directly used in any function.

As a result of using global variables to hold local results, the total number of reads and writes for global variables has increased from 2 to 5. This optimization reduces data copies by reusing global variables.

Close the code generation report.

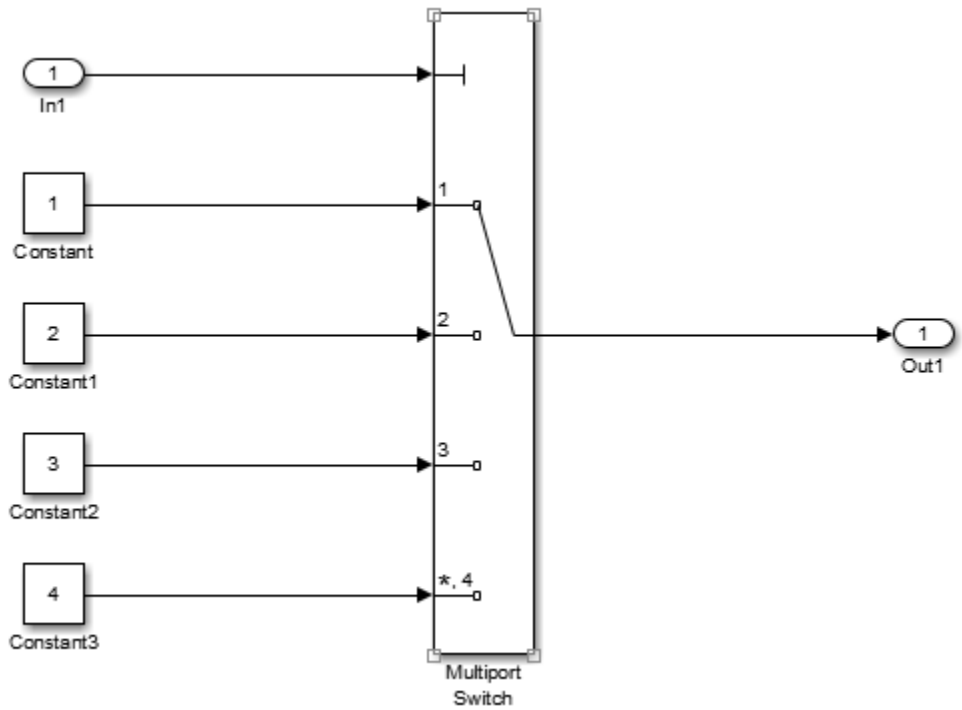
Minimize Global Data Access

Generate optimized code that reads from and writes to global variables less frequently.

Example Model

In the model `MinimizeGlobalDataAccess`, five signals feed into a Multiport Switch block.

```
model = 'MinimizeGlobalDataAccess';
load_system('MinimizeGlobalDataAccess.slx')
```



Generate Code Without Optimization

1. In the Configuration Parameters dialog box, verify that the **Signal storage reuse** parameter is selected.
2. In the Configuration Parameters dialog box, for the **Optimize global data access** parameter, select `None` or enter the following command in the MATLAB® Command Window:

```
set_param('MinimizeGlobalDataAccess','GlobalVariableUsage','None');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: MinimizeGlobalDataAccess
### Successful completion of build procedure for: MinimizeGlobalDataAccess

Build Summary

Top model targets built:

Model                      Action                        Rebuild Reason
========================================================================================
MinimizeGlobalDataAccess  Code generated and compiled.  Code generation information file does no

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.011s
```

View the generated code without the optimization. Here is a portion of `MinimizeGlobalDataAccess.c`.

```
cfile = fullfile(pwd,'MinimizeGlobalDataAccess_ert_rtw',...
    'MinimizeGlobalDataAccess.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void MinimizeGlobalDataAccess_step(void)
{
  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  switch ((int32_T)rtU.In1) {
   case 1:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant'
     */
    rtY.Out1 = 1.0;
    break;

   case 2:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    rtY.Out1 = 2.0;
    break;

   case 3:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant2'
     */
    rtY.Out1 = 3.0;
    break;

   default:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant3'
     */
    rtY.Out1 = 4.0;
    break;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */
}
```

In the Static Code Metrics Report, examine the `Global Variables` section.

1. In the Code Generation Report window, select **Static Code Metrics Report**.
2. Scroll down to the `Global Variables` section.
3. Select the **[+]** sign before each variable to expand it.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[−] rtU	8	1	1
In1	8	1	1
[−] rtY	8	4	4
Out1	8	4	4
[−] rtM	4	0*	0*
errorStatus	4	0	0
Total	**20**	**5**	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 5.

Enable Optimization and Generate Code

In the Configuration Parameters dialog box, for the **Optimize global data access** parameter, select `Minimize global data access` or enter the following command in the MATLAB Command Window:

```
set_param('MinimizeGlobalDataAccess',...
    'GlobalVariableUsage','Minimize global data access');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: MinimizeGlobalDataAccess
### Successful completion of build procedure for: MinimizeGlobalDataAccess

Build Summary

Top model targets built:

Model                     Action                        Rebuild Reason
========================================================================================
MinimizeGlobalDataAccess  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.092s
```

View the generated code with the optimization. Here is a portion of `MinimizeGlobalDataAccess.c`.

```
cfile = fullfile(pwd,'MinimizeGlobalDataAccess_ert_rtw',...
    'MinimizeGlobalDataAccess.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void MinimizeGlobalDataAccess_step(void)
{
  real_T tmp_Out1;
```

```
  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  switch ((int32_T)rtU.In1) {
   case 1:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant'
     */
    tmp_Out1 = 1.0;
    break;

   case 2:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    tmp_Out1 = 2.0;
    break;

   case 3:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant2'
     */
    tmp_Out1 = 3.0;
    break;

   default:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant3'
     */
    tmp_Out1 = 4.0;
    break;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */

  /* Outport: '<Root>/Out1' */
  rtY.Out1 = tmp_Out1;
}
```

In `MinimizeGlobalDataAccess.c`, the code assigns a constant value to the local variable `tmp_Out1` in each case statement. The last statement in the code copies the value of `tmp_Out1` to the global variable `rtY.Out1`. Fewer global variable references result in fewer instructions and improved execution speed.

In the **Static Code Metrics Report**, examine the `Global Variables` section. As a result of minimizing global data accesses, the total number of reads and writes for global variables has decreased from 5 to 2.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[–] rtU	8	1	1
In1	8	1	1
[–] rtY	8	1	1
Out1	8	1	1
[–] rtM	4	0*	0*
errorStatus	4	0	0
Total	**20**	**2**	

* The global variable is not directly used in any function.

Close the code generation report.

See Also

Optimize global data access

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35
- “Customize Stack Space Allocation” on page 76-93
- “Reuse Global Block Outputs in the Generated Code” on page 78-12

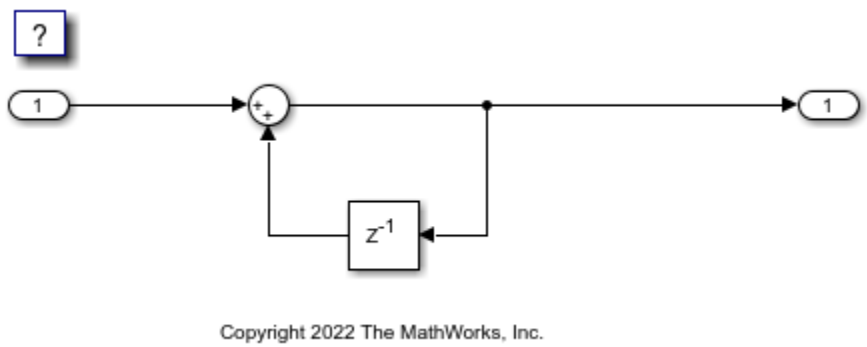
Reuse Global Block Outputs in the Generated Code

Reduce ROM and RAM consumption and data copies and increase execution speed of generated code. Configure the code generator to reuse global variables by selecting the model configuration parameter **Reuse global block outputs**.

Example Model

In the Command Window, open the `GlobalReuse` model.

```
model='GlobalReuse';
open_system('GlobalReuse')
```



Generate Code Without Optimization

1. On the Configuration Parameters dialog box, verify that **Signal storage reuse** is selected.
2. On the **Code Generation > Report** pane, verify that **Generate static code metrics** is selected.
3. Clear **Reuse global block outputs** and click **Apply**. Alternatively, you can programmatically set this parameter:

```
set_param(model,'GlobalBufferReuse', 'off');
```

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

```
slbuild(model);
```

```
### Starting build procedure for: GlobalReuse
### Successful completion of build procedure for: GlobalReuse

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================================
GlobalReuse  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.67s
```

View the generated code without the optimization. Here is a portion of `GlobalReuse.c`.

```
cfile = fullfile('GlobalReuse_ert_rtw','GlobalReuse.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void GlobalReuse_step(void)
{
  /* Sum: '<Root>/Sum' incorporates:
   *  Delay: '<Root>/Delay'
   *  Inport: '<Root>/In1'
   */
  rtDW.Delay_DSTATE += rtU.In1;

  /* Outport: '<Root>/Out1' incorporates:
   *  Delay: '<Root>/Delay'
   */
  rtY.Out1 = rtDW.Delay_DSTATE;
}
```

The generated code contains a data copy to the global variable `rtDW.Delay_DSTATE`. Open the Static Code Metrics Report. The total number of reads and writes for global variables is 8. The total size is 32 bytes.

Generate Code With Optimization

On the Configuration Parameters dialog box, select **Reuse global block outputs** and click **Apply**. Alternatively, you can programmatically set this parameter:

```
set_param(model,'GlobalBufferReuse', 'on');
```

Generate code.

```
slbuild(model);
```

```
### Starting build procedure for: GlobalReuse
### Successful completion of build procedure for: GlobalReuse

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================
GlobalReuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.364s
```

View the generated code with the optimization. Here is a portion of `GlobalReuse.c`.

```
cfile = fullfile('GlobalReuse_ert_rtw','GlobalReuse.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void GlobalReuse_step(void)
{
  /* Sum: '<Root>/Sum' incorporates:
   *  Delay: '<Root>/Delay'
```

```
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 += rtU.In1;
}
```

The code generator eliminates a data copy, reduces two statements to one statement and three global variables to two global variables.

Open the Static Code Metrics Report. For global variables, this optimization reduces the total number of reads and writes for global variables from 8 to 5 and the total size from 32 bytes to 24 bytes.

```
bdclose(model)
rtwdemoclean;
```

See Also

Reuse global block outputs

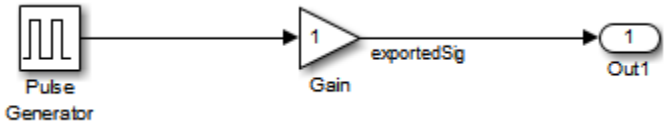
Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 76-35
- “Optimize Global Variable Usage” on page 78-2

Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. Clearing the **MAT-file logging** option and setting the TLC variable `FullRootOutputVector` to `0`, both defaults for Embedded Coder, eliminate code and data storage associated with root output ports.

Consider the model in the following block diagram. The signal `exportedSig` has `exportedGlobal` storage class.



In the default case, the output of the Gain block is written to the signal storage location, `exportedSig`. The code generator does not generate code or data for the `Out1` block, which has become a virtual block.

```
/* Gain Block: <Root>/Gain */
  exportedSig = rtb_PulseGen * VirtOutPortLogOFF_P.Gain_Gain;
```

In cases where you enable **MAT-file logging** or set `FullRootOutputVector = 1`, the generated code represents root output ports as members of an external outputs vector.

The following code fragment was generated with **MAT-file logging** enabled. The output port is represented as a member of the external outputs vector `VirtOutPortLogON_Y`. The Gain block output value is copied to `exportedSig` and to the external outputs vector.

```
/* Gain Block: <Root>/Gain */
  exportedSig = rtb_PulseGen * VirtOutPortLogON_P.Gain_Gain;

/* Outport Block: <Root>/Out1 */
  VirtOutPortLogON_Y.Out1 = exportedSig;
```

Data maintenance in the external outputs vector can be significant for smaller models that perform benchmarks.

You can force root output ports to be stored in the external outputs vector (regardless of the setting of **MAT-file logging**) by setting the TLC variable `FullRootOutputVector` to `1`. Add the statement

```
%assign FullRootOutputVector = 1
```

to the Embedded Coder system target file. Alternatively, you can enter the assignment from the MATLAB command line using the `set_param` command, the model parameter `TLCOptions`, and the TLC option `-a`. For more information, see “Specify TLC for Code Generation” and “Configure TLC”.

For more information on how to control signal storage in generated code, see “How Generated Code Stores Internal Signal, State, and Parameter Data” on page 36-49 and “C Code Generation Configuration for Model Interface Elements” on page 33-2.

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Specify Buffer Reuse for Signals in a Path” on page 78-17
- “Optimize Global Variable Usage” on page 78-2

Specify Buffer Reuse for Signals in a Path

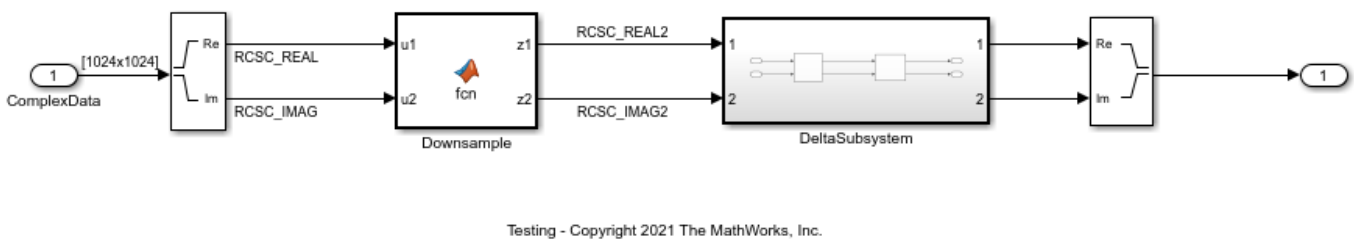
If your model has the optimal parameter settings for removing data copies, you might be able to remove additional data copies by using the `Reusable` storage class on the signals through the Code Mappings editor or `Simulink.Signal` data objects. This optimization decreases data copies and memory consumption and increases code execution speed. After studying the generated code and the Static Code Metrics Report and identifying areas where you think buffer reuse is possible, you specify the `Reusable` storage class on signal lines.

You can specify buffer reuse on signals that include a pair of root inport and outport signals. You can also specify buffer reuse on just a pair of root inport and outport signals. This optimization reduces ROM and RAM consumption because there are less global variables and data copies in the generated code. Code execution speed also increases.

Example Model

The model `ex_reusable_csc` contains the nonreusable subsystem `DeltaSubsystem` and the MATLAB Function block `Downsample`. `DeltaSubsystem` contains the MATLAB Function blocks `DeltaX` and `DeltaY`.

```
model ='ex_reusable_csc';
open_system(model);
```



Specify Reusable Storage Class in Code Mappings Editor

1. Open the **Embedded Coder** app.
2. To open the Code Mappings editor, click **Code Interface** > **Individual Element Code Mappings**.
3. In the model, select the `RCSC_REAL` signal line. To view this signal or any signal in the Code Mappings editor, on the **Signals/States** tab, click the **Add selected signals to code mappings** button.
4. For the row that represents the signal line, set the **Storage Class** column to `Reusable`.
5. Similary, add these signals to the Code Mappings editor and set **Storage Class** to `Reusable`:

- `RCSC_IMAG`
- `RCSC_REAL2`
- `RCSC_IMAG2`

And these signals inside the subsystem:

- `RCSC_REAL`

- RCSC_IMAG
- RCSC_REAL2
- RCSC_IMAG2

Alternatively, use these commands in the MATLAB® Command Window:

```
cmo = coder.mapping.utils.create(model);
cm = coder.mapping.api.get(model);

portHandles = get_param(...
    'ex_reusable_csc/Complex to Real-Imag','portHandles');
addSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)],...
    'StorageClass','Reusable');

portHandles = get_param(...
    'ex_reusable_csc/Downsample','portHandles');
addSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)],...
    'StorageClass','Reusable');

portHandles = get_param(...
    'ex_reusable_csc/DeltaSubsystem/DeltaX','portHandles');
addSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)],...
    'StorageClass','Reusable');

portHandles = get_param(...
    'ex_reusable_csc/DeltaSubsystem/DeltaY','portHandles');
addSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)],...
    'StorageClass','Reusable');
```

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: ex_reusable_csc
### Successful completion of build procedure for: ex_reusable_csc

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================
ex_reusable_csc  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.604s
```

For buffer reuse, the `ex_reusable_csc.c` file contains these global variables:

```
cfile = fullfile(pwd,...
    'ex_reusable_csc_ert_rtw','ex_reusable_csc.c');
rtwdemodbtype(cfile,...
    '/* Definition for custom storage class: Reusable */',...
    '/* External inputs (root inport signals with default storage) */',1,0);
```

```
/* Definition for custom storage class: Reusable */
static real_T RCSC_IMAG[1048576];      /* '<Root>/Complex to Real-Imag' */
```

```
static real_T RCSC_IMAG2[262144];      /* '<Root>/Downsample' */
static real_T RCSC_REAL[1048576];      /* '<Root>/Complex to Real-Imag' */
static real_T RCSC_REAL2[262144];      /* '<Root>/Downsample' */
```

The `ex_reusable_csc.c` file contains this code:

```
cfile = fullfile(pwd,...
    'ex_reusable_csc_ert_rtw','ex_reusable_csc.c');
rtwdemodbtype(cfile,...
    '/* Output and update for atomic system: ''<Root>/DeltaSubsystem'' */',...
    '/* Output and update for atomic system: ''<Root>/Downsample'' */',1,0);
rtwdemodbtype(cfile,...
    '/* Model step function */','/* Model initialize function */',1,0);
```

```
/* Output and update for atomic system: '<Root>/DeltaSubsystem' */
static void DeltaSubsystem(void)
{
  /* ComplexToRealImag: '<Root>/Complex to Real-Imag' incorporates:
   *  MATLAB Function: '<S1>/DeltaX'
   *  SignalConversion generated from: '<S1>/DeltaX'
   * */
  DeltaX(&RCSC_REAL2[0], &RCSC_IMAG2[0], &RCSC_REAL[0], &RCSC_IMAG[0]);

  /* SignalConversion generated from: '<Root>/Downsample' incorporates:
   *  ComplexToRealImag: '<Root>/Complex to Real-Imag'
   *  MATLAB Function: '<S1>/DeltaY'
   *  SignalConversion generated from: '<S1>/DeltaY'
   * */
  DeltaY(&RCSC_REAL[0], &RCSC_IMAG[0], &RCSC_REAL2[0], &RCSC_IMAG2[0]);
}

/* Model step function */
void ex_reusable_csc_step(void)
{
  int32_T i;

  /* ComplexToRealImag: '<Root>/Complex to Real-Imag' incorporates:
   *  Inport: '<Root>/ComplexData'
   */
  for (i = 0; i < 1048576; i++) {
    RCSC_REAL[i] = rtU.ComplexData[i].re;
    RCSC_IMAG[i] = rtU.ComplexData[i].im;
  }

  /* End of ComplexToRealImag: '<Root>/Complex to Real-Imag' */

  /* MATLAB Function: '<Root>/Downsample' */
  Downsample(&RCSC_REAL[0], &RCSC_IMAG[0], &RCSC_REAL2[0], &RCSC_IMAG2[0]);

  /* Outputs for Atomic SubSystem: '<Root>/DeltaSubsystem' */
  DeltaSubsystem();

  /* End of Outputs for SubSystem: '<Root>/DeltaSubsystem' */

  /* Outport: '<Root>/Out1' incorporates:
   *  RealImagToComplex: '<Root>/Real-Imag to Complex'
   *  SignalConversion generated from: '<S1>/DeltaY'
```

```
   * */
  for (i = 0; i < 261121; i++) {
    rtY.Out1[i].re = RCSC_REAL2[i];
    rtY.Out1[i].im = RCSC_IMAG2[i];
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The variables `RCSC_REAL` and `RCSC_IMAG` hold the outputs of the Complex to Real-Image block and `DeltaX`. These variables hold the inputs to the `DeltaY` block. The variables `RCSC_REAL2` and `RCSC_IMAG2` hold the outputs of `Downsample` and `DeltaY`. These variables hold the inputs to the `DeltaX` block. By interleaving buffers in this way, you eliminate redundant data copies in the generated code.

To remove the signals from the Code Mappings editor, on the **Signals/States** tab, select the signals and click the **Remove selected signals to code mappings** button.

Alternatively, use these commands in the MATLAB Command Window:

```
portHandles = get_param(...
    'ex_reusable_csc/Complex to Real-Imag','portHandles');
removeSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)]);

portHandles = get_param(...
    'ex_reusable_csc/Downsample','portHandles');
removeSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)]);

portHandles = get_param(...
    'ex_reusable_csc/DeltaSubsystem/DeltaX','portHandles');
removeSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)]);

portHandles = get_param(...
    'ex_reusable_csc/DeltaSubsystem/DeltaY','portHandles');
removeSignal(cm,[portHandles.Outport(1),portHandles.Outport(2)]);
```

When you generate code for the model, the `ex_reusable_csc.h` contains two extra global variables:

```
slbuild(model);
cfile = fullfile(pwd,...
    'ex_reusable_csc_ert_rtw','ex_reusable_csc.h');
rtwdemodbtype(cfile,...
    '/* Block signals and states',...
    '} D_Work;',1,0);
```

```
### Starting build procedure for: ex_reusable_csc
### Successful completion of build procedure for: ex_reusable_csc

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
================================================================================
ex_reusable_csc  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
```

```
Build duration: 0h 0m 15.752s

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T z2[262144];                    /* '<S1>/DeltaY' */
  real_T z1[262144];                    /* '<S1>/DeltaY' */
  real_T z1_m[261632];                  /* '<S1>/DeltaX' */
  real_T z2_c[261632];                  /* '<S1>/DeltaX' */
  real_T RCSC_REAL[1048576];            /* '<Root>/Complex to Real-Imag' */
  real_T RCSC_IMAG[1048576];            /* '<Root>/Complex to Real-Imag' */
```

The `ex_reusable_csc.c` file now contains this code.

```
cfile = fullfile(pwd,...
    'ex_reusable_csc_ert_rtw','ex_reusable_csc.c');
rtwdemodbtype(cfile,...
    '/* Output and update for atomic system: ''<Root>/DeltaSubsystem'' */',...
    '/* Output and update for atomic system: ''<Root>/Downsample'' */',1,0);
rtwdemodbtype(cfile,...
    '/* Model step function */','/* Model initialize function */',1,0);
```

```
/* Output and update for atomic system: '<Root>/DeltaSubsystem' */
static void DeltaSubsystem(void)
{
  /* MATLAB Function: '<S1>/DeltaX' */
  DeltaX(rtDWork.z2, rtDWork.z1, rtDWork.z1_m, rtDWork.z2_c);

  /* MATLAB Function: '<S1>/DeltaY' */
  DeltaY(rtDWork.z1_m, rtDWork.z2_c, &rtDWork.z1[0], &rtDWork.z2[0]);
}

/* Model step function */
void ex_reusable_csc_step(void)
{
  int32_T i;

  /* ComplexToRealImag: '<Root>/Complex to Real-Imag' incorporates:
   *  Inport: '<Root>/ComplexData'
   */
  for (i = 0; i < 1048576; i++) {
    rtDWork.RCSC_REAL[i] = rtU.ComplexData[i].re;
    rtDWork.RCSC_IMAG[i] = rtU.ComplexData[i].im;
  }

  /* End of ComplexToRealImag: '<Root>/Complex to Real-Imag' */

  /* MATLAB Function: '<Root>/Downsample' */
  Downsample(rtDWork.RCSC_REAL, rtDWork.RCSC_IMAG, rtDWork.z2, rtDWork.z1);

  /* Outputs for Atomic SubSystem: '<Root>/DeltaSubsystem' */
  DeltaSubsystem();

  /* End of Outputs for SubSystem: '<Root>/DeltaSubsystem' */

  /* Outport: '<Root>/Out1' incorporates:
   *  RealImagToComplex: '<Root>/Real-Imag to Complex'
   */
```

```
  for (i = 0; i < 261121; i++) {
    rtY.Out1[i].re = rtDWork.z1[i];
    rtY.Out1[i].im = rtDWork.z2[i];
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The generated code contains two additional global variables for holding block inputs and outputs.

Limitations for the Model That Use Reusable Storage Class in Code Mappings Editor

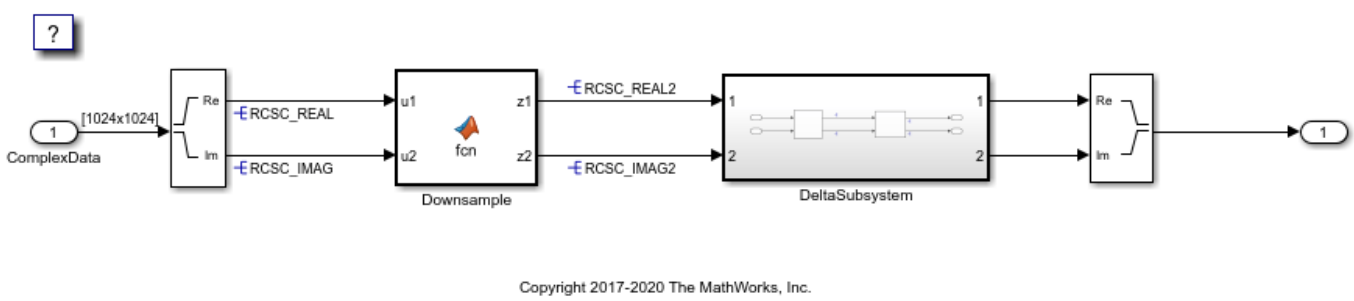
For reused data elements, you can specify the same value for the **Identifier** property. If you do not specify a value for the **Identifier**, the code generator uses the same signal label to name the reusable signal in the generated code. The code generator does not reuse signals if:

- The signals have the same labels but different identifiers.
- The signals have the same identifier but different values for these properties: **DataScope**, **HeaderFile**, **DefinitionFile**, and **Owner**. If there is a mismatch in any of the properties, the code generator stops and produces an error if the model configuration parameter **Detect non-reused custom storage classes** is set to `error`.

Specify a Simulink Signal Object for Reuse

1. Open the model `ReusableStorageClass`.

```
model ='ReusableStorageClass';
open_system(model);
```



2. The signals in the model resolve to the corresponding `Simulink.Signal` data objects in the Base Workspace. Right-click on the `RCSC_REAL` signal line. From the context menu, select **Properties**. In the Signal Properties dialog box, inspect that the signal name is `RCSC_REAL` and the parameter **Signal name must resolve to Simulink signal object** is selected. This setting indicates that there is a `Simulink.Signal` data object in the Base Workspace with the same name as `RCSC_REAL`.

3. In the model, navigate into the `DeltaSubsystem` subsystem. Select the `RCSC_REAL` signal line in this subsystem. This signal also resolves to the signal object in the Base Workspace.

4. In the Base Workspace, double click the data object `RCSC_REAL`. The `Simulink.Signal` dialog box opens.

5. On the **Code Generation** tab, inspect that the **Storage class** parameter is set to `Reusable`. This setting instructs the code generator to generate code for the signal as a global variable named `RCSC_REAL`. With the `Reusable` storage class, the generated code can store the output of the

Complex to Real-Imag block (at the root-level of the model) and the output of the MATLAB Function block `DeltaX` (in the subsystem) in the `RCSC_REAL` global variable.

6. In the **Apps** gallery, under **Code Generation**, click **Embedded Coder**. The **C Code** tab opens.

7. Click the **Build** button.

The generated code achieves the same results as the preceding workflow where you add signals to the Code Mappings editor and set **Storage Class** to `Reusable`.

Note: You can specify buffer reuse on signals that the code generator cannot implement. For those cases, use two diagnostics to specify the message type that the model displays. In the Configuration Parameters dialog box, these diagnostics are **Detect non-reused custom storage classes** and **Detect ambiguous custom storage class final values**.

```
bdclose(model)
```

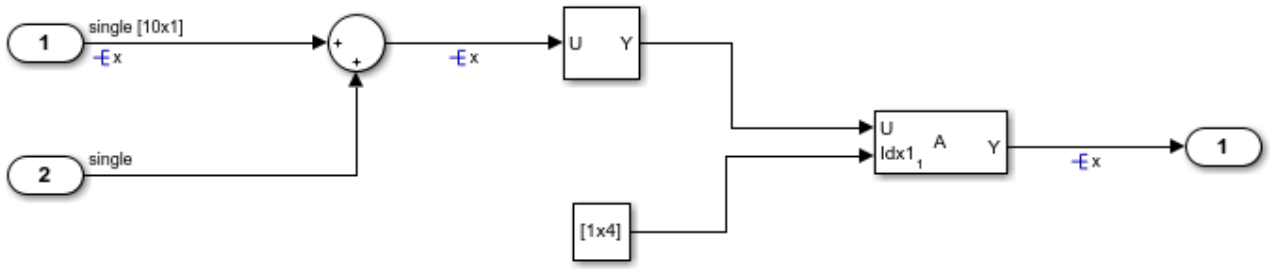
Buffer Reuse for Unit Delay and Delay Blocks

To reuse the signal of a Unit Delay or Delay block, use the same reusable storage class specification for a pair of input and state arguments or a pair of output and state arguments of a Unit Delay or a Delay block. For Delay blocks, you must set the **Delay length** parameter to `1` and **Initial condition > Source** to `Dialog`. To access these parameters, in the model, open the Property Inspector and click the block in the model.

Limitations for Root Inport and Outport Signals

These limitations apply to a model in which you specify buffer reuse for a pair of root inport and outport signals:

- The output ports cannot be conditional.
- If the code generator cannot reuse the same buffer in a top model, the generated code contains additional buffers. If the top model is a referenced model, the code generator reports an error. To resolve the error, remove the `Simulink.Signal` specification from the signal that connects to the outport port.
- When you run the executable that the code generator produces, and you reuse a pair of root inport and outport signals, when the root input value is zero, the root output value must also be zero. If the output value is nonzero and you reuse the signals, then the results from the simulation can differ from the results that the executable produces.
- There might be a simulation versus code generation mismatch when you assign the same Reusable storage class to pair of root inport and root outport signals. This mismatch occurs when an Assignment block drives the Root Outport block, and the Assignment block does not assign a value to every element of the output signal because the Y port of the Assignment block is not active. This image shows a model in which the mismatch occurs:



During simulation, the unwritten Assignment block output values are zero. During code generation, the unwritten output values are the same as the input.

Limitations for the Model That Use Simulink Signal Object

These limitations apply to a model in which you specify buffer reuse for signals by using a `Simulink.Signal` data object:

- Signals that you specify for reuse must have the same data types and sampling rates.
- For user-specified buffer reuse, blocks that modify a signal specified for reuse must execute before blocks that use the original signal value. Sometimes the code generator has to change the block operation order so that buffer reuse can occur. For models in which the code generator is unable to reorder block operations, buffer reuse does not occur.
- For models in which the code generator reorders block operations so that `Simulink.Signal` reuse can occur, you can observe the difference in the sorted order. On the **Debug** tab, in the **Diagnostics** section select the **Information Overlays** drop-down arrow. In the **Blocks** section of the dialog, select **Execution Order**. To display the sorted execution order during simulation, on the **Modeling** tab, select **Update Model** . To display the execution order in the generated code, on the **C Code** tab, select **Build Model**.
- You can specify buffer reuse for a chain of reusable subsystems as long as these subsystems do not get reused anywhere in the model.

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “C Code Generation Configuration for Model Interface Elements”
- “Choose Storage Class for Controlling Data Representation in Generated Code” on page 36-75
- “Virtualized Output Ports Optimization” on page 78-15
- “Design Data Interface by Configuring Inport and Outport Blocks” on page 36-169

Specify Buffer Reuse for MATLAB Function Blocks in a Path

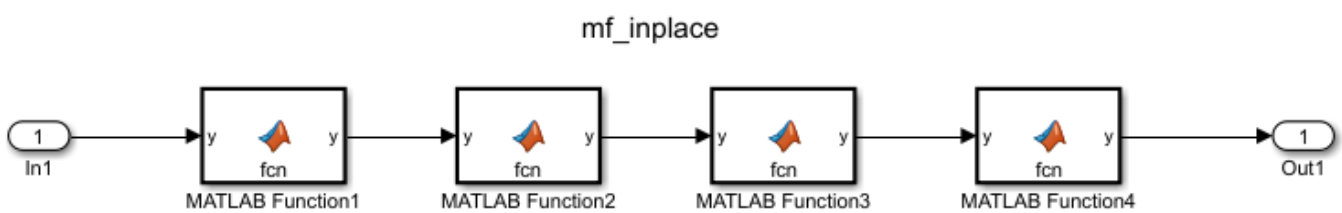
In this section...

"Example Model" on page 78-25
"Generate Code with Optimization" on page 78-25

You can specify buffer reuse across MATLAB Function blocks by using the same variable name for the input and output arguments. The code generator tries to reuse the output of one MATLAB Function block as the input to the next MATLAB Function block. This optimization conserves RAM and ROM consumption and reduces data copies.

Example Model

1. Use Inport, Outport, and MATLAB Function blocks to create the model `mf_inplace`.



2. Open each MATLAB Function block and copy the following code:

```
function y = fcn(y)
%#codegen

y=y+4;
```

3. Open the Configuration Parameters dialog box. On the **Code Generation** tab, change the **System target file** to `ert.tlc`.
4. On the **Solver** tab, change the **Type** parameter to `Fixed-step`.

Generate Code with Optimization

Generate code for the model. The `mf_inplace.c` file contains this code:

```
void mf_inplace_MATLABFunction(real_T *rty_y)
{
  *rty_y += 4.0;
}

void mf_inplace_step(void)
{
  real_T rtb_y_p5;
  rtb_y_p5 = mf_inplace_U.In1;
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_Y.Out1 = rtb_y_p5;
  mf_inplace_MATLABFunction(&mf_inplace_Y.Out1);
}
```

The code generator reuses the variable `rtb_y_p5` for the input and output arguments of each MATLAB Function block.

Note On the **Code Generation** tab in the Subsystem Block Parameters dialog box, if the **Function packaging** parameter is set to `Nonreusable function` and the **Function interface** parameter is set to `Allow arguments (Optimized)`, the code generator cannot reuse the input and output arguments.

See Also

Related Examples

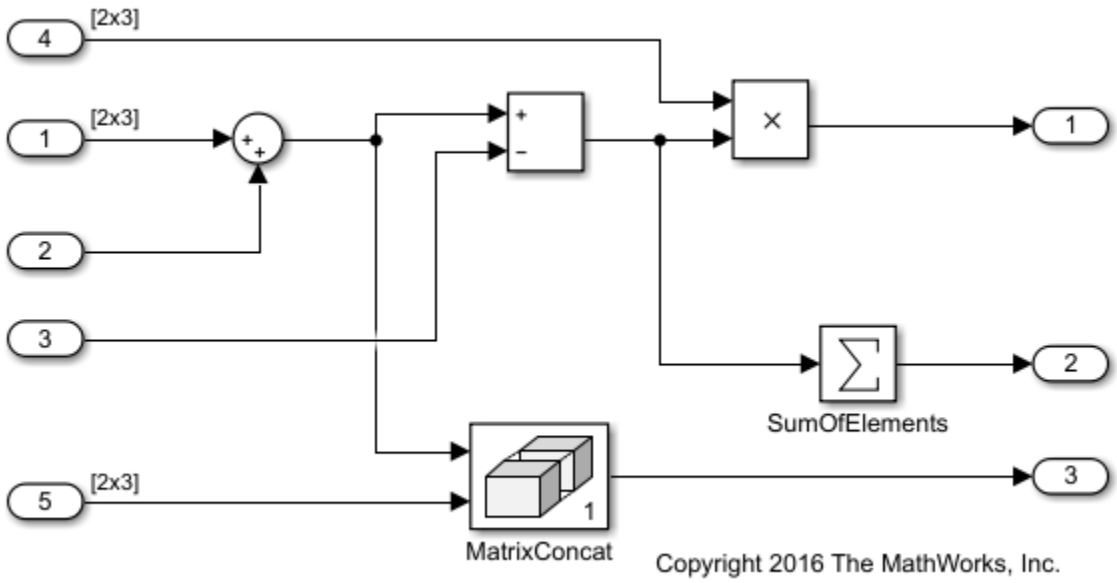
- “Implement MATLAB Functions in Simulink with MATLAB Function Blocks”
- “Specify Buffer Reuse for Signals in a Path” on page 78-17

Remove Data Copies by Reordering Block Operations in the Generated Code

This example show how to remove data copies by changing the **Optimize block order in the generated code** parameter from `off` to `Improved Execution Speed`. Changing this setting indicates to the code generator to reorder block operations where possible to remove data copies. This parameter is in the Configuration Parameters dialog box. This optimization conserves RAM and ROM consumption.

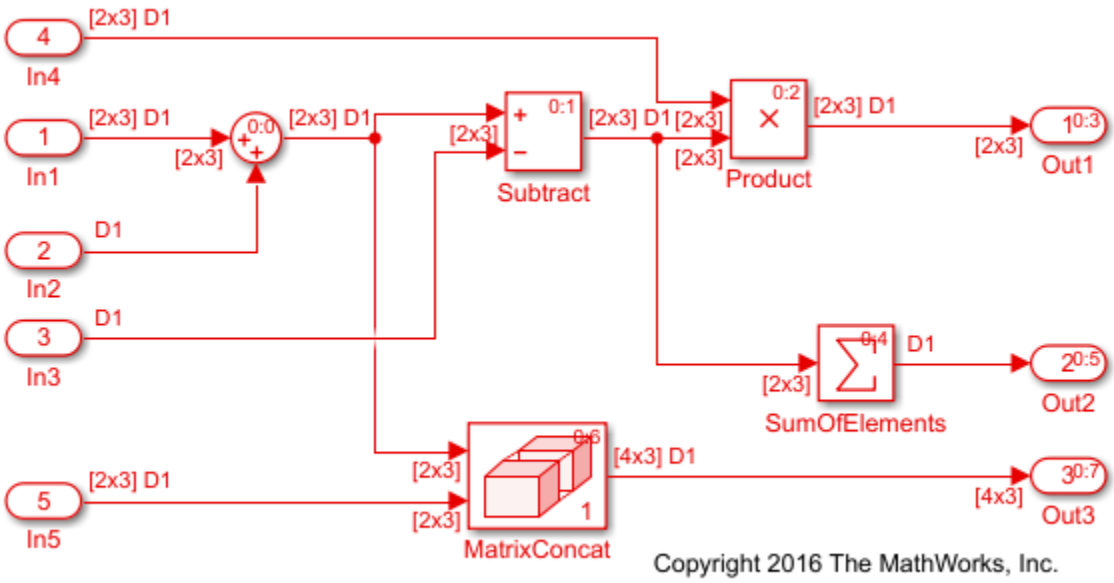
Example Model

In the model `ex_optimizeblockorder`, the signal that leaves the Sum block enters a Subtract block and a Concatenate block. The signal that leaves the Subtract block enters a Product block and a Sum of Elements block.



Generate Code without Optimization

The image shows the model `ex_optimizeblockorder` after a model build. The red numbers indicate the default block order in the generated code. The Subtract block executes before the Concatenate block. The Product block executes before the Sum of Elements block.



View the generated code without the optimization. Here is the `ex_optimizeblockorder_step` function.

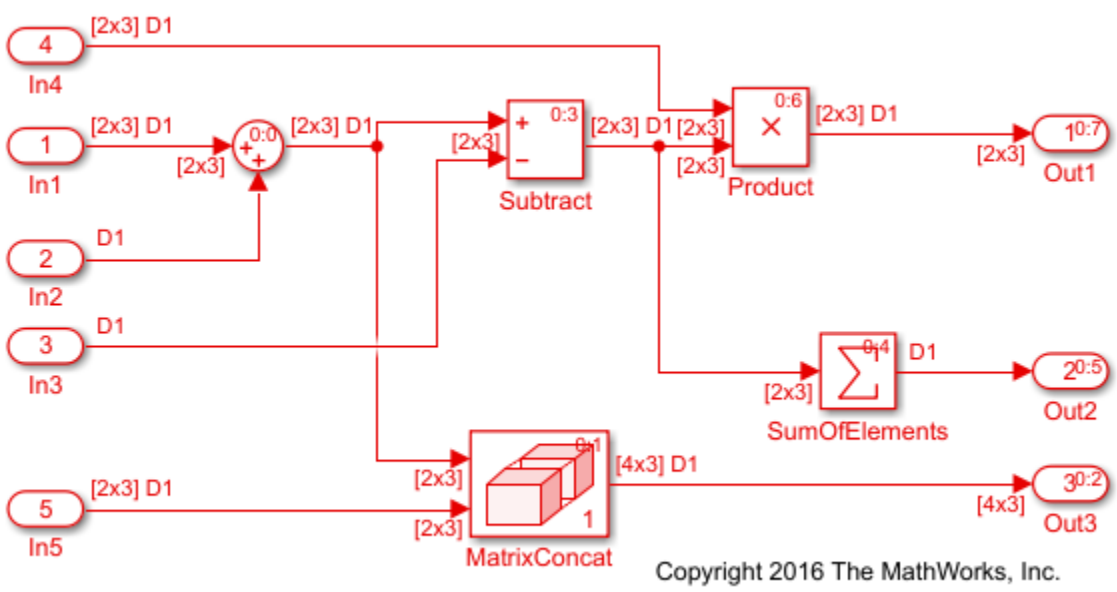
```
/* Model step function */
void ex_optimizeblockorder_step(void)
{
  real_T rtb_Sum2x3[6];
  int32_T i;
  real_T rtb_Sum2x3_d;
  real_T rtb_Subtract;

  /* Sum: '<Root>/SumOfElements' */
  rtY.Out2 = -0.0;
  for (i = 0; i < 6; i++) {
    /* Sum: '<Root>/Sum2x3' incorporates:
     *  Inport: '<Root>/In1'
     *  Inport: '<Root>/In2'
     */
    rtb_Sum2x3_d = rtU.In1[i] + rtU.In2;

    /* Sum: '<Root>/Subtract' incorporates:
     *  Inport: '<Root>/In3'
     */
    rtb_Subtract = rtb_Sum2x3_d - rtU.In3;

    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In4'
     *  Product: '<Root>/Product'
     */
    rtY.Out1[i] = rtU.In4[i] * rtb_Subtract;

    /* Sum: '<Root>/Sum2x3' */
    rtb_Sum2x3[i] = rtb_Sum2x3_d;
```

```
    /* Sum: '<Root>/SumOfElements' */
    rtY.Out2 += rtb_Subtract;
  }

  /* Concatenate: '<Root>/MatrixConcat ' */
  for (i = 0; i < 3; i++) {
    /* Outport: '<Root>/Out3' incorporates:
     *  Inport: '<Root>/In5'
     */
    rtY.Out3[i << 2] = rtb_Sum2x3[i << 1];
    rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
    rtY.Out3[1 + (i << 2)] = rtb_Sum2x3[(i << 1) + 1];
    rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
  }

  /* End of Concatenate: '<Root>/MatrixConcat ' */
}
```

With the default order, the generated code contains three buffers, `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract`. The generated code contains these temporary variables and associated data copies because the Matrix Concatenate block must use the output from the Sum block and the Sum of Elements block must use the output from the Subtract block.

Generate Code with Optimization

The image shows the `ex_optimizeblockorder` model after setting the **Optimize block operation order in generated code** parameter to `Improved Execution Speed` and building the model. The Subtract block executes after the Concatenate block. The Product block executes after the Sum of Elements block.



In the optimized code, the three buffers `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract` and their associated data copies are gone. The generated code does not require these temporary variables to hold the outputs of the Sum and Subtract blocks because the Subtract block executes after the Concatenate block and the Product block executes after the Sum of Elements block.

```
/* Model step function */
void ex_optimizeblockorder_step(void)
{
  int32_T i;

  /* Sum: '<Root>/Sum2x3' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  for (i = 0; i < 6; i++) {
    rtY.Out1[i] = rtU.In1[i] + rtU.In2;
  }

  /* End of Sum: '<Root>/Sum2x3' */

  /* Concatenate: '<Root>/MatrixConcat ' */
  for (i = 0; i < 3; i++) {
    /* Outport: '<Root>/Out3' incorporates:
     *  Inport: '<Root>/In5'
     */
    rtY.Out3[i << 2] = rtY.Out1[i << 1];
    rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
    rtY.Out3[1 + (i << 2)] = rtY.Out1[(i << 1) + 1];
    rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
  }

  /* End of Concatenate: '<Root>/MatrixConcat ' */

  /* Sum: '<Root>/SumOfElements' */
  rtY.Out2 = -0.0;
  for (i = 0; i < 6; i++) {
    /* Sum: '<Root>/Subtract' incorporates:
     *  Inport: '<Root>/In3'
     */
    rtY.Out1[i] -= rtU.In3;

    /* Sum: '<Root>/SumOfElements' */
    rtY.Out2 += rtY.Out1[i];

    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In4'
     *  Product: '<Root>/Product'
     */
    rtY.Out1[i] *= rtU.In4[i];
  }
}
```

To implement buffer reuse, the code generator does not violate user-specified block priorities.

See Also

Optimize block operation order in generated code

Related Examples

- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code” on page 79-52
- “Data Copy Reduction”

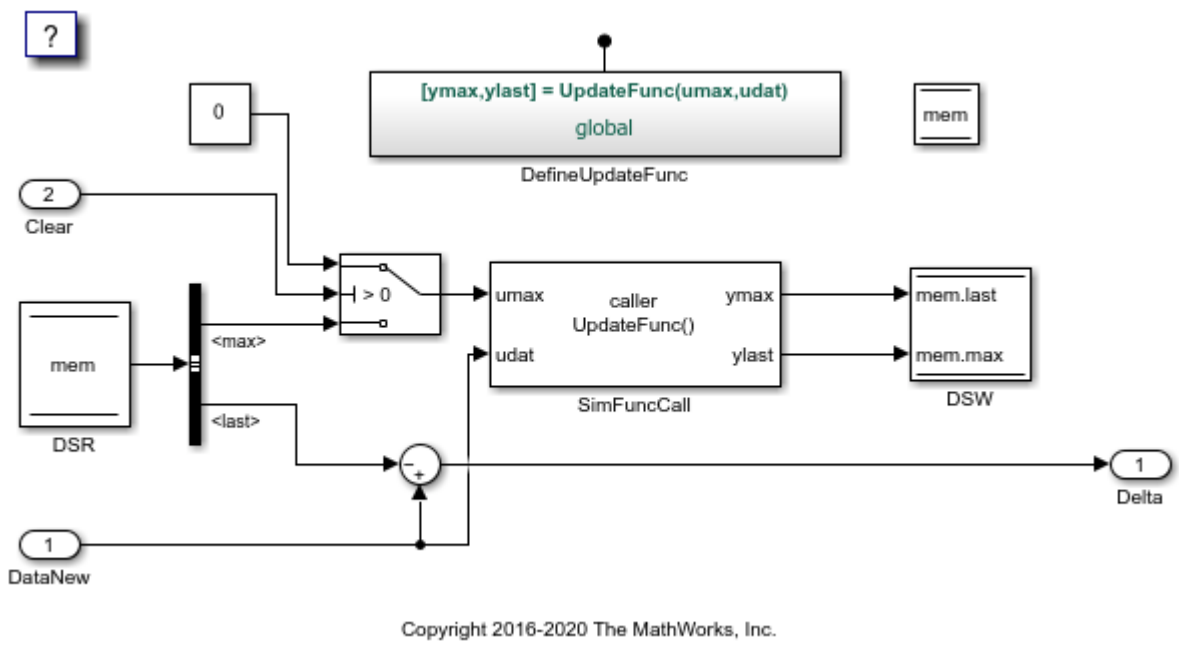
Data Copy Reduction for Data Store Read and Data Store Write Blocks

This example shows how the code generator removes temporary buffers for Data Store Read and Data Store Write blocks. This optimization improves execution speed and reduces RAM consumption.

Example Model

The model `rtwdemo_optimizedatastorebuffers` contains the Function caller `UpdateFunc`, which calls the Simulink Function `DefineUpdateFunc`. The Data Store Read block `DSR` reads from `mem`. The Data Store Write block `DSW` writes to `mem`.

```
model='rtwdemo_optimizedatastorebuffers';
open_system(model);
```



Generate Code Without Optimization

In the Configuration Parameters dialog box, deselect the **Reuse buffers for Data Store Read and Data Store Write blocks parameter** or at the MATLAB® command prompt, enter this command:

```
set_param(model,'OptimizeDataStoreBuffers','off');
```

Build the model.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_optimizedatastorebuffers
### Successful completion of build procedure for: rtwdemo_optimizedatastorebuffers
```

```
Build Summary

Top model targets built:

Model                              Action                        Rebuild Reason
==========================================================================================
rtwdemo_optimizedatastorebuffers  Code generated and compiled.  Code generation information file

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.752s
```

View the generated code without the optimization. This code is in `rtwdemo_optimizedatastorebuffers.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimizedatastorebuffers_ert_rtw',...
    'rtwdemo_optimizedatastorebuffers.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void step(void)
{
  real_T rtb_DSR_last;
  real_T rtb_SimFuncCall_o1;
  real_T rtb_Sum_p;

  /* DataStoreRead: '<Root>/DSR' */
  rtb_DSR_last = mem.last;

  /* Switch: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   *  DataStoreRead: '<Root>/DSR'
   *  Inport: '<Root>/Clear'
   */
  if (rtU.Clear) {
    rtb_SimFuncCall_o1 = 0.0;
  } else {
    rtb_SimFuncCall_o1 = mem.max;
  }

  /* End of Switch: '<Root>/Switch' */

  /* FunctionCaller: '<Root>/SimFuncCall' incorporates:
   *  Inport: '<Root>/DataNew'
   */
  UpdateFunc(rtb_SimFuncCall_o1, rtU.DataNew, &rtb_SimFuncCall_o1, &rtb_Sum_p);

  /* DataStoreWrite: '<Root>/DSW' */
  mem.last = rtb_SimFuncCall_o1;
  mem.max = rtb_Sum_p;

  /* Outport: '<Root>/Delta' incorporates:
   *  DataStoreRead: '<Root>/DSR'
   *  Inport: '<Root>/DataNew'
   *  Sum: '<Root>/Sum'
   */
  rtY.Delta = rtU.DataNew - rtb_DSR_last;
}
```

The generated code contains data copies for the Data Store Read and Data Store Write blocks, respectively.

Generate Code with Optimization

In the Configuration Parameters dialog box, clear the **Reuse buffers for Data Store Read and Data Store Write blocks** parameter or at the MATLAB command prompt, enter this command:

```
set_param(model,'OptimizeDataStoreBuffers','on');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_optimizedatastorebuffers
### Successful completion of build procedure for: rtwdemo_optimizedatastorebuffers

Build Summary

Top model targets built:

Model                              Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizedatastorebuffers  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.481s
```

View the generated code with the optimization. This code is in `rtwdemo_optimizedatastorebuffers.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimizedatastorebuffers_ert_rtw',...
    'rtwdemo_optimizedatastorebuffers.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void step(void)
{
  real_T rtb_DSR_last;
  real_T tmp;

  /* DataStoreRead: '<Root>/DSR' */
  rtb_DSR_last = mem.last;

  /* Switch: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   *  DataStoreRead: '<Root>/DSR'
   *  Inport: '<Root>/Clear'
   */
  if (rtU.Clear) {
    tmp = 0.0;
  } else {
    tmp = mem.max;
  }

  /* FunctionCaller: '<Root>/SimFuncCall' incorporates:
   *  DataStoreWrite: '<Root>/DSW'
```

```
   *  Inport: '<Root>/DataNew'
   *  Switch: '<Root>/Switch'
   */
  UpdateFunc(tmp, rtU.DataNew, &mem.last, &mem.max);

  /* Outport: '<Root>/Delta' incorporates:
   *  DataStoreRead: '<Root>/DSR'
   *  Inport: '<Root>/DataNew'
   *  Sum: '<Root>/Sum'
   */
  rtY.Delta = rtU.DataNew - rtb_DSR_last;
}
```

The data copy for the Data Store Write block is not in the generated code. The code contains the data copy for the Data Store Read block because the Sum block executes after the Data Store Write block. The generated code contains the variable `rtb_DSR_last` to hold the output of the Sum block. Therefore, the Sum block gets the values that `SimFuncCall` calculates at the start of the time step rather than those values at the next time step. If the priority of the Sum block is lower than `SimFuncCall`, the code generator can remove the data copy for the Data Store Read block. Some other cases in which the code generator might not eliminate data copies are:

- A Simulink Function internally writes to the Data Store Memory block.
- The Data Store Read or Data Store Write blocks select elements of an array from the Data Store Memory block.
- The Data Store Memory block has a custom storage class.
- The Data Store Read and Data Store Write blocks occur on the same block unless that block is a Bus Assignment block or an Assignment block.

Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Reuse buffers for Data Store Read and Data Store Write blocks

Related Examples

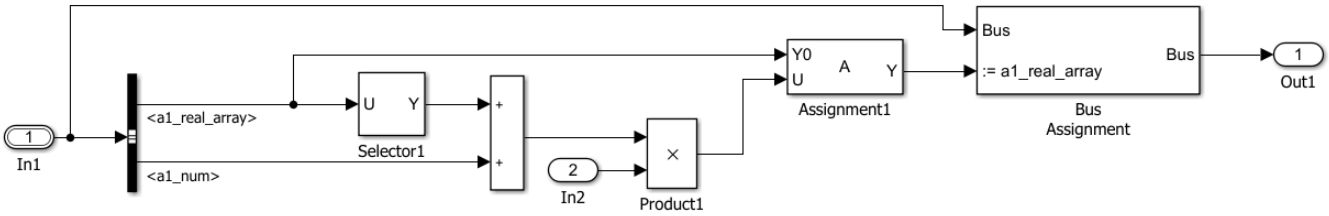
- “Reuse Local Block Outputs and Generate Code”

Reduce Data Copies for Bus Assignment Blocks in Generated Code

For models containing a Bus Assignment block, if possible, the code generator uses the same variable for the block input and output. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

Example Model

For example, in `bus_assignoptim`, a bus signal containing six elements feeds into a Bus Assignment block and a Bus Selector block. The Bus Assignment block assigns new values to the bus element `a1_real_array`. This bus signal feeds into `Out1`.



Generate Code Without Optimization

Generate code without the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to `off`. Without the optimization the `bus_assignoptim_step` function contains this code:

```
void bus_assignoptim_step(void)
{
  real_T rtb_Assignment[36];
  int32_T i;

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   *  Selector: '<Root>/Selector'
   *  Sum: '<Root>/Sum1'
   */
  for (i = 0; i < 36; i++) {
    rtb_Assignment[i] = bus_assignoptim_U.In1.a1_real_array[i];
  }

  for (i = 0; i < 2; i++) {
    rtb_Assignment[(int32_T)(i + 22)] = (bus_assignoptim_U.In1.a1_real_array
      [(int32_T)(i + 22)] + bus_assignoptim_U.In1.a1_num) *
      bus_assignoptim_U.In2;
  }

  /* End of Assignment: '<Root>/Assignment' */

  /* Outport: '<Root>/Out' incorporates:
```

```
   *  BusAssignment: '<Root>/Bus Assignment'
   *  Inport: '<Root>/In1'
   */
  bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

  /* BusAssignment: '<Root>/Bus Assignment' incorporates:
   *  Outport: '<Root>/Out'
   */
  for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.a1_real_array[i] = rtb_Assignment[i];
  }
}
```

The generated code contains the temporary array `rtb_Assignment` for holding data before this data is assigned to `bus_assignoptim_Y.Out.a1_real_array`.

Generate Code with Optimization

Generate code with the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to `on`. With the optimization, the `bus_assignoptim_step` function contains this code:

```
/* Model step function */
void bus_assignoptim_step(void)
{
  int32_T i;

  /* Outport: '<Root>/Out' incorporates:
   *  Inport: '<Root>/In1'
   *  SignalConversion: '<Root>/TmpBusAssignmentBufferAtBus AssignmentInport1'
   */
  bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Outport: '<Root>/Out'
   *  Product: '<Root>/Product'
   *  Selector: '<Root>/Selector'
   *  Sum: '<Root>/Sum1'
   */
  for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.a1_real_array[i] =
      bus_assignoptim_U.In1.a1_real_array[i];
  }

  for (i = 0; i < 2; i++) {
    bus_assignoptim_Y.Out.a1_real_array[(int32_T)(i + 22)] =
      (bus_assignoptim_U.In1.a1_real_array[(int32_T)(i + 22)] +
       bus_assignoptim_U.In1.a1_num) * bus_assignoptim_U.In2;
  }

  /* End of Assignment: '<Root>/Assignment' */
}
```

The generated code does not contain the temporary array `rtb_Assignment1` for holding data. The generated code directly assigns this data to `bus_assignoptim_Y.Out.a1_real_array`.

See Also

Perform in-place updates for Assignment and Bus Assignment blocks

Related Examples

- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code” on page 79-52
- “Data Copy Reduction”

Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse

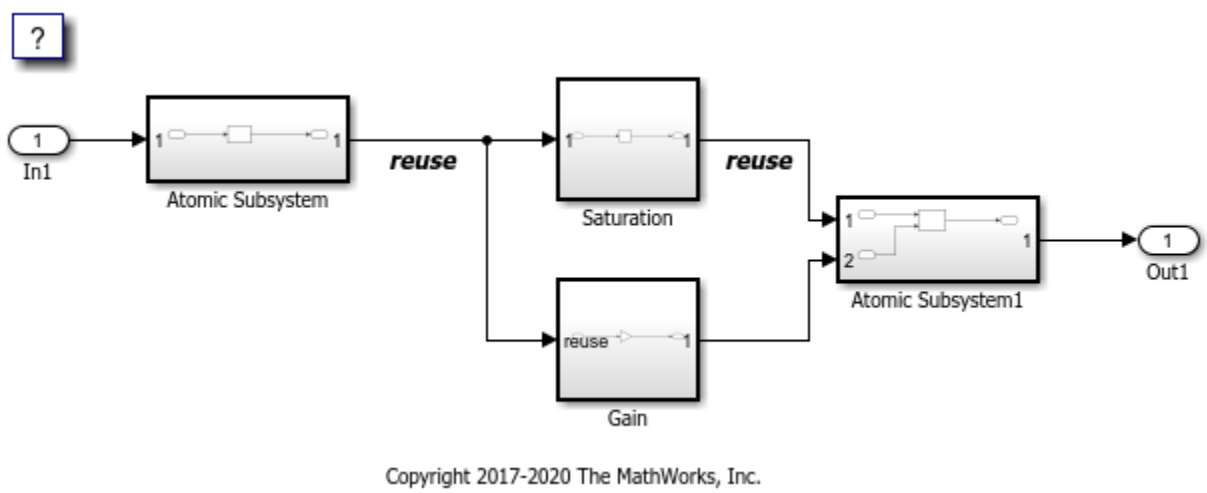
If your model has the optimal parameter settings for removing data copies, you might be able to remove additional data copies by using signal labels. After studying the generated code and the Static Code Metrics Report and identifying areas where you think buffer reuse is possible, you can add labels to signal lines. If possible, the code generator reorders block operations to implement the reuse specification.

Specifying buffer reuse improves execution speed and may reduce RAM consumption.

Example Model

The model `SignalLabelReuse` demonstrates how you can use signal labels to request buffer reuse for block input and output signals. The generated code can use the same variable for the Saturation and Atomic Subsystem block outputs because those signals contain the same label and the **Use Signal Labels to Guide Buffer Reuse** parameter is selected.

```
model='SignalLabelReuse';
open_system(model);
```



Generate Code Without Optimization

The **Use Signal Labels to Guide Buffer Reuse** parameter is on by default. Open the Configuration Parameters dialog box and clear the parameter. Alternatively, use the command-line.

```
set_param(model, 'LabelGuidedReuse', 'off');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: SignalLabelReuse
### Successful completion of build procedure for: SignalLabelReuse

Build Summary
```

```
Top model targets built:

Model             Action                        Rebuild Reason
==========================================================================================
SignalLabelReuse  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.884s
```

View the generated `SignalLabelReuse_step` function.

```
file = fullfile('SignalLabelReuse_ert_rtw','SignalLabelReuse.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void SignalLabelReuse_step(void)
{
  real_T rtb_Bias[4];
  real_T rtb_Saturation[4];
  real_T rtb_Bias_0;
  int32_T i;
  AtomicSubsystem(rtU.In1, rtb_Bias);
  for (i = 0; i < 4; i++) {
    rtb_Bias_0 = rtb_Bias[i];
    if (rtb_Bias_0 > 10.0) {
      rtb_Saturation[i] = 10.0;
    } else if (rtb_Bias_0 < -10.0) {
      rtb_Saturation[i] = -10.0;
    } else {
      rtb_Saturation[i] = rtb_Bias_0;
    }

    rtb_Bias[i] = 3.0 * rtb_Bias_0;
  }

  AtomicSubsystem1(rtb_Saturation, rtb_Bias, rtY.Out1);
}
```

By default, within the for loop, the Gain block executes after the Saturation block. The generated code contains two local variables `rtb_Saturation` and `rtb_Bias` for holding intermediate results.

Generate Code with Optimization

Open the Configuration Parameters dialog box. Select the **Use Signal Labels to Guide Buffer Reuse** parameter. Alternatively, use the command-line.

```
set_param(model, 'LabelGuidedReuse', 'on');
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: SignalLabelReuse
### Successful completion of build procedure for: SignalLabelReuse

Build Summary
```

```
Top model targets built:

Model             Action                        Rebuild Reason
==================================================================================
SignalLabelReuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.511s
```

View the generated `SignalLabelReuse_step` function.

```
file = fullfile('SignalLabelReuse_ert_rtw','SignalLabelReuse.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1)
```

```
/* Model step function */
void SignalLabelReuse_step(void)
{
  real_T rtb_Gain1[4];
  real_T Out1;
  int32_T i;
  AtomicSubsystem(rtU.In1, rtY.Out1);
  for (i = 0; i < 4; i++) {
    Out1 = rtY.Out1[i];
    rtb_Gain1[i] = 3.0 * Out1;
    if (Out1 > 10.0) {
      rtY.Out1[i] = 10.0;
    } else if (Out1 < -10.0) {
      rtY.Out1[i] = -10.0;
    } else {
      rtY.Out1[i] = Out1;
    }
  }

  AtomicSubsystem1(rtY.Out1, rtb_Gain1, rtY.Out1);
}
```

The code generator changed the block execution order so that within the `for` loop, the Saturation block executes after the Gain block. The generated code contains one local variable `rtb_Gain1` for holding intermediate results.

In addition to requesting which buffers to reuse, another possible use case is to request buffer reuse for block input and output signals that are complex.

Using Label-Based Reuse Versus Reusable Storage Classes

You can also use the same `Reusable` storage class specification on different signal lines to specify which buffers to reuse. As compared to using `Reusable` storage classes, using signal labels to specify reuse has these benefits:

- Signal labels are not observation points in the generated code, so using them does not block other optimizations such as dead code elimination and expression folding.
- The code generator does not force a signal with a label to be a global variable in the generated code. The signal can be a local or global variable.

- Signals with labels can reuse buffers with signals that do not have labels, so reuse among local and global variables is possible.

Using signal labels is a more conservative method of specifying reuse than using `Reusable` storage classes. The code generator does not implement label-based reuse when it can implement reuse by using `Reusable` storage classes in these cases:

- Reuse that can potentially prevent other optimizations from occurring in the generated code.
- Reuse on root inport and outport ports.
- Reuse across model reference or subsystem boundaries.
- Within a subsystem, buffer reuse on an intermediary signal and an input or output port.

See Also

Use signal labels to guide buffer reuse

Related Examples

- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code” on page 79-52
- “Remove Data Copies by Reordering Block Operations in the Generated Code” on page 78-27
- “Specify Buffer Reuse for Signals in a Path” on page 78-17

Execution Speed in Embedded Coder

- “Reduce Memory Usage for Signals” on page 79-2
- “Remove Zero-Initialization Code” on page 79-4
- “Simplify Multiply Operations in Array Indexing” on page 79-9
- “Replace boolean with Specific Integer Data Type” on page 79-12
- “Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow” on page 79-15
- “Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data” on page 79-18
- “Optimize Generated Code by Consolidating Redundant If-Else Statements” on page 79-22
- “Optimize Generated Code for Fixed-Point Data Operations” on page 79-26
- “Unroll Parallel for-Loop That Has Small Number of Iterations” on page 79-28
- “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31
- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code” on page 79-52
- “Optimize SIMD Code by Performing Fused Multiply Add Operations” on page 79-63
- “Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface” on page 79-67
- “Accelerate Pedestrian Detection with SIMD Code” on page 79-69

Reduce Memory Usage for Signals

The configuration parameter **Signal storage reuse** reduces the memory requirements of your real-time program by enabling parameters that reuse the memory allocated for signals with an `auto` storage class. These parameters are the following:

- **Enable local block outputs**
- **Reuse local block outputs**
- **Reuse global block outputs**
- **Optimize global data access**

When the **Enable local block outputs** parameter is on, where possible the code generator declares block outputs as local variables instead of global variables. Replacing global variables with local variables improves execution speed and reduces RAM/ROM consumption. Creating more local variables can increase stack usage. Some of the global variables that the code generator can localize include:

- Global signals that cross subsystem boundaries
- Global signals across Simulink and Stateflow domains
- Unused global state variables
- Redundant local Data Store Memory block signals

When the **Reuse local block outputs** parameter is on, wherever possible the code generator reuses local (function) variables for block outputs. When you select **Reuse global block outputs**, the code generator reuses global (function) variables wherever possible.

The **Optimize global data access** parameter has these settings:

- `None`
- `Use global to hold temporary results`
- `Minimize global data access`

When you select `None`, the code generator uses the default optimizations. The setting `Use global to hold temporary results` maximizes the use of global variables. The setting `Minimize global data access` minimizes the use of global variables by using local variables to hold intermediate values.

Clearing **Signal storage reuse** makes all block outputs global and unique, which often significantly increases RAM and ROM usage.

The code generator does not localize global variables for MATLAB system objects or AUTOSAR.

See Also

Related Examples

- “Enable and Reuse Local Block Outputs in Generated Code”
- “Reuse Global Block Outputs in the Generated Code” on page 78-12

- “Optimize Global Variable Usage” on page 78-2

Remove Zero-Initialization Code

The Remove root level I/O zero initialization and Remove internal data zero initialization parameters control whether the generated code contains initialization code for internal data (block states and block outputs) and external data (root inports and outports) whose value is zero. Eliminating zero-initialization code accelerates model initialization, reduces ROM consumption, and increases the execution speed of the generated code.

During startup, standards-compliant C and C++ compilers initialize global data to zero, eliminating the need to include zero-initialization code for this data in the generated code. Standards-compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before selecting the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** parameters:

- If your compiler is not standards-compliant, confirm that it initializes global data to zero.
- If you set the Code interface packaging parameter to `Reusable function` or `C++ class`, confirm that data is either statically allocated or that the code initializes dynamically allocated data to zero.

If you set the **Code interface packaging** parameter to `Reusable function` and select the Use dynamic memory allocation for model initialization parameter:

- The **Remove root level I/O zero initialization** and **Remove internal data zero initialization** check boxes are cleared.
- At the command line, `ZeroExternalMemoryAtStartup` and `ZeroInternalMemoryAtStartup` parameters are set to `'on'`.

If you set the **Code interface packaging** parameter to `C++ class` and select the Use dynamic memory allocation for model block instantiation parameter:

- The **Remove internal data zero initialization** check box is cleared.
- The `ZeroInternalMemoryAtStartup` parameter is set to `'on'` and is read-only.

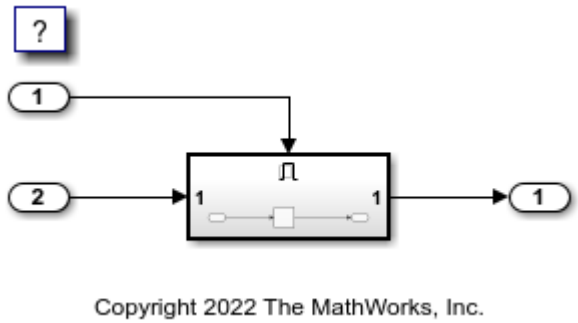
Remove Zero Initialization Code for Internal Data

This example shows how to eliminate code that initializes internal data to zero.

Example Model

Open the `InternalZeroInitialization` model. The model contains an enabled subsystem whose initial output is zero. The subsystem contains a Unit Delay block whose initial condition is `0`.

```
model = 'InternalZeroInitialization';
open_system(model);
```



Generate Code Without Optimization

Build the model by using Embedded Coder®.

```
evalc('slbuild(model)');
```

This code is in the `InternalZeroInitialization.c` file.

```
cfile = fullfile('InternalZeroInitialization_ert_rtw','InternalZeroInitialization.c');
rtwdemodbtype(cfile,'/* Model initialize', '* File trailer', 1, 0);
```

```
/* Model initialize function */
void InternalZeroInitialization_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtM, (NULL));

  /* states (dwork) */
  (void) memset((void *)&rtDWork, 0,
                sizeof(D_Work));

  /* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem' */
  /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
  rtDWork.UnitDelay_DSTATE = 0.0;

  /* End of SystemInitialize for SubSystem: '<Root>/Enabled Subsystem' */
}

/*
```

Enable Optimization

Open the Configuration Parameters dialog box. On the **Optimization** pane, select **Remove internal data zero initialization**.

Alternatively, you can use the command prompt to enable the optimization. Set the model parameter `ZeroInternalMemoryAtStartup` to `'off'`.

```
set_param(model, 'ZeroInternalMemoryAtStartup', 'off');
```


```
set_param(model, 'ZeroInternalMemoryAtStartup', 'off');
```

Generate Code with Optimization

Build the model by using Embedded Coder.

```
evalc('slbuild(model)');
```

This code is in the `InternalZeroInitialization.c` file. The generated code does not initialize internal data by assignment to zero.

```
rtwdemodbtype(cfile,'/* Model initialize', '* File trailer', 1, 0);
```

```
/* Model initialize function */
void InternalZeroInitialization_initialize(void)
{
  /* (no initialization code required) */
}

/*
```

```
bdclose(model)
rtwdemoclean;
```

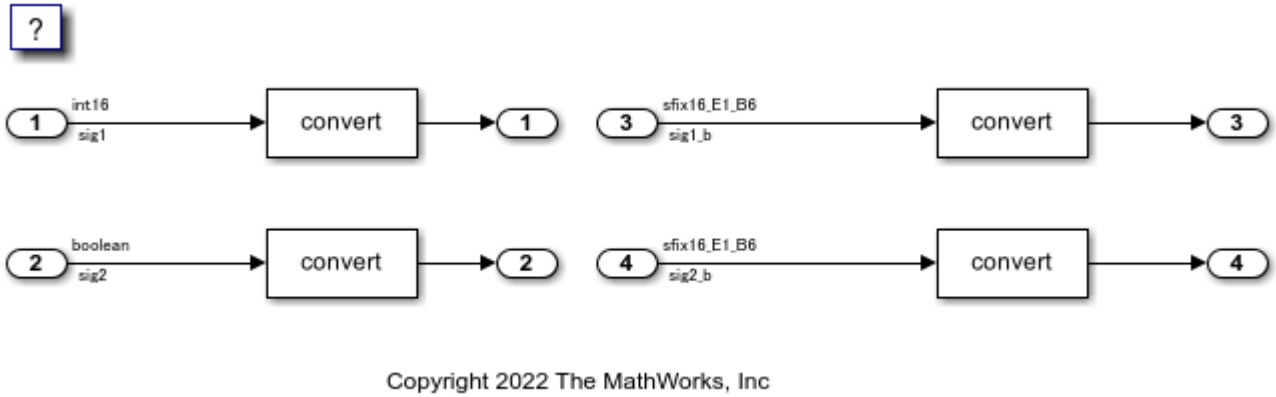
Remove Initialization Code from Root-Level Inports and Outports Set to Zero

This example shows how to remove initialization code from root-level inports and outports set to zero.

Example Model

In the `RootZeroInitialization` model, all of the input and output signals have a numeric value of zero. Because signals `sig1` and `sig2` have data types `int16` and `Boolean`, respectively, and all of the output signals have data type `double`, these signals also have initial values of bitwise zero. The signals have an integer bit pattern of `0`, meaning that all bits are off. Signals `sig1_b` and `sig2_b` have a fixed-point data type with bias, so their initial value is not bitwise zero.

```
model = 'RootZeroInitialization';
open_system(model);
```



Generate Code

```
% Build the model by using Embedded Coder.
set_param(model, 'ZeroExternalMemoryAtStartup','on');
evalc('slbuild(model)');
```

These lines of `RootZeroInitialization.c` code show the initialization of root-level inports and outports without the optimization. The four input signals are individually initialized as global variables. The four output signals are members of a global structure that the `memset` function initializes to bitwise zero.

```
cfile = fullfile('RootZeroInitialization_ert_rtw',...
    'RootZeroInitialization.c');
rtwdemodbtype(cfile, '/* Model initialize function */',...
    'trailer for generated code', 1, 0);
```

```
/* Model initialize function */
void initialize(void)
{
  /* Registration code */

  /* external inputs */
  sig1 = 0;
  sig2 = false;
  sig1_b = -3;
  sig2_b = -3;

  /* external outputs */
  (void)memset(&rtY, 0, sizeof(ExternalOutputs));
}

/*
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Remove root level I/O zero initialization**.

Alternatively, use the command-line API to enable the optimization:

```
 set_param(model, 'ZeroExternalMemoryAtStartup','off');
```

Generate Code with Optimization

The optimized code does not contain initialization code for the input signals `sig1`, `sig2`, and the four output signals because their initial values are bitwise zero.

Build the model.

```
evalc('slbuild(model)');
```

Here is the `RootZeroInitialization.c` optimized code in the initialization function.

```
cfile = fullfile('RootZeroInitialization_ert_rtw',...
    'RootZeroInitialization.c');
rtwdemodbtype(cfile, '/* Model initialize function */',...
    'trailer for generated code', 1, 0);
```

```
/* Model initialize function */
void initialize(void)
{
  /* Registration code */

  /* external inputs */
  sig1_b = -3;
  sig2_b = -3;
}

/*
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
```

Additional Information

- You can use the **Use memset to initialize floats and doubles** parameter to control the representation of zero during initialization. See Use memset to initialize floats and doubles to 0.0.
- The code does not initialize zero values whose storage classes have the `Imported` scope, regardless of the settings of the **Remove internal data zero initialization** and **Remove root level I/O zero initialization** parameters.
- If you specify values (zero or nonzero) in an Initialize Function block, the code initializes those values, regardless of the settings of the **Remove internal data zero initialization** and **Remove root level I/O zero initialization** parameters.
- If you connect an inport to an Initialize Function block, the code does not initialize a value for that inport, regardless of the setting of the **Remove root level I/O zero initialization** parameter.

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Optimize Generated Code Using memset Function” on page 76-95

Simplify Multiply Operations in Array Indexing

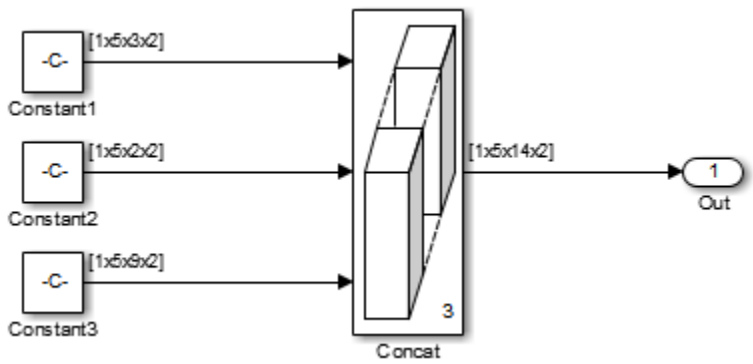
In this section...

"Example Model" on page 79-9
"Generate Code" on page 79-9
"Generate Code with Optimization" on page 79-10

The generated code might have multiply operations when indexing an element of an array. You can select the optimization parameter Simplify array indexing to replace multiply operations in the array index with a temporary variable. This optimization can improve execution speed by reducing the number of times the multiply operation executes.

Example Model

If you have the following model:



The Constant blocks have the following **Constant value**:

- `Const1`: `reshape(1:30,[1 5 3 2])`
- `Const2`: `reshape(1:20,[1 5 2 2])`
- `Const3`: `reshape(1:90,[1 5 9 2])`

The Concatenate block parameter **Mode** is set to `Multidimensional array`. The Constant blocks **Sample time** parameter is set to `-1`.

Generate Code

Building the model with the **Simplify array indexing** parameter turned off generates the following code:

```
int32_T i;
int32_T i_0;
int32_T i_1;

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 3; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * i_1) + 70 * i] =
```

```
        ex_arrayindex_P.Constant1_Value[(5 * i_1 + i_0) + 15 * i];
    }
  }
}

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 2; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 3)) + 70 * i] =
        ex_arrayindex_P.Constant2_Value[(5 * i_1 + i_0) + 10 * i];
    }
  }
}

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 9; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 5)) + 70 * i] =
        ex_arrayindex_P.Constant3_Value[(5 * i_1 + i_0) + 45 * i];
    }
  }
}
```

Generate Code with Optimization

Open the Configuration Parameters dialog box and select the **Simplify array indexing** parameter. Build the model again. In the generated code, `[(i_0 + tmp_1) + tmp]` replaces a multiply operation in the array index, `[(i_0 + 5 * i_1) + 70 * i]`. The generated code is now:

```
int32_T i;
int32_T i_0;
int32_T i_1;
int32_T tmp;
int32_T tmp_0;
int32_T tmp_1;

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
  for (i_1 = 0; i_1 < 3; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + tmp_1) + tmp] =
        ex_arrayindex_P.Constant1_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 15;
}

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
```

```
  for (i_1 = 0; i_1 < 2; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 15] =
        ex_arrayindex_P.Constant2_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 10;
}

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
  for (i_1 = 0; i_1 < 9; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 25] =
        ex_arrayindex_P.Constant3_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 45;
}
```

See Also

Simplify array indexing

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 76-48
- “Vector Operation Optimization” on page 76-99

Replace boolean with Specific Integer Data Type

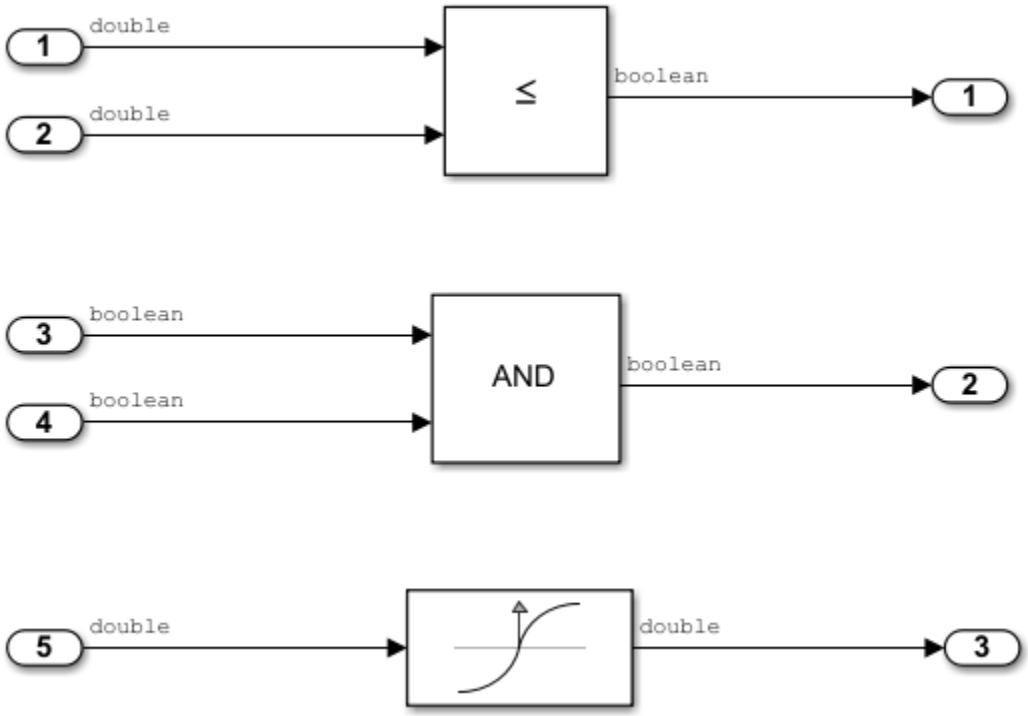
Depending on the architecture of the processor that your production hardware uses, you can improve the execution speed of the generated code. Select a specific integer data type for the built-in type `boolean`. Using data type replacement, in the generated code you can replace the `boolean` built-in data type with one of these integer types:

- `int8`
- `uint8`
- `int`*n*

To match the integer word size for the production hardware, replace *n* with 8, 16, or 32 .

Example Model

The model `ex_bool` contains two blocks that output `boolean` values and two blocks that take `boolean` values as inputs. This example shows how to replace the data type `boolean` with the integer data type `int32` in the code generated for a 32-bit hardware target.



Generate Code That Contains the Default boolean Data Type

View the generated file `rtwtypes.h`. The `typdef` statements contain the generic type definition of `boolean_T`, which is the code generation name for `boolean`.

```
/*=============================================================================*
 * Generic type definitions: boolean_T, char_T, byte_T, int_T, uint_T,       *
```

```
 *                           real_T, time_T, ulong_T.                        *
 *=============================================================================*/
typedef double real_T;
typedef double time_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned int uint_T;
typedef unsigned long ulong_T;
typedef char char_T;
typedef unsigned char uchar_T;
typedef char_T byte_T;
```

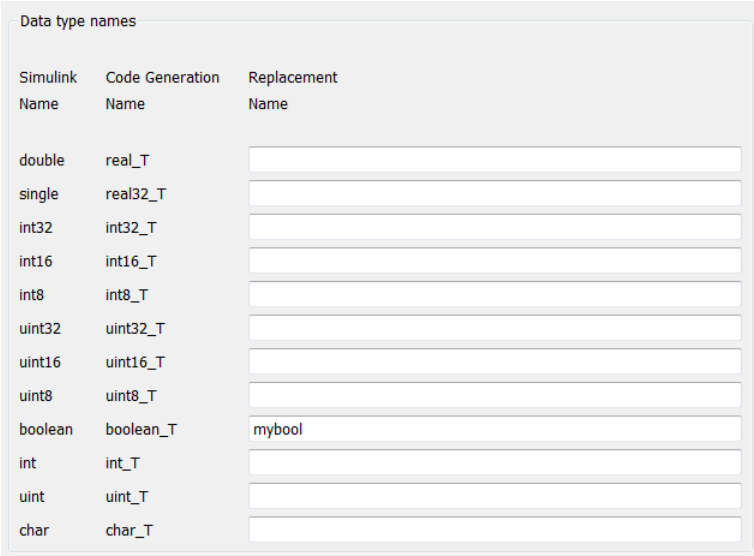
View the generated file `ex_bool.c`. The code declares `boolean` variables by using the type `boolean_T`.

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1;                          /* '<Root>/In1' */
  real_T In2;                          /* '<Root>/In2' */
  boolean_T In3;                       /* '<Root>/In3' */
  boolean_T In4;                       /* '<Root>/In4' */
  real_T In5;                          /* '<Root>/In5' */
} ExtU_ex_bool_T;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  boolean_T Out1;                      /* '<Root>/Out1' */
  boolean_T Out2;                      /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
} ExtY_ex_bool_T;
```

Generate Code That Contains the Target boolean Data Type

1. Define a `Simulink.AliasType` object with a base type of `int32`. Name the object using the replacement name that you want to appear in the generated code.

   ```
   mybool = Simulink.AliasType;
   mybool.BaseType = 'int32';
   ```
2. In the Configuration Parameters dialog box, specify the **Replacement Name** field for the data type `boolean` as `mybool`.



View the generated file `rtwtypes.h`. The code maps the identifier `mybool` to the native integer type of the target hardware by creating `typedef` statements.

```
/* Generic type definitions ... */
...
typedef int boolean_T;
 ...
/* Define Simulink Coder replacement data types. */
typedef boolean_T mybool;    /* User defined replacement datatype for boolean_T */
```

View the generated file *`ex_bool`*`.c`. The code declares `boolean` variables that use the type `mybool`.

```
  /* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1;                          /* '<Root>/In1' */
  real_T In2;                          /* '<Root>/In2' */
  mybool In3;                          /* '<Root>/In3' */
  mybool In4;                          /* '<Root>/In4' */
  real_T In5;                          /* '<Root>/In5' */
} ExtU_ex_bool_T;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  mybool Out1;                         /* '<Root>/Out1' */
  mybool Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
} ExtY_ex_bool_T;
```

See Also

`Simulink.AliasType`

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement”

Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow

Optimize generated code by removing code that protects parameter expressions from integer overflow for integers and fixed-point data. If you are sure that you will not tune the terms of tunable parameter expressions to cause them to evaluate to a value that is below or above the representable value of a data type, enable this optimization. This optimization increases execution speed and results in smaller code that reduces ROM consumption.

Risks

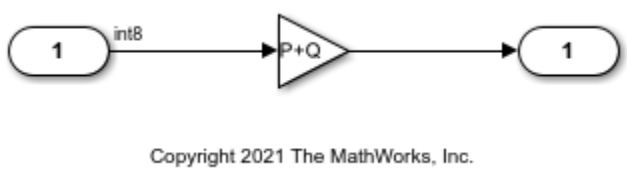
When you select the `EfficientTunableParamExpr` parameter, and you tune the terms of the expression so that it encounters an integer overflow, the result will wrap around in the generated code. This behavior can cause the application to behave unexpectedly at run-time.

NOTE: If you enable this optimization, it is possible that simulation results and results from generated code are not in bit-for-bit agreement. This example requires Embedded Coder®.

Example Model

In the model matlab:mEfficientParamExpr, an input signal of type `int8` feeds into a Gain block.

```
model = 'mEfficientParamExpr';
open_system(model);
```



Generate Code

In your system temporary folder, create another temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model, 'EfficientTunableParamExpr', 'off');
evalc('slbuild(model)');
```

View the generated code without the optimization. Here is a portion of `mEfficientParamExpr.c`.

```
cfile = fullfile(cgDir,'mEfficientParamExpr_ert_rtw','mEfficientParamExpr.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

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

```
  int32_T tmp;

  /* Gain: '<Root>/Gain' */
  tmp = P + Q;
  if (tmp > 127) {
    tmp = 127;
  } else if (tmp < -128) {
    tmp = -128;
  }

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 = (int8_T)(tmp * rtU.In1);
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Remove code from tunable parameter expressions that saturates out-of-range values**.

Alternatively, use the command-line API to enable the optimization:

```
set_param(model, 'EfficientTunableParamExpr', 'on');
```

Generate Code with Optimization

The optimized code does not contain code that checks the bounds of the representable value of a data type.

Build the model.

```
evalc('slbuild(model)');
```

The following is a portion of `mEfficientParamExpr.c`. The code that protects against expression overflow is not in the generated code.

```
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void mEfficientParamExpr_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 = (int8_T)((int8_T)(P + Q) * rtU.In1);
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Related Examples

- “Check configuration parameters for generation of inefficient saturation code”
- Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values
- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data

Optimize generated code by removing code that protects against division by zero and overflows in division `INT_MIN/-1` operations for integers and fixed-point data. If you are sure that these arithmetic exceptions do not occur during program execution, enable this optimization but it may lead to a quotient that cannot be represented.

This optimization:

- Increases execution speed.
- Results in smaller code thereby reducing ROM consumption.

Risks

When you select the `NoFixptDivByZeroProtection` parameter, the code generator removes code that protects against the following errors:

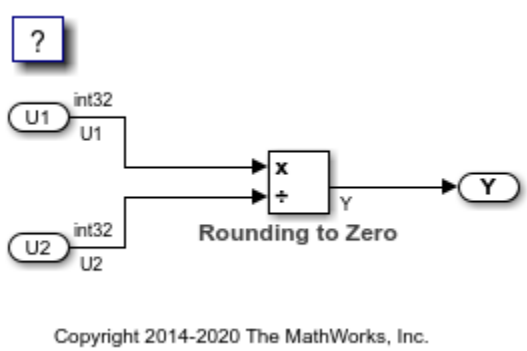
- When you divide by zero it is undefined and results in a runtime error in the generated code.
- When you divide the minimum representable value of a signed integer by negative one, the ideal result is equal to the maximum representable value plus one (INT_MAX + 1), which is not representable. This exception may cause the application to unexpectedly halt or crash at run-time.

NOTE: If you enable this optimization, it is possible that simulation results and results from generated code are not in bit-for-bit agreement. This example requires an Embedded Coder® license.

Example Model

In the model rtwdemo_nzcheck, two signals of type `int8` feed into a divide block.

```
model = 'rtwdemo_nzcheck';
open_system(model);
```



Generate Code

In your system's temporary folder, create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model, 'NoFixptDivByZeroProtection', 'off');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_nzcheck
### Successful completion of code generation for: rtwdemo_nzcheck

Build Summary

Top model targets built:

Model            Action           Rebuild Reason
========================================================================================
rtwdemo_nzcheck  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.674s
```

View the generated code without the optimization. Here is a portion of `rtwdemo_nzcheck.c`.

```
cfile = fullfile(cgDir,'rtwdemo_nzcheck_ert_rtw','rtwdemo_nzcheck.c');
rtwdemodbtype(cfile,'/* Real-time model','/* Model initialize function',1, 1);
```

```
/* Real-time model */
static RT_MODEL_rtwdemo_nzcheck rtwdemo_nzcheck_M_;
RT_MODEL_rtwdemo_nzcheck *const rtwdemo_nzcheck_M = &rtwdemo_nzcheck_M_;
int32_T div_s32(int32_T numerator, int32_T denominator)
{
  int32_T quotient;
  uint32_T tempAbsQuotient;
  if (denominator == 0) {
    quotient = numerator >= 0 ? MAX_int32_T : MIN_int32_T;

    /* Divide by zero handler */
  } else {
    tempAbsQuotient = (numerator < 0 ? ~(uint32_T)numerator + 1U : (uint32_T)
                       numerator) / (denominator < 0 ? ~(uint32_T)denominator +
      1U : (uint32_T)denominator);
    quotient = (numerator < 0) != (denominator < 0) ? -(int32_T)tempAbsQuotient :
      (int32_T)tempAbsQuotient;
  }

  return quotient;
}

/* Model step function */
void rtwdemo_nzcheck_step(void)
{
  /* Product: '<Root>/Divide' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  Y = div_s32(U1, U2);
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Remove code that protects against division arithmetic exceptions**.

Alternatively, you may use the command-line API to enable the optimization:

```
set_param(model, 'NoFixptDivByZeroProtection', 'on');
```

Generate Code with Optimization

The optimized code does not contain code that checks for whether or not the divisor has a value of zero.

Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_nzcheck
### Successful completion of code generation for: rtwdemo_nzcheck

Build Summary

Top model targets built:

Model            Action           Rebuild Reason
========================================================================
rtwdemo_nzcheck  Code generated.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.385s
```

The following is a portion of `rtwdemo_nzcheck.c`. The code that protects against division arithmetic exceptions is not in the generated code.

```
rtwdemodbtype(cfile,'/* Real-time model','/* Model initialize function',1, 1);
```

```
/* Real-time model */
static RT_MODEL_rtwdemo_nzcheck rtwdemo_nzcheck_M_;
RT_MODEL_rtwdemo_nzcheck *const rtwdemo_nzcheck_M = &rtwdemo_nzcheck_M_;

/* Model step function */
void rtwdemo_nzcheck_step(void)
{
  /* Product: '<Root>/Divide' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  Y = U1 / U2;
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Additional Information

There are several other factors that can affect the appearance of the generated code for division operations. The generated code for blocks containing MATLAB® code with integer or fixed-point division operations differs from the built-in Divide block in Simulink®. To balance the efficiency and semantics of fixed-point and integer divisions in these blocks, use `fi` objects and set the `fimath` properties to fit your needs and requires a Fixed-Point Designer™ license. Rounding and overflow modes also affect the size and efficiency of the generated code.

See Also

Remove code that protects against division arithmetic exceptions

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Remove Code for Out-of-Range Floating Point to Integer Conversions” on page 76-20
- “Remove Code That Maps NaN to Integer Zero” on page 76-23

Optimize Generated Code by Consolidating Redundant If-Else Statements

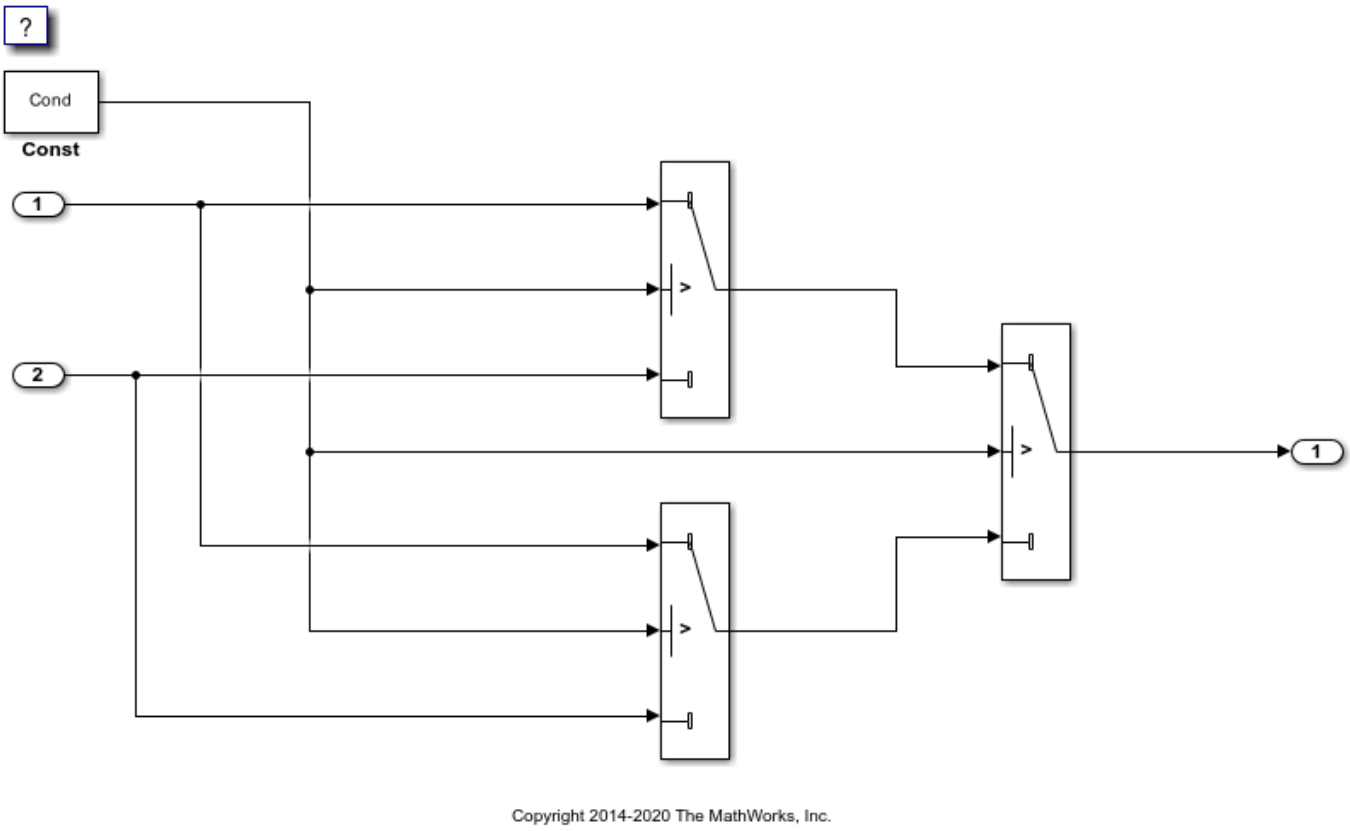
This example shows how to optimize generated code by combining `if-else` statements that share the same condition. This optimization:

- Improves control flow.
- Reduces code size.
- Reduces RAM consumption.
- Increases execution speed.

Example

The model rtwdemo_controlflow_opt contains three Switch blocks. The Constant block provides the control input to the Switch blocks. The variable named `Cond` determines the value of the Constant block.

```
model = 'rtwdemo_controlflow_opt';
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_controlflow_opt
### Successful completion of build procedure for: rtwdemo_controlflow_opt

Build Summary

Top model targets built:

Model                     Action                        Rebuild Reason
========================================================================================================
rtwdemo_controlflow_opt  Code generated and compiled.  Code generation information file does not

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.99s
```

These lines of `rtwdemo_controlflow_opt.c` code show that in the generated code, two `if-else` statements and one `else-if` statement represent the three Switch blocks.

```
cfile = fullfile(cgDir,'rtwdemo_controlflow_opt_ert_rtw',...
    'rtwdemo_controlflow_opt.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_controlflow_opt_step(void)
{
  /* Switch: '<Root>/Switch3' incorporates:
   *  Constant: '<Root>/Const'
   *  Switch: '<Root>/Switch2'
   */
  if (Cond) {
    /* Switch: '<Root>/Switch1' */
    if (Cond) {
      /* Outport: '<Root>/Out1' incorporates:
       *  Inport: '<Root>/In1'
       */
      rtY.Out1 = rtU.In1;
    } else {
      /* Outport: '<Root>/Out1' incorporates:
       *  Inport: '<Root>/In2'
       */
      rtY.Out1 = rtU.In2;
    }

    /* End of Switch: '<Root>/Switch1' */
  } else if (Cond) {
    /* Switch: '<Root>/Switch2' incorporates:
     *  Inport: '<Root>/In1'
     *  Outport: '<Root>/Out1'
     */
    rtY.Out1 = rtU.In1;
  } else {
```

```
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In2'
     */
    rtY.Out1 = rtU.In2;
  }

  /* End of Switch: '<Root>/Switch3' */
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Code generation-> Code Style** pane, clear **Preserve condition expression in if statement**. This parameter is on by default.

Alternatively, use the command-line API to turn off the parameter:

```
set_param(model, 'PreserveIfCondition', 'off');
```

Generate Code with Optimization

In the optimized code, the code generator consolidates the two `if-else` statements and one `else-if` statement into one `if-else` statement. The code generator consolidates these statements because they all share the same condition. There is no intervening code that affects the outcomes of these statements.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_controlflow_opt
### Successful completion of build procedure for: rtwdemo_controlflow_opt

Build Summary

Top model targets built:

Model                     Action                        Rebuild Reason
========================================================================================
rtwdemo_controlflow_opt  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.81s
```

Here is the `rtwdemo_controlflow_opt.c` optimized code.

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_controlflow_opt_step(void)
{
  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/Const'
   *  Switch: '<Root>/Switch3'
   */
  if (Cond) {
    /* Outport: '<Root>/Out1' incorporates:
```

```
     *  Inport: '<Root>/In1'
     */
    rtY.Out1 = rtU.In1;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In2'
     */
    rtY.Out1 = rtU.In2;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Preserve condition expression in if statement

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Eliminate Dead Code Paths in Generated Code” on page 76-56

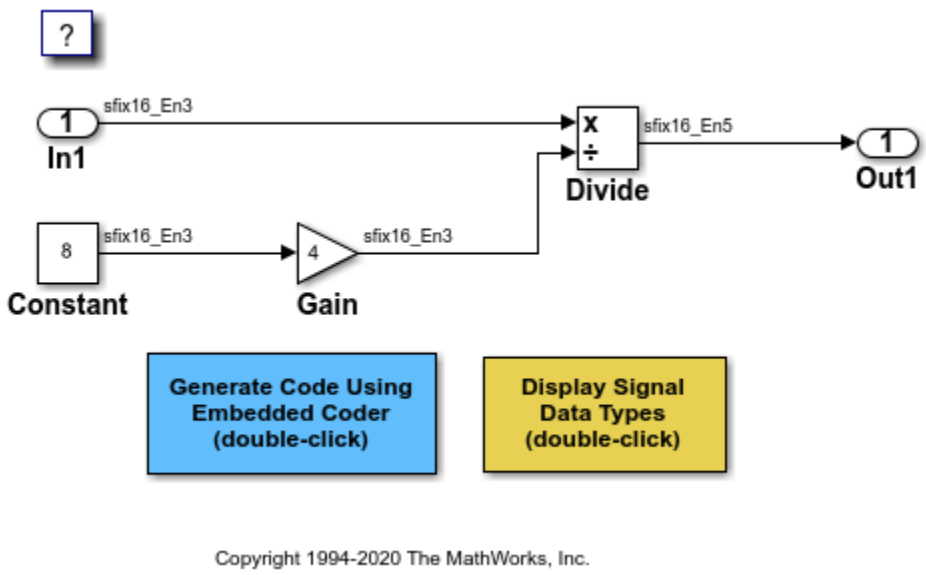
Optimize Generated Code for Fixed-Point Data Operations

This example shows how the code generator optimizes fixed-point operations by replacing expensive division operations with highly efficient product operations. This optimization improves execution speed.

Example Model

In the model rtwdemo_fixptdiv, two fixed point signals connect to a Divide block. The **Number of inputs** parameter has the value `/*`.

```
model='rtwdemo_fixptdiv';
load_system(model)
set_param(model,'HideAutomaticNames','off','SimulationCommand','Update')
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model,'GenCodeOnly','on');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_fixptdiv
### Successful completion of code generation for: rtwdemo_fixptdiv

Build Summary

Top model targets built:
```

```
Model              Action           Rebuild Reason
========================================================================================
rtwdemo_fixptdiv  Code generated.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.273s
```

View the generated code. Here is a portion of `rtwdemo_fixptdiv.c`.

```
cfile = fullfile(cgDir,'rtwdemo_fixptdiv_ert_rtw','rtwdemo_fixptdiv.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_fixptdiv_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Product: '<Root>/Divide'
   */
  rtY.Out1 = (int16_T)(rtU.In1 >> 3);
}
```

The generated code contains a highly efficient right shift operation instead of an expensive division operation. The generated code also contains the precomputed value for the constant input to the Product block.

Note that the resulting operation also includes the adjustment in signal scaling from 2^-3 to 2^-5.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Fixed Point”

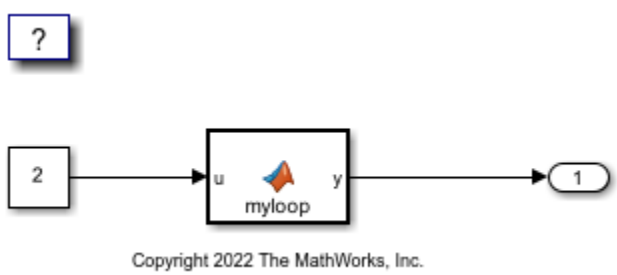
Unroll Parallel for-Loop That Has Small Number of Iterations

This example shows how to use the model configuration parameter Loop unrolling threshold to generate optimized code by unrolling parallel `for`-loops (`parfor`-loops). For a small number of loop iterations that perform some simple calculation, parallelization is inefficient as it introduces overhead, which includes time taken for thread creation, data synchronization between threads, and thread deletion. By unrollling small parallel `for`-loops, you can achieve improved execution speed. Unrolling loops that have a large number of iterations can significantly increase code generation time and generate inefficient code.

Example Model

Open the example model `rtwdemo_unroll_parfor`. The model is configured to generate multithreaded code by selecting the **Generate parallel for loops** parameter.

```
model = 'rtwdemo_unroll_parfor';
open_system(model)
```



The MATLAB Function block contains this code:

```
type('myloop.m')
```

```
function y = myloop(u)
  y = ones(1,100);
  parfor (i = 1:5)
    y(i) = i+u;
  end
end
```

The code contains a `parfor`-loop that has five iterations.

Generate Code by Using the Default Threshold Value

1. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

2. Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_unroll_parfor
### Successful completion of build procedure for: rtwdemo_unroll_parfor
```

```
Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
rtwdemo_unroll_parfor  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.984s
```

Inspect the generated `rtwdemo_unroll_parfor_step` step function in the `rtwdemo_unroll_parfor.c` .

```
file = fullfile('rtwdemo_unroll_parfor_ert_rtw','rtwdemo_unroll_parfor.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void rtwdemo_unroll_parfor_step(void)
{
  int32_T i;
  int32_T i_0;

  /* MATLAB Function: '<Root>/MATLAB Function' */
  for (i_0 = 0; i_0 < 100; i_0++) {
    rtwdemo_unroll_parfor_Y.Outport[i_0] = 1.0;
  }

#pragma omp parallel for num_threads(omp_get_max_threads())

  for (i = 0; i < 5; i++) {
    rtwdemo_unroll_parfor_Y.Outport[i] = ((real_T)i + 1.0) + 2.0;
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

The code generator generates a parallel `for`-loop for the MATLAB Function block because it has 5 iterations which is equal to the default threshold value.

Generate Code by Raising the Threshold Value

1. Open the Configuration Parameters dialog box. On the **Optimization** pane, set the **Loop unrolling threshold** to `6` . Alternatively, use the command-line API:

```
set_param(model, 'RollThreshold','6')
```

Build the model. Inspect the generated `rtwdemo_unroll_parfor_step` step function in the `rtwdemo_unroll_parfor.c`.

```
slbuild(model);
file = fullfile('rtwdemo_unroll_parfor_ert_rtw','rtwdemo_unroll_parfor.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
### Starting build procedure for: rtwdemo_unroll_parfor
### Successful completion of build procedure for: rtwdemo_unroll_parfor
```

```
Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
rtwdemo_unroll_parfor  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.306s

/* Model step function */
void rtwdemo_unroll_parfor_step(void)
{
  int32_T i;

  /* MATLAB Function: '<Root>/MATLAB Function' */
  for (i = 0; i < 100; i++) {
    rtwdemo_unroll_parfor_Y.Outport[i] = 1.0;
  }

  rtwdemo_unroll_parfor_Y.Outport[0] = 3.0;
  rtwdemo_unroll_parfor_Y.Outport[1] = 4.0;
  rtwdemo_unroll_parfor_Y.Outport[2] = 5.0;
  rtwdemo_unroll_parfor_Y.Outport[3] = 6.0;
  rtwdemo_unroll_parfor_Y.Outport[4] = 7.0;

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

The code generator unrolls the parallel `for`-loop and produces a copy of the loop body for each iteration because it has fewer iterations compared to the threshold value.

Clean Up Example Folders and Files

Close the model and remove temporary folders and files.

```
bdclose(model);
cd(currentDir);
rtwdemoclean;
```

See Also

Related Examples

- “Configure Loop Unrolling Threshold” on page 76-46

Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink

Develop and use code replacement libraries to replace function and operators in generated code. Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware
- Integration with existing application code
- Compliance with a standard, such as AUTOSAR
- Modification of code behavior, such as enabling or disabling nonfinite or inline support
- Application- or project-specific code requirements, such as use of BLAS or elimination of `math.h`, system header files, or calls to `memcpy` or `memset`.

You can configure a model such that the code generator uses a code replacement library that MathWorks® provides. If you have an Embedded Coder® license, you can develop your own code replacement library interactively with the Code Replacement Tool or programmatically.

This example uses models to show a variety of ways you can define code replacement mappings programmatically. Each model includes buttons that you can use to

- View the code replacement library table definitions from the Code Replacement Viewer
- Open the library table definition file in the editor
- Open the library registration file in the editor
- View the model configuration
- Generate code

For more information, see “Define Code Replacement Library Optimizations” on page 72-36 and “Register Code Replacement Library” on page 72-51.

Steps for Developing a Code Replacement Library

1. Identify your code replacement requirements with respect to function or operating mappings, build information, and registration information.
2. Prepare for code replacement library development (for example, identify or develop models to test your library).
3. Define code replacement mappings.
4. Specify build information for replacement code.
5. Register code replacement mappings.
6. Verify code replacements.
7. Deploy the library.

For more information, see “Quick Start Code Replacement Library Development - Simulink” on page 72-21.

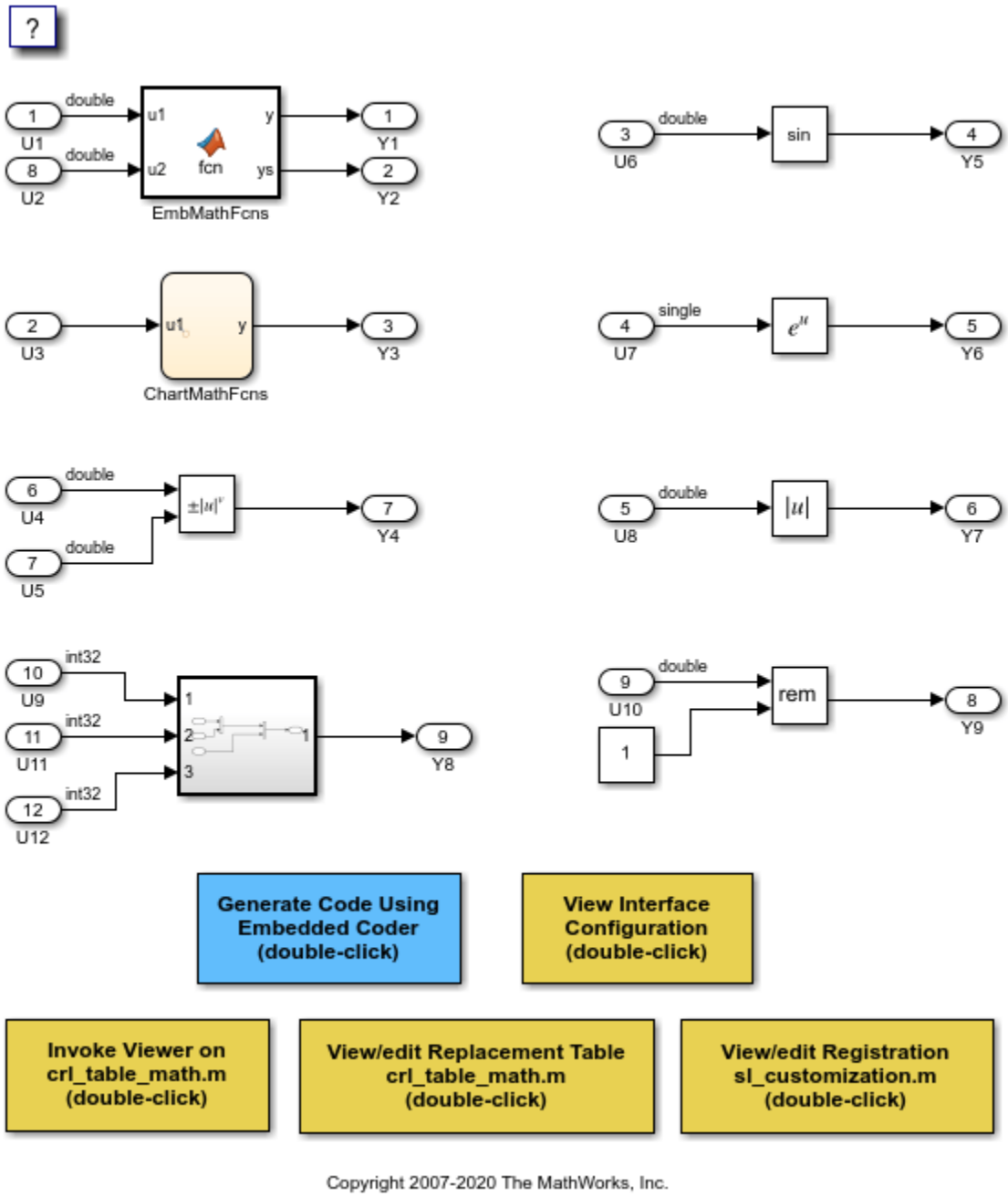
Math Function Replacement

This example defines and registers code replacement mappings for math functions. You can define code replacement mappings for a variety of functions (see “Code You Can Replace From Simulink Models” on page 72-6).

Open the model `rtwdemo_crlmath` and use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

For more information, see “Math Function Code Replacement” on page 72-64.

```
open_system('rtwdemo_crlmath')
```



Addition and Subtraction Operator Replacement

This example shows how to define and register code replacement mappings for addition (+) and subtraction (-) operations. When defining entries for addition and subtraction operations, you can specify which of the following algorithms (`EntryInfoAlgorithm`) your library functions implement:

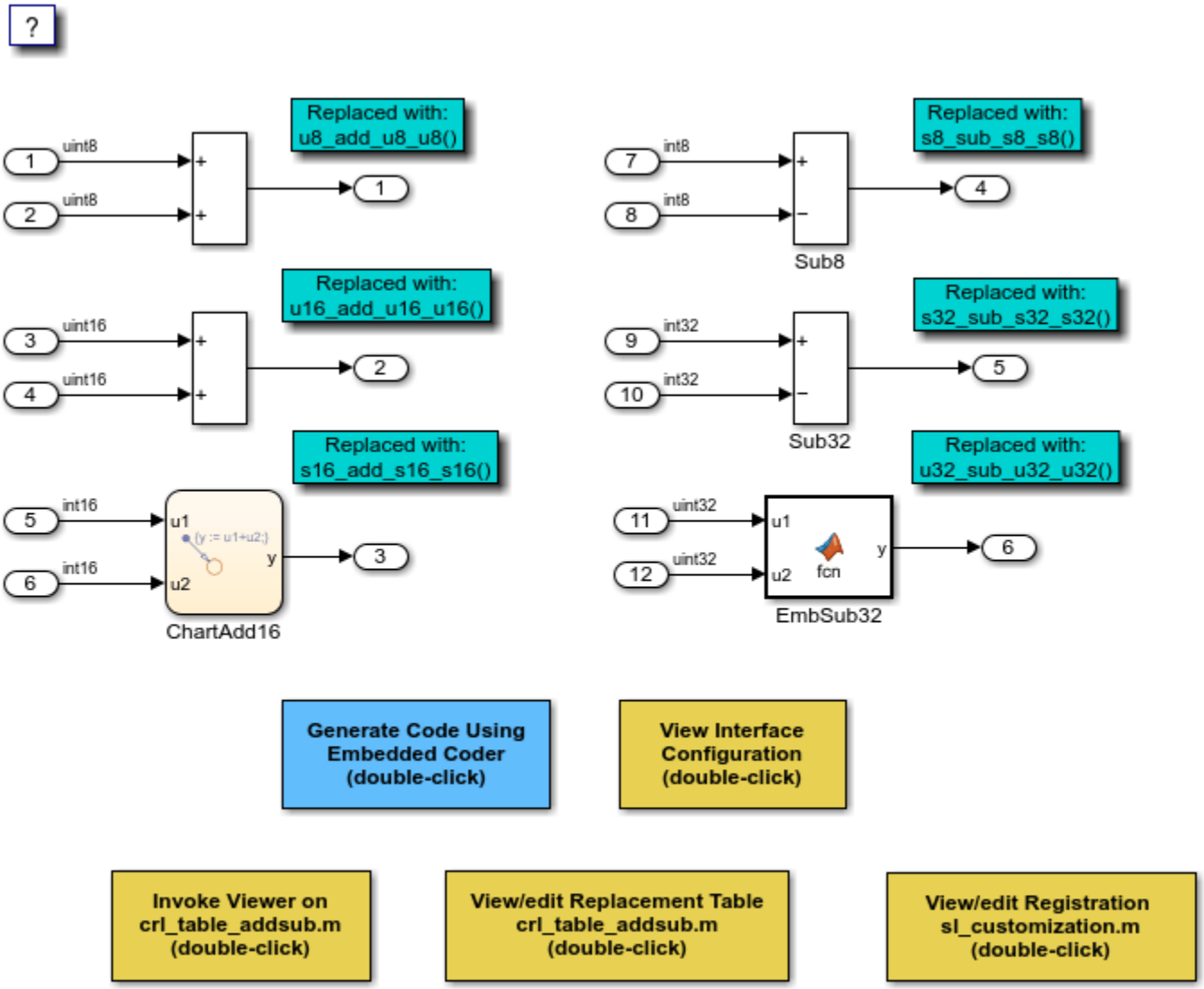
- Cast-before-operation (CBO) (`RTW_CAST_BEFORE_OP`), the default
- Cast-after-operation (CAO) (`RTW_CAST_AFTER_OP`)

1. Open the model `rtwdemo_crladdsub`. The model shows how to define and register code replacement mappings for scalar addition subtraction (-) operations on two operands with the following pairings of built-in integer data types:

- `int8`, `uint8`
- `int16`, `uint16`
- `int32`, `uint32`

CBO, the default algorithm, is assumed.

```
open_system('rtwdemo_crladdsub')
```



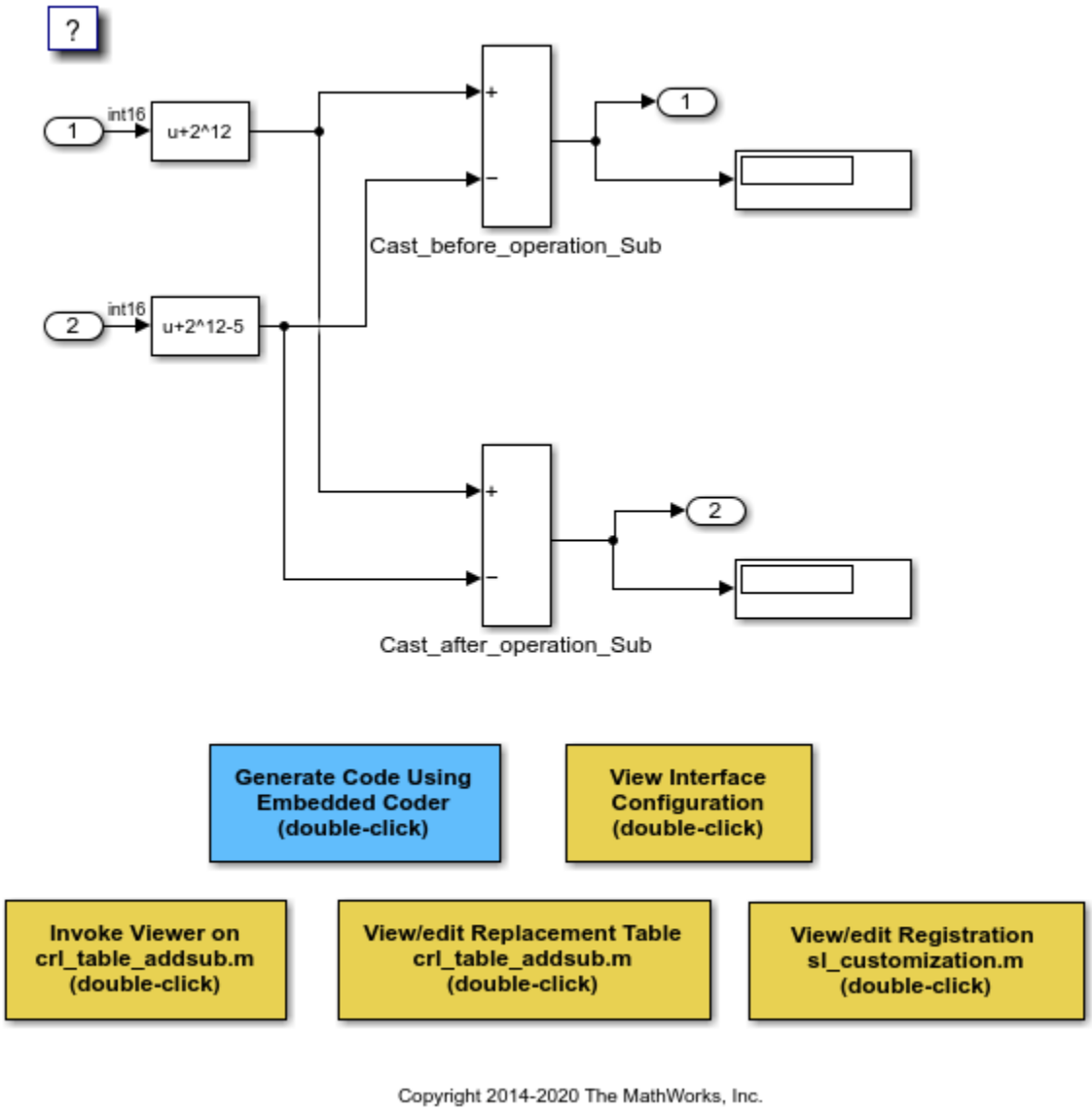
Copyright 2007-2020 The MathWorks, Inc.

2. Use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

3. Explore the differences between the CBO and CAO algorithms by opening the model `rtwdemo_crl_cbo_cao`. The model shows two SUM blocks, each demonstrating one of the algorithm settings to match a corresponding code replacement entry.

- For the `Cast_before_operation_Sub` block, the code generator calculates the subtraction operation in the accumulator data type (`int8`), which is smaller than the input data type (`int16`). This is equivalent to a CBO because the code generator converts the input to the output data type before calculating the result.
- For the `Cast_after_operation_Sub` block, the code generator calculates the subtraction operation in the accumulator data type (`int32`), which is larger than the input data type (`int16`). This is equivalent to a CAO because the code generator produces the same result by calculating the result without loss of range or precision before converting to the output data type.

```
open_system('rtwdemo_crl_cbo_cao')
```



4. Use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

For more information on addition and subtraction operator replacement, see “Replace Addition and Subtraction Operator Code” on page 72-153. For more information on the addition and subtraction algorithm (`EntryInfoAlgorithm`) options, see `setTflCOperationEntryParameters`.

Multiplication and Division Operator Replacement for Built-In Integers

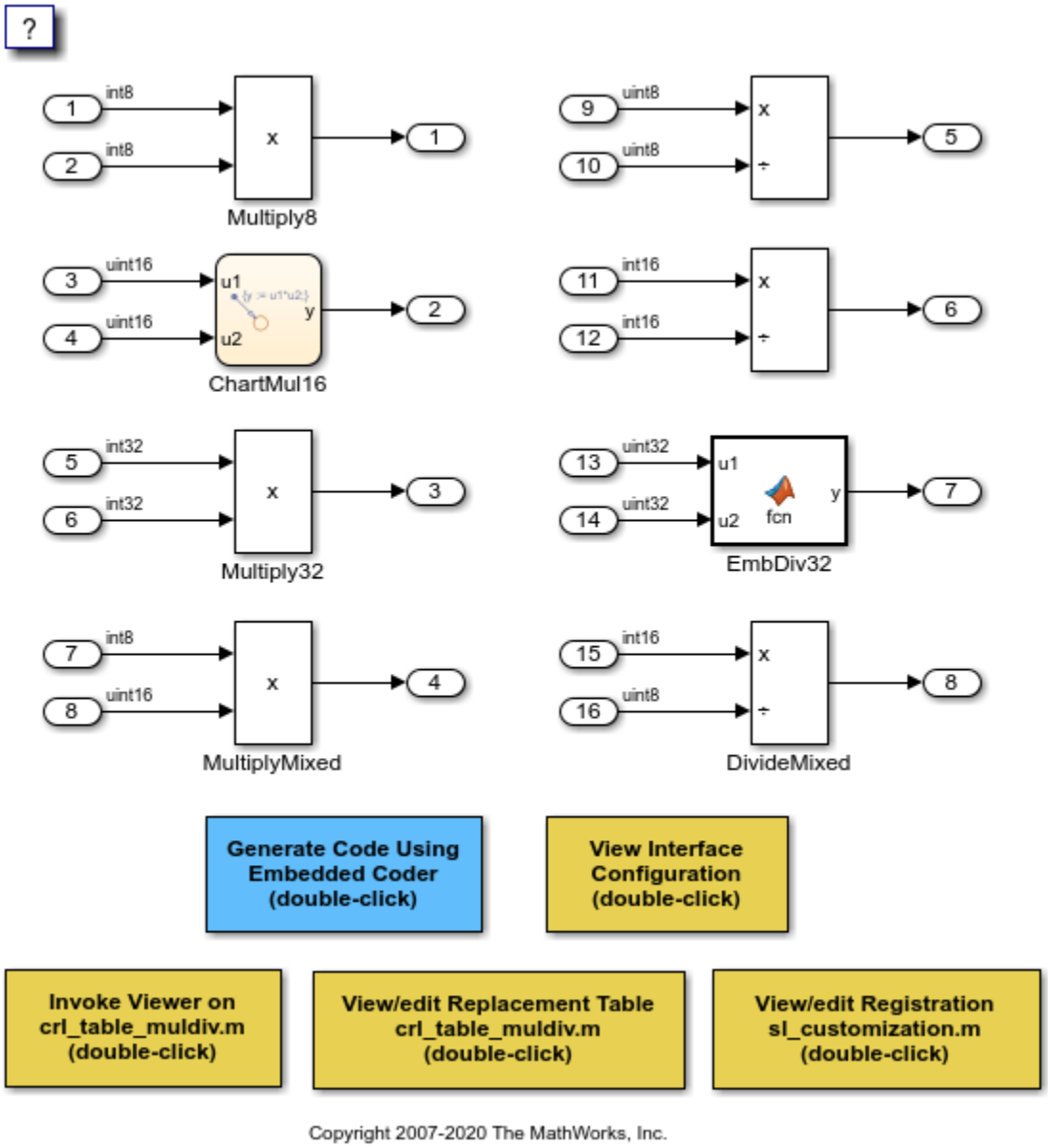
This example defines and registers code replacement mappings for scalar multiplication (*) and division (/) operations. The operations take two operands with the following pairings of built-in integer data types:

- `int8`, `uint8`

- `int16`, `uint16`
- `int32`, `uint32`

Open the model `rtwdemo_crlmuldiv` and use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

```
open_system('rtwdemo_crlmuldiv')
```



Scalar Operator Replacement

This example defines and registers code replacement mappings for scalar operations: addition, subtraction, multiplication, complex conjugate, cast, arithmetic shift right, and arithmetic shift left.

Supported types include:

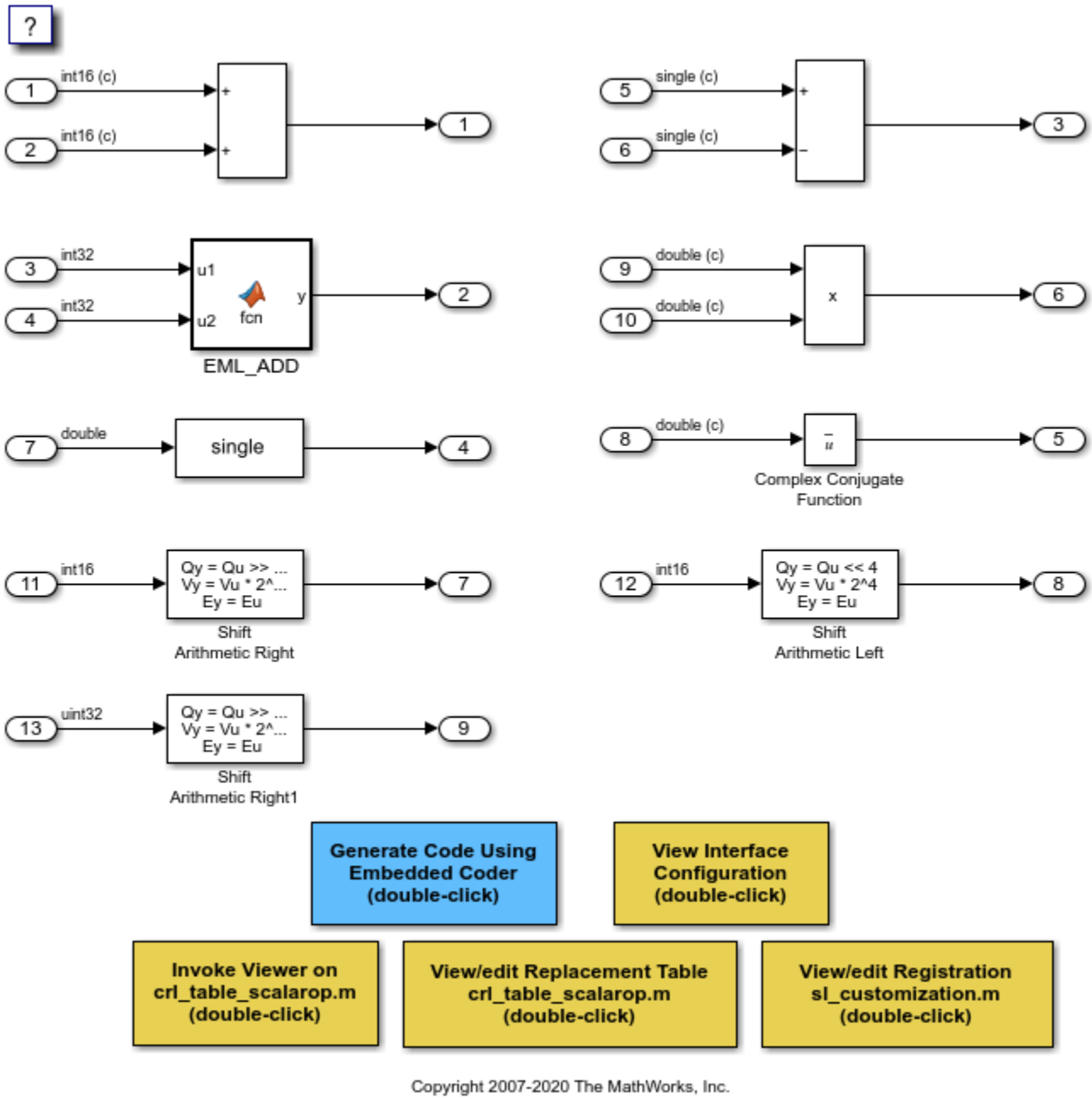
- single, double
- int8, uint8
- int16, uint16
- int32, uint32
- csingle, cdouble
- cint8, cuint8
- cint16, cuint16
- cint32, cuint32
- fixed-point integers
- mixed types (different type on each input)

Open the model `rtwdemo_crlscalarops` and use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

For more information on scalar operator replacement, “Scalar Operator Code Replacement” on page 72-145.

CBO, the default algorithm for addition and subtraction operations, is assumed.

```
open_system('rtwdemo_crlscalarops')
```



Matrix Operator Replacement

This example defines and registers code replacement mappings for matrix operations: addition, subtraction, multiplication, transposition, conjugate, and Hermitian.

Supported types include:

- `single`, `double`
- `int8`, `uint8`

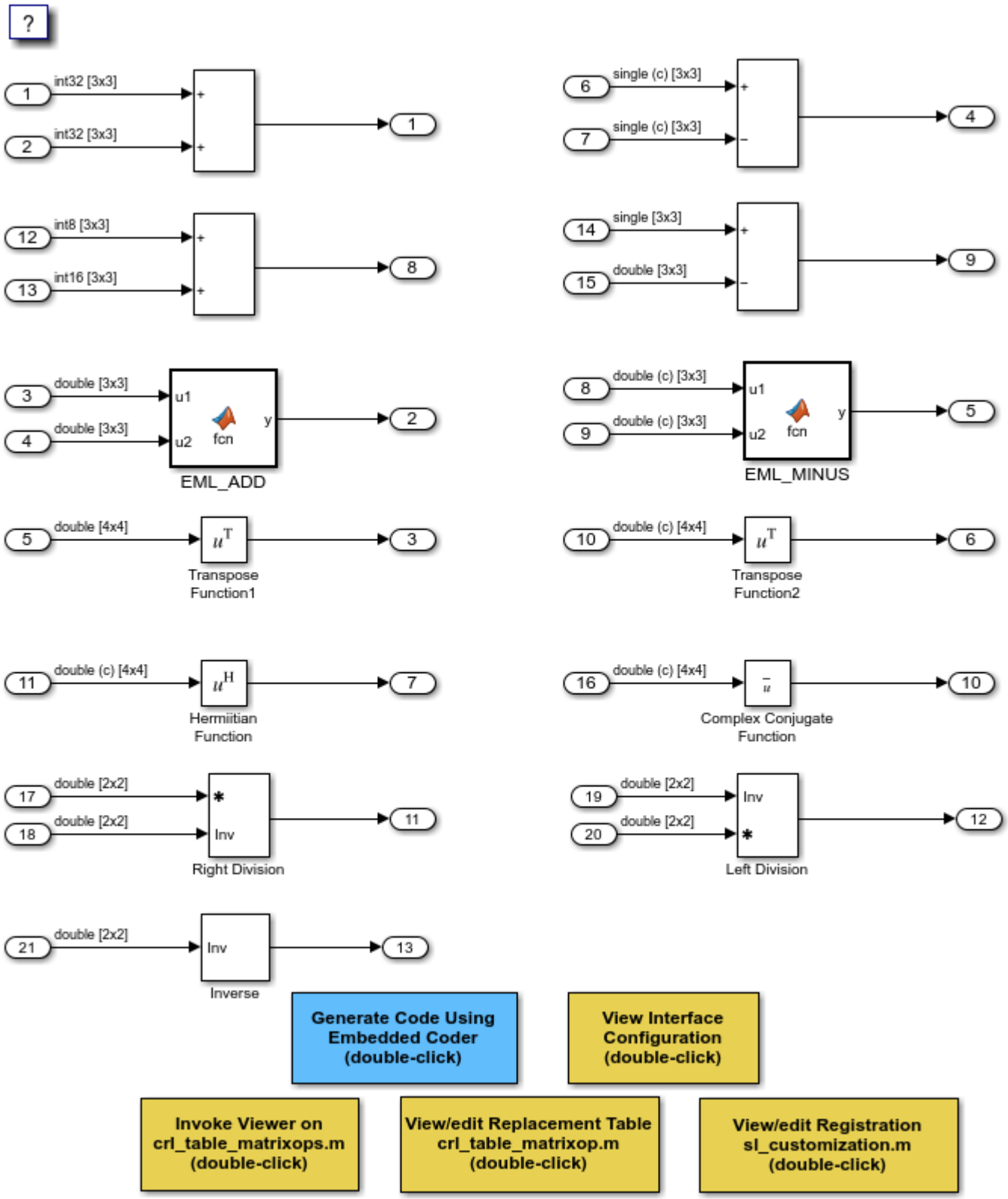
- `int16`, `uint16`
- `int32`, `uint32`
- `csingle`, `cdouble`
- `cint8`, `cuint8`
- `cint16`, `cuint16`
- `cint32`, `cuint32`
- fixed-point integers
- mixed types (different type on each input)

Open the model `rtwdemo_crlmatops`, which shows some of these replacements. Use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

For more information on matrix operator replacement, see “Small Matrix Operation to Processor Code Replacement” on page 72-177.

CBO, the default algorithm for addition and subtraction operations, is assumed.

```
open_system('rtwdemo_crlmatops')
```


?
1
int32 [3x3]
2
int32 [3x3]
1
6
single (c) [3x3]
7
single (c) [3x3]
4
12
int8 [3x3]
13
int16 [3x3]
8
14
single [3x3]
15
double [3x3]
9
3
double [3x3]
4
double [3x3]
u1
u2
fcn
y
EML_ADD
2
8
double (c) [3x3]
9
double (c) [3x3]
u1
u2
fcn
y
EML_MINUS
5
5
double [4x4]
Transpose
Function1
3
10
double (c) [4x4]
Transpose
Function2
6
11
double (c) [4x4]
Hermiitian
Function
7
16
double (c) [4x4]
Complex Conjugate
Function
10
17
double [2x2]
18
double [2x2]
Inv
Right Division
11
19
double [2x2]
20
double [2x2]
Inv
Left Division
12
21
double [2x2]
Inv
Inverse
13
Generate Code Using
Embedded Coder
(double-click)
View Interface
Configuration
(double-click)
Invoke Viewer on
crl_table_matrixops.m
(double-click)
View/edit Replacement Table
crl_table_matrixop.m
(double-click)
View/edit Registration
sl_customization.m
(double-click)
Copyright 2007-2020 The MathWorks, Inc.

Matrix Multiplication Replacement for BLAS

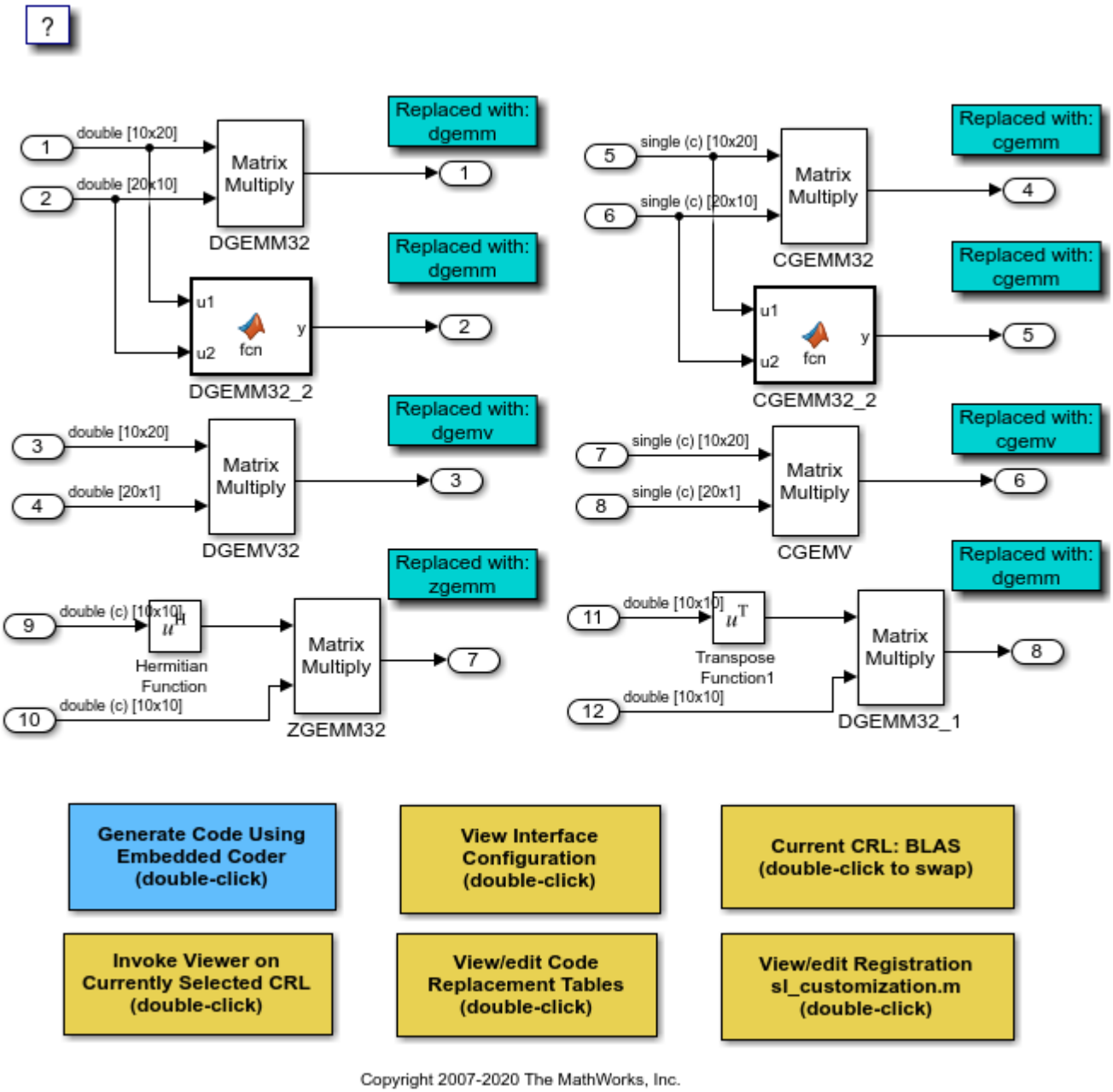
This example defines and registers code replacement mappings for Basic Linear Algebra Subroutines (BLAS) subroutines `xGEMM` and `xGEMV`. You can map the following operations to a BLAS subroutine:

- Matrix multiplication
- Matrix multiplication with transpose on single or both inputs
- Matrix multiplication with Hermitian operation on single or both inputs

Open the model `rtwdemo_crlblas` and use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

For more information on matrix multiplication replacement for BLAS, see “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 72-160.

```
open_system('rtwdemo_crlblas')
```



Fixed-Point Operator Replacement for Basic Operators

This example defines and registers code replacement mappings for scalar addition (+), subtraction (-), multiplication (*), and division (/) operations. The operations take two operands with fixed-point data types.

You can define code replacements as matching:

- Slope/bias scaling combination on the inputs and output
- Binary-point scaling combination on the inputs and output

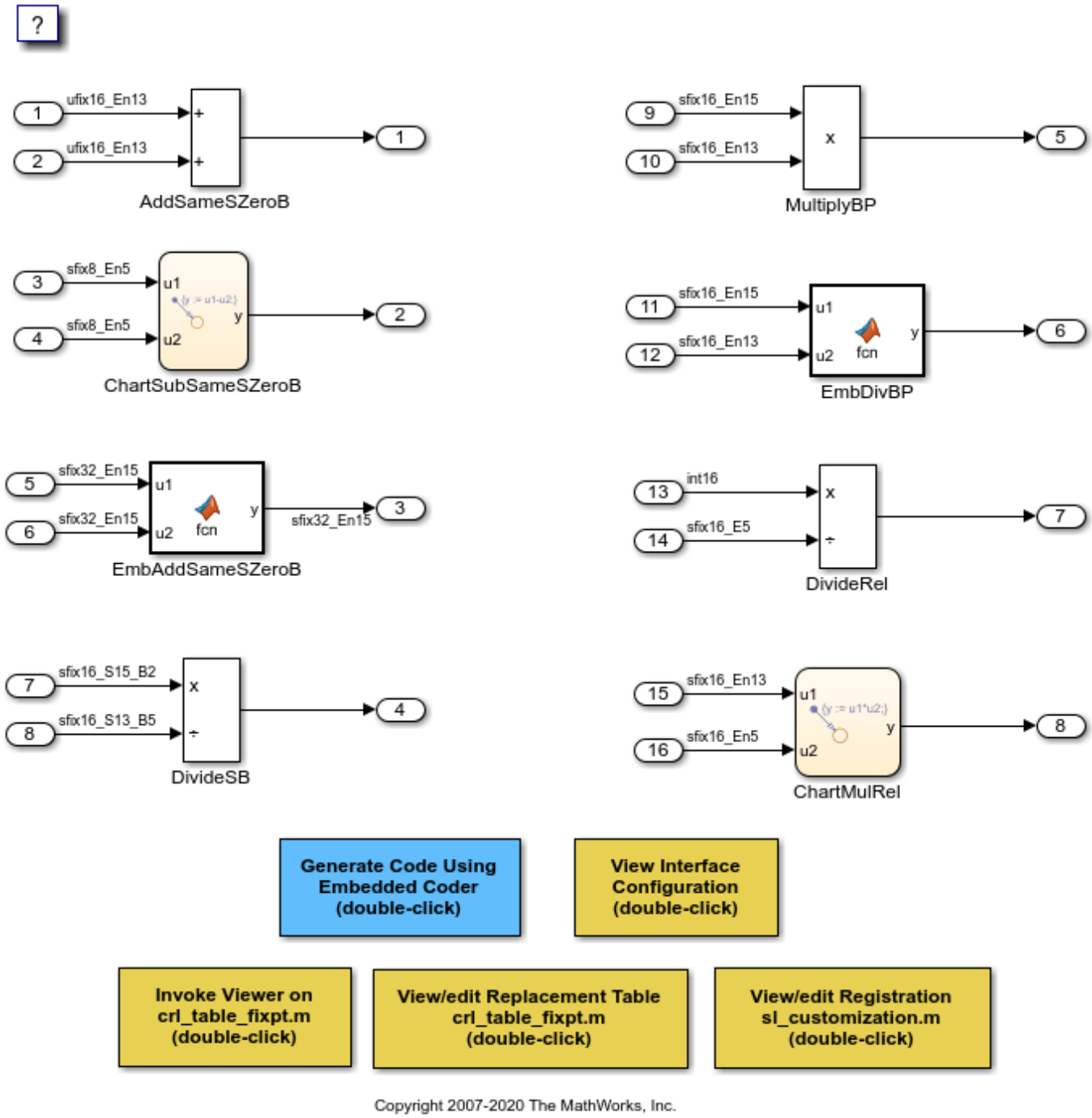
- Relative scaling between the inputs and output
- Same slope value and a zero net bias across the inputs and output

Open the model `rtwdemo_crlfixpt` and use the buttons at the bottom of the model window to explore the files that define and register the code replacement library mappings.

- By default, for addition and subtraction operator code replacements, the code generator assumes that replacement code implements a cast-before-operation (CBO) algorithm.
- Using fixed-point data types in a model requires a Fixed-Point Designer™ license.

For more information about fixed-point operator code replacement, see “Fixed-Point Operator Code Replacement” on page 72-188.

```
open_system('rtwdemo_crlfixpt')
```



Match and Replacement Process Customization for Functions

This example defines and registers code replacement mappings for custom entries. You can create your own entry by subclassing from `RTW.TflCFunctionEntryML` or `RTW.TflCOperationEntryML`. Your entry class must implement a `do_match` method that customizes your matching logic or modifies the matched entry. The `do_match` method must have a fixed preset signature.

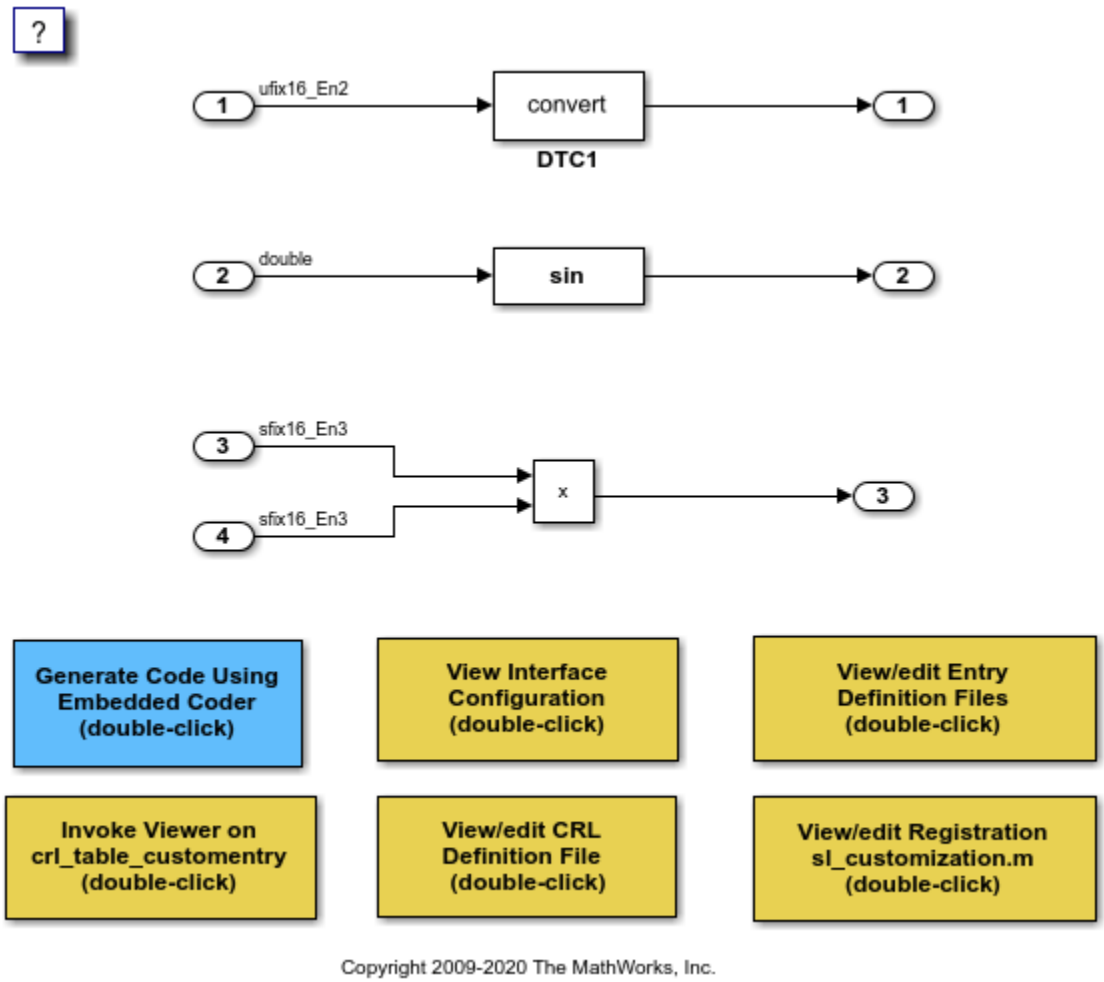
Open the model `rtwdemo_crlcustomentry`. The model shows how to modify the matched entry by injecting constants as additional implementation function arguments. `DTC1`, `Trigonometric Function`, and `Product` show custom entry code replacement for:

- Cast operation that demonstrates how to extract the fraction lengths from types and pass them into the implementation function - `Out1 = custom_cast(In1, 2, 4)`.
- Sine that demonstrates how to pass a constant value to the implementation function - `Out2=custom_sin(In2, 1)`.
- Multiplication operation that demonstrates how to compute the net slope of an operation and pass that into the implementation function - `Out3=custom_multiply_shift_right(In3,In4,3)`.

Use the buttons at the bottom of the model window to explore the files that define and register the code replacement mappings.

For more information on custom entries, see “Customize Match and Replacement Process” on page 72-132.

```
open_system('rtwdemo_crlcustomentry')
```



MATLAB Function Replacement

This example defines and registers code replacement mappings for MATLAB® functions specified in the MATLAB Function block. The function can be opted for replacement by specifying `coder.replace` within it. This feature supports replacement of MATLAB functions with the following:

- Single or multiple inputs
- Single or multiple outputs
- Scalar and matrix inputs and outputs

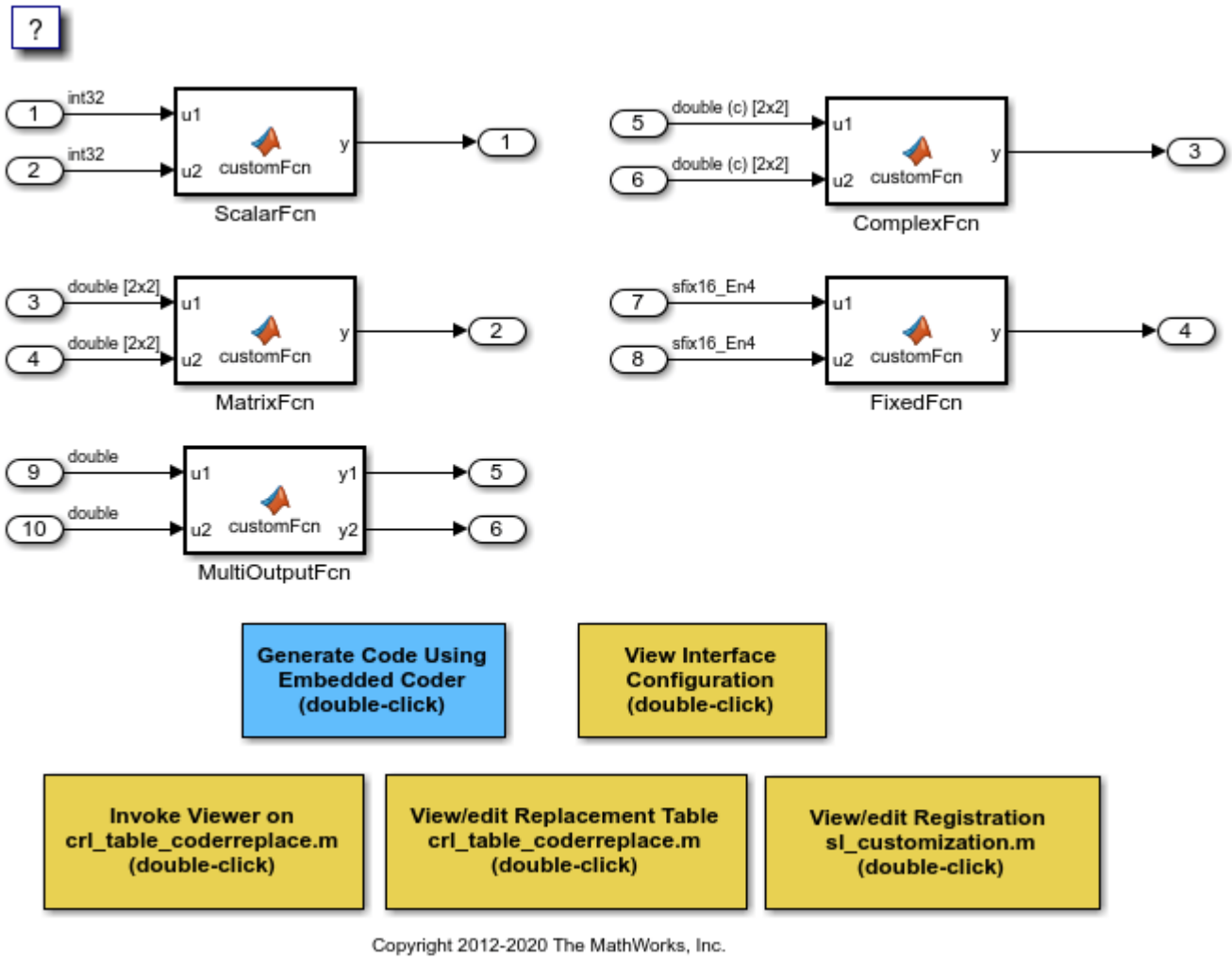
Open the model `rtwdemo_crlcoderreplace`. The model shows some of these requirements. Supported types include:

- `single`, `double`
- `int8`, `uint8`
- `int16`, `uint16`
- `int32`, `uint32`
- `csingle`, `cdouble`
- `cint8`, `cuint8`
- `cint16`, `cuint16`
- `cint32`, `cuint32`
- Fixed-point integers
- Mixed types (different type on each input)

Use the buttons at the bottom of the model window to explore the files that define and register the code replacement mappings.

For more information on MATLAB function replacement, see `coder.replace`.

```
open_system('rtwdemo_crlcoderreplace')
```



Data Alignment for Function Implementations

This example shows how to specify the alignment of matrix operands passed into a replacement function. Some target-specific function implementations require data to be aligned to optimize application performance. To configure data alignment for a function implementation:

- Specify the data alignment requirements in a table entry. You can specify alignment for implementation function arguments individually or collectively.
- Specify the data alignment capabilities and syntax for your compiler. Attach an `AlignmentSpecification` object to the `TargetCharacteristics` object of the registry entry specified in your `rtwTargetInfo.m` file.

Open the model `rtwdemo_crlalign`. The model shows three data alignment code replacement scenarios:

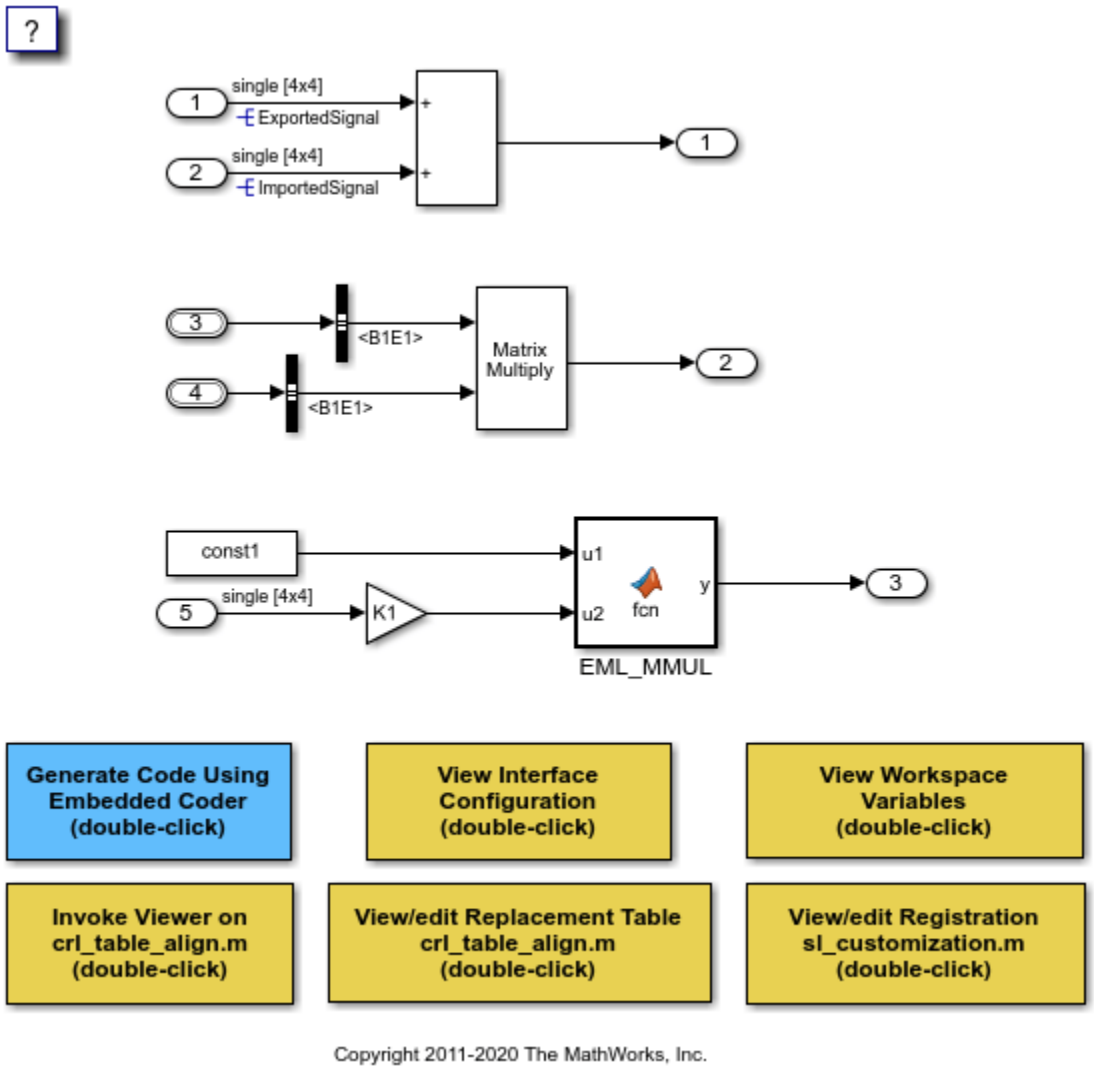
- `Add` - Alignment of exported and imported signals. You can specify an exact value for the alignment in the Signal Properties dialog box or allow the code generator to determine the best alignment based on usage by leaving the alignment value set to -1.

- `Product` - Alignment of virtual and nonvirtual bus types. You can specify an exact value for the alignment for the **Alignment** property of the Simulink.Bus object or allow the code generator to determine the best alignment based on usage by leaving the **Alignment** property set to -1.
- `EML_MMUL` - Alignment of local variables, global variables, and block parameters.

Use the buttons at the bottom of the model window to explore the files that define and register the code replacement mappings. Note that the model is configured to use a GCC, Clang, MSVC, or MinGW compiler.

For more information on specifying data alignment for code replacement, see “Data Alignment for Code Replacement” on page 72-105.

```
open_system('rtwdemo_crlalign')
```



Code Replacement Library Exploration and Verification

This example shows Code Replacement Viewer. You can use the Code Replacement Viewer to:

- Explore which code replacement library to use
- Verify the list of tables in a library and the entries in each table
- Review table entry specifications
- Troubleshoot code replacement misses

The following commands open the Code Replacement Viewer for code replacement table `crl_tablemuldiv`:

```
crl = crl_table_muldiv;
crviewer(crl);
```

For more information on the Code Replacement Viewer, see “Verify Code Replacement Library” on page 72-55.

Build Information

For each entry in a code replacement table, you can specify build information such as the following, for replacement functions:

- Header file dependencies
- Source file dependencies
- Additional include paths
- Additional source paths
- Additional link flags

Additionally, you can specify `RTW.copyFileToBuildDir` to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation. You can specify `RTW.copyFileToBuildDir` by setting it as the value of:

- Property `GenCallback` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`.
- Argument `genCallback` in a call to `registerCFunctionEntry`, `registerCOperationEntry`, or `registerCSemaphoreEntry`.

Note: Models in this example are configured for code generation only because the implementations for the replacement functions are not provided.

For more information on specifying build information, see Specify build information in “Define Code Replacement Library Optimizations” on page 72-36.

Reserved Identifiers

Each function implementation name defined by a code replacement table entry is reserved as a unique identifier. You can specify other identifiers with a table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier related compile and link issues.

For more information on specifying reserved identifiers, see “Reserved Identifiers and Code Replacement” on page 72-131.

Remove Example Code Replacement Libraries

When you finish using the example models, remove the example code replacement libraries and close the example models with these commands:

```
rmpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','crl_demo'));
sl_refresh_customizations;

close_system('rtwdemo_crladdsub', 0)
close_system('rtwdemo_crl_cbo_cao', 0)
close_system('rtwdemo_crlmuldiv', 0)
close_system('rtwdemo_crlfixpt', 0)
close_system('rtwdemo_crlmath', 0)
close_system('rtwdemo_crlmatops', 0)
close_system('rtwdemo_crlblas', 0)
close_system('rtwdemo_crlscalarops', 0)
close_system('rtwdemo_crlcustomentry', 0)
close_system('rtwdemo_crlcoderreplace', 0)
close_system('rtwdemo_crlalign', 0)

drawnow;
if exist('h','var') && ishghandle(h)
  close(h);
end


clear h;
clear isDaDemoSupported;
clear crl;

rtwdemoclean;
```

```
Warning: Unrecognized function or variable
'simmechanics.sli.internal.register_datatypes'.
Warning: Undefined function 'pslinkprivate' for input arguments of type 'char'.
Warning: Unrecognized function or variable 'customizationXpcCrlRegistration'.
```

Improve Execution Efficiency by Reordering Block Operations in the Generated Code

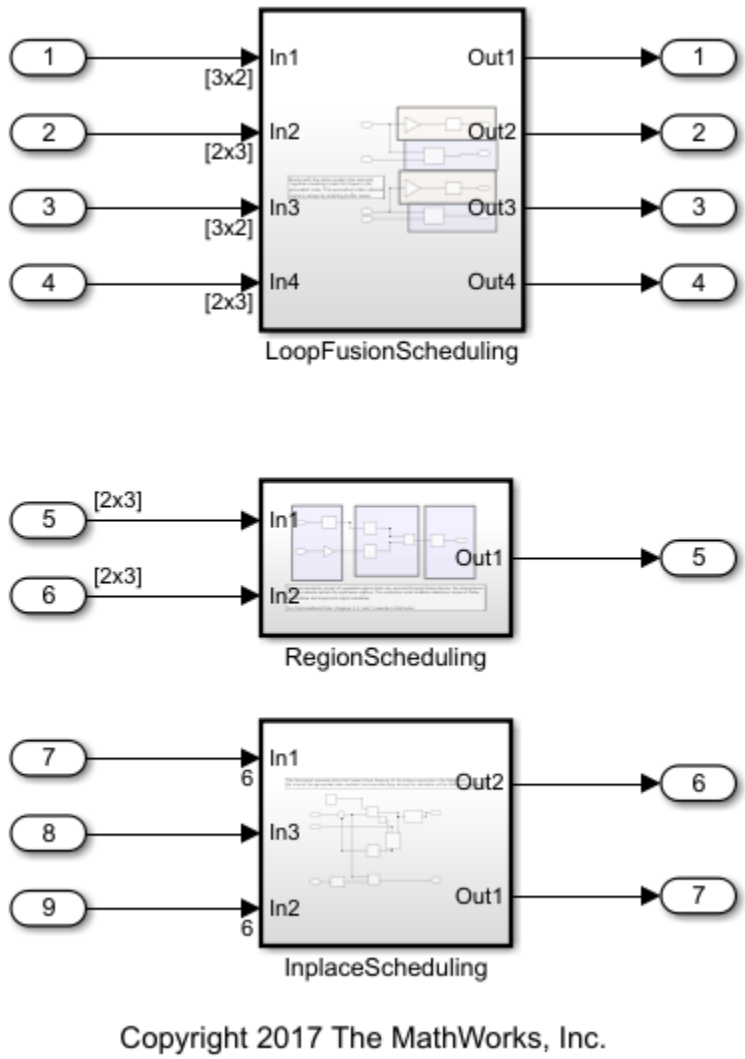
To improve execution efficiency, the code generator can change the block execution order. In the Configuration Parameters dialog box, when you set the **Optimize Block Order** parameter to `Improved Execution Speed`, the code generator can change the block operation order to implement these optimizations:

- Eliminate data copies for blocks that perform inplace operations (that is, use the same input and output variable) and contain algorithm code with unnecessary data copies.
- Combine more `for` loops by executing blocks together that have the same size.
- Reuse the same variable for the input, output, and state of a Unit Delay block by executing the Unit Delay block before upstream blocks.

These optimizations improve execution speed and conserve RAM and ROM consumption.

Example Model

Open the model `rtwdemo_optimizeblockorder`. This model contains three subsystems for demonstrating how reordering block operations improves execution efficiency.



`for` Loop Fusion

The subsystem `LoopFusionScheduling` shows how the code generator reorders block operations so that blocks that have the same output size execute together. This reordering enables `for` loop fusion. Set the **Optimize block order in the generated code** parameter to `Off`.

```
model = 'rtwdemo_optimizeblockorder';
load_system(model);
set_param(model,'OptimizeBlockOrder','off');
```

In your system's temporary folder, create a folder for the build and inspection process and build the model.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
==============================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.364s
```

View the generated code without the optimization. Code for the `LoopFusionScheduling` subsystem:

```
cfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/LoopFusionScheduling','<Root>/RegionScheduling',1, 0);
```

```
/* Output and update for atomic system: '<Root>/LoopFusionScheduling' */
static void LoopFusionScheduling(const real_T rtu_In1[6], const real_T rtu_In2[6],
  const real_T rtu_In3[6], const real_T rtu_In4[6], real_T rty_Out1[6], real_T
  rty_Out2[9], real_T rty_Out3[6], real_T rty_Out4[9])
{
  real_T rtu_In2_0;
  real_T rtu_In2_1;
  int32_T i;
  int32_T rtu_In2_tmp;

  /* Bias: '<S2>/Bias' incorporates:
   *  Gain: '<S2>/Gain'
   */
  for (i = 0; i < 6; i++) {
    rty_Out1[i] = -0.3 * rtu_In1[i] + 0.5;
  }

  /* End of Bias: '<S2>/Bias' */

  /* Product: '<S2>/Product' */
  for (i = 0; i < 3; i++) {
    rtu_In2_tmp = i << 1;
    rtu_In2_0 = rtu_In2[rtu_In2_tmp];
    rtu_In2_1 = rtu_In2[rtu_In2_tmp + 1];
    for (rtu_In2_tmp = 0; rtu_In2_tmp < 3; rtu_In2_tmp++) {
      rty_Out2[rtu_In2_tmp + 3 * i] = rtu_In1[rtu_In2_tmp + 3] * rtu_In2_1 +
        rtu_In2_0 * rtu_In1[rtu_In2_tmp];
    }
  }

  /* End of Product: '<S2>/Product' */

  /* Bias: '<S2>/Bias1' incorporates:
   *  Gain: '<S2>/Gain1'
   */
  for (i = 0; i < 6; i++) {
```

```
    rty_Out3[i] = -0.3 * rtu_In3[i] + 0.5;
  }

  /* End of Bias: '<S2>/Bias1' */

  /* Product: '<S2>/Product1' */
  for (i = 0; i < 3; i++) {
    rtu_In2_tmp = i << 1;
    rtu_In2_0 = rtu_In4[rtu_In2_tmp];
    rtu_In2_1 = rtu_In4[rtu_In2_tmp + 1];
    for (rtu_In2_tmp = 0; rtu_In2_tmp < 3; rtu_In2_tmp++) {
      rty_Out4[rtu_In2_tmp + 3 * i] = rtu_In3[rtu_In2_tmp + 3] * rtu_In2_1 +
        rtu_In2_0 * rtu_In3[rtu_In2_tmp];
    }
  }

  /* End of Product: '<S2>/Product1' */
}
```

With the default execution order, the blocks execute from left to right and from top to bottom. As a result, there are separate `for` loops for the two combinations of Gain and Bias blocks and the Product blocks.

Generate code with the optimization. Set the **Optimize block order in the generated code** parameter to `Improved Execution Speed` and build the model.

```
set_param(model,'OptimizeBlockOrder','Speed');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.657s
```

View the generated code with the optimization.

```
ccfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/LoopFusionScheduling','<Root>/RegionScheduling',1, 0);
```

```
/* Output and update for atomic system: '<Root>/LoopFusionScheduling' */
static void LoopFusionScheduling(const real_T rtu_In1[6], const real_T rtu_In2[6],
  const real_T rtu_In3[6], const real_T rtu_In4[6], real_T rty_Out1[6], real_T
  rty_Out2[9], real_T rty_Out3[6], real_T rty_Out4[9])
{
  real_T rtu_In2_0;
  real_T rtu_In2_1;
  real_T rtu_In4_0;
```

```
  real_T rtu_In4_1;
  int32_T i;
  int32_T rtu_In2_tmp;
  int32_T tmp;
  for (i = 0; i < 3; i++) {
    /* Product: '<S2>/Product' incorporates:
     *  Product: '<S2>/Product1'
     */
    rtu_In2_tmp = i << 1;
    rtu_In2_0 = rtu_In2[rtu_In2_tmp];
    rtu_In2_1 = rtu_In2[rtu_In2_tmp + 1];

    /* Product: '<S2>/Product1' incorporates:
     *  Product: '<S2>/Product'
     */
    rtu_In4_0 = rtu_In4[rtu_In2_tmp];
    rtu_In4_1 = rtu_In4[rtu_In2_tmp + 1];
    for (rtu_In2_tmp = 0; rtu_In2_tmp < 3; rtu_In2_tmp++) {
      /* Product: '<S2>/Product' incorporates:
       *  Product: '<S2>/Product1'
       */
      tmp = 3 * i + rtu_In2_tmp;
      rty_Out2[tmp] = rtu_In1[rtu_In2_tmp + 3] * rtu_In2_1 + rtu_In2_0 *
        rtu_In1[rtu_In2_tmp];

      /* Product: '<S2>/Product1' */
      rty_Out4[tmp] = rtu_In3[rtu_In2_tmp + 3] * rtu_In4_1 + rtu_In4_0 *
        rtu_In3[rtu_In2_tmp];
    }
  }

  for (i = 0; i < 6; i++) {
    /* Bias: '<S2>/Bias' incorporates:
     *  Gain: '<S2>/Gain'
     */
    rty_Out1[i] = -0.3 * rtu_In1[i] + 0.5;

    /* Bias: '<S2>/Bias1' incorporates:
     *  Gain: '<S2>/Gain1'
     */
    rty_Out3[i] = -0.3 * rtu_In3[i] + 0.5;
  }
}
```

In the optimized code, blocks with the same output size execute together. The two sets of Gain and Bias blocks have an output dimension size of `6`, so they execute together. The Product blocks have an output dimension size of `9`, so they execute together. The fusion of `for` loops enables the code generator to set the value of the expression `3 * i + i_0` equal to the temporary variable `tmp_0`. This optimization also improves execution efficiency.

Buffer Reuse for the Input, Output, and State of Unit Delay Blocks

The subsystem `RegionScheduling` shows how the code generator reorders block operations to enable buffer reuse for the input, output, and state of Unit Delay blocks. When computation is part of separate regions that connect only through Delay blocks, the code generator can change the block execution order so that the downstream regions execute before the upstream regions. This execution

order enables maximum reuse of Delay block states and input and output variables. Set the **Optimize block order in the generated code** parameter to `Off` and build the model.

```
set_param(model,'OptimizeBlockOrder','off');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.876s
```

View the generated code without the optimization. Code for the `RegionScheduling` subsystem:

```
cfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/RegionScheduling','/* Model step function',1, 0);
```

```
/* Output and update for atomic system: '<Root>/RegionScheduling' */
static void RegionScheduling(const real_T rtu_In1[6], const real_T rtu_In2[6],
  real_T rty_Out1[6], rtDW_RegionScheduling *localDW)
{
  real_T rtb_Sum_0;
  int32_T i;
  for (i = 0; i < 6; i++) {
    /* Sum: '<S3>/Sum' incorporates:
     *  UnitDelay: '<S3>/Delay'
     *  UnitDelay: '<S3>/UnitDelay'
     */
    rtb_Sum_0 = localDW->Delay_DSTATE[i] + localDW->UnitDelay_DSTATE[i];

    /* UnitDelay: '<S3>/UnitDelay2' */
    rty_Out1[i] = localDW->UnitDelay2_DSTATE[i];

    /* Update for UnitDelay: '<S3>/Delay' incorporates:
     *  Bias: '<S3>/Bias'
     */
    localDW->Delay_DSTATE[i] = rtu_In1[i] + 3.0;

    /* Update for UnitDelay: '<S3>/UnitDelay' incorporates:
     *  Gain: '<S3>/Gain'
     */
    localDW->UnitDelay_DSTATE[i] = 2.0 * rtu_In2[i];

    /* Update for UnitDelay: '<S3>/UnitDelay2' incorporates:
     *  Sum: '<S3>/Sum'
     */
    localDW->UnitDelay2_DSTATE[i] = rtb_Sum_0;
  }
}
```

With the default execution order, the generated code contains the extra, temporary variable `rtb_Sum` and a data copy.

Generate code with the optimization. Set the **Optimize block order in the generated code** parameter to `Improved Execution Speed` and build the model.

```
set_param(model,'OptimizeBlockOrder','Speed');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.757s
```

View the generated code with the optimization.

```
cfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/RegionScheduling','/* Model step function',1, 0);
```

```
/* Output and update for atomic system: '<Root>/RegionScheduling' */
static void RegionScheduling(const real_T rtu_In1[6], const real_T rtu_In2[6],
  real_T rty_Out1[6], rtDW_RegionScheduling *localDW)
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    /* UnitDelay: '<S3>/UnitDelay2' */
    rty_Out1[i] = localDW->UnitDelay2_DSTATE[i];

    /* Sum: '<S3>/Sum' incorporates:
     *  UnitDelay: '<S3>/Delay'
     *  UnitDelay: '<S3>/UnitDelay'
     *  UnitDelay: '<S3>/UnitDelay2'
     */
    localDW->UnitDelay2_DSTATE[i] = localDW->Delay_DSTATE[i] +
      localDW->UnitDelay_DSTATE[i];

    /* Bias: '<S3>/Bias' incorporates:
     *  UnitDelay: '<S3>/Delay'
     */
    localDW->Delay_DSTATE[i] = rtu_In1[i] + 3.0;

    /* Gain: '<S3>/Gain' incorporates:
     *  UnitDelay: '<S3>/UnitDelay'
     */
    localDW->UnitDelay_DSTATE[i] = 2.0 * rtu_In2[i];
  }
}
```

In the optimized code, the blocks in Regions 3, 2, and 1 execute in that order. With that execution order, the generated code does not contain the temporary variable `rtb_Sum` and the corresponding data copy.

Eliminate Data Copies for Blocks That Perform Inplace Operations

The subsystem `InplaceScheduling` shows how the code generator reorders block operations to eliminate data copies for blocks that perform inplace operations. In the Configuration Parameters dialog box, set the **Optimize block order in the generated code** parameter to `Off` and build the model.

```
set_param(model,'OptimizeBlockOrder','off');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.507s
```

View the generated code without the optimization. Code for the `InplaceScheduling` subsystem:

```
cfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/InplaceScheduling','<Root>/LoopFusionScheduling',1, 0);
```

```
/* Output and update for atomic system: '<Root>/InplaceScheduling' */
static void InplaceScheduling(void)
{
  real_T rtb_Max[6];
  real_T acc;
  int32_T idx1;
  int32_T idx2;
  int32_T k;
  for (idx1 = 0; idx1 < 6; idx1++) {
    /* Sum: '<S1>/Sum2x3' incorporates:
     *  Inport: '<Root>/In7'
     *  UnitDelay: '<S1>/Unit Delay'
     */
    acc = rtU.In7[idx1] + rtDWork.UnitDelay_DSTATE[idx1];

    /* Sum: '<S1>/Sum2x3' */
    rtDWork.UnitDelay_DSTATE[idx1] = acc;

    /* MinMax: '<S1>/Max' */
    if (acc <= 2.0) {
      rtb_Max[idx1] = 2.0;
    } else {
      rtb_Max[idx1] = acc;
```

```
    }

    /* End of MinMax: '<S1>/Max' */
  }

  /* S-Function (sdsp2norm2): '<S1>/Normalization' incorporates:
   *  Outport: '<Root>/Out7'
   */
  idx1 = 0;
  idx2 = 0;
  acc = 0.0;
  for (k = 0; k < 6; k++) {
    acc += rtb_Max[idx1] * rtb_Max[idx1];
    idx1++;
  }

  acc = 1.0 / (sqrt(acc) + 1.0E-10);
  for (k = 0; k < 6; k++) {
    rtY.Out7[idx2] = rtb_Max[idx2] * acc;
    idx2++;

    /* Outport: '<Root>/Out6' incorporates:
     *  Bias: '<S1>/Bias'
     *  Inport: '<Root>/In8'
     *  Outport: '<Root>/Out7'
     *  Product: '<S1>/Product'
     */
    rtY.Out6[k] = (rtU.In8 + 1.0) * rtDWork.UnitDelay_DSTATE[k];

    /* Switch: '<S1>/Switch' incorporates:
     *  Inport: '<Root>/In9'
     */
    if (rtU.In9[k] > 0.0) {
      /* Update for UnitDelay: '<S1>/Unit Delay' incorporates:
       *  Sum: '<S1>/Sum2x3'
       */
      rtDWork.UnitDelay_DSTATE[k] = 0.0;
    } else {
      /* Update for UnitDelay: '<S1>/Unit Delay' incorporates:
       *  Sum: '<S1>/Sum2x3'
       */
      rtDWork.UnitDelay_DSTATE[k] = rtb_Max[k];
    }

    /* End of Switch: '<S1>/Switch' */
  }

  /* End of S-Function (sdsp2norm2): '<S1>/Normalization' */
}
```

With the default execution order, the Max block executes before the Product block. To hold the Sum block output, the generated code contains two variables, `UnitDelay_DSTATE` and `rtb_Max`.

Generate code with the optimization. Set the **Optimize block order in the generated code** parameter to `Improved Execution Speed` and build the model.

```
set_param(model,'OptimizeBlockOrder','Speed');
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_optimizeblockorder
### Successful completion of build procedure for: rtwdemo_optimizeblockorder

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
========================================================================================
rtwdemo_optimizeblockorder  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.578s
```

View the generated code with the optimization.

```
cfile = fullfile(cgDir,'rtwdemo_optimizeblockorder_ert_rtw',...
    'rtwdemo_optimizeblockorder.c');
rtwdemodbtype(cfile,'<Root>/InplaceScheduling','<Root>/LoopFusionScheduling',1, 0);
```

```
/* Output and update for atomic system: '<Root>/InplaceScheduling' */
static void InplaceScheduling(void)
{
  real_T acc;
  int32_T idx1;
  int32_T idx2;
  int32_T k;
  for (idx1 = 0; idx1 < 6; idx1++) {
    /* Sum: '<S1>/Sum2x3' incorporates:
     *  Inport: '<Root>/In7'
     *  UnitDelay: '<S1>/Unit Delay'
     */
    acc = rtU.In7[idx1] + rtDWork.UnitDelay_DSTATE[idx1];

    /* MinMax: '<S1>/Max' incorporates:
     *  Sum: '<S1>/Sum2x3'
     */
    rtDWork.UnitDelay_DSTATE[idx1] = acc;

    /* Outport: '<Root>/Out6' incorporates:
     *  Bias: '<S1>/Bias'
     *  Inport: '<Root>/In8'
     *  Product: '<S1>/Product'
     */
    rtY.Out6[idx1] = (rtU.In8 + 1.0) * acc;

    /* MinMax: '<S1>/Max' incorporates:
     *  Sum: '<S1>/Sum2x3'
     */
    if (acc <= 2.0) {
      /* MinMax: '<S1>/Max' */
      rtDWork.UnitDelay_DSTATE[idx1] = 2.0;
    } else {
      /* MinMax: '<S1>/Max' */
      rtDWork.UnitDelay_DSTATE[idx1] = acc;
    }
```

```
    /* End of MinMax: '<S1>/Max' */
  }

  /* S-Function (sdsp2norm2): '<S1>/Normalization' incorporates:
   *  Outport: '<Root>/Out7'
   */
  idx1 = 0;
  idx2 = 0;
  acc = 0.0;
  for (k = 0; k < 6; k++) {
    acc += rtDWork.UnitDelay_DSTATE[idx1] * rtDWork.UnitDelay_DSTATE[idx1];
    idx1++;
  }

  acc = 1.0 / (sqrt(acc) + 1.0E-10);
  for (k = 0; k < 6; k++) {
    rtY.Out7[idx2] = rtDWork.UnitDelay_DSTATE[idx2] * acc;
    idx2++;
  }

  /* End of S-Function (sdsp2norm2): '<S1>/Normalization' */

  /* Update for UnitDelay: '<S1>/Unit Delay' */
  for (idx1 = 0; idx1 < 6; idx1++) {
    /* Switch: '<S1>/Switch' incorporates:
     *  Inport: '<Root>/In9'
     *  MinMax: '<S1>/Max'
     */
    if (rtU.In9[idx1] > 0.0) {
      rtDWork.UnitDelay_DSTATE[idx1] = 0.0;
    }

    /* End of Switch: '<S1>/Switch' */
  }

  /* End of Update for UnitDelay: '<S1>/Unit Delay' */
}
```

The optimized code does not contain the variable `rtb_Max` or the data copy. The generated code contains one variable, `UnitDelay_DSTATE`, for holding the Sum block output. The Product block reads from `UnitDelay_DSTATE` and the Max block reads from and writes to `UnitDelay_DSTATE`.

To implement buffer reuse, the code generator does not violate user-specified block priorities.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Optimize block operation order in generated code

Related Examples

- “Remove Data Copies by Reordering Block Operations in the Generated Code” on page 78-27
- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

Optimize SIMD Code by Performing Fused Multiply Add Operations

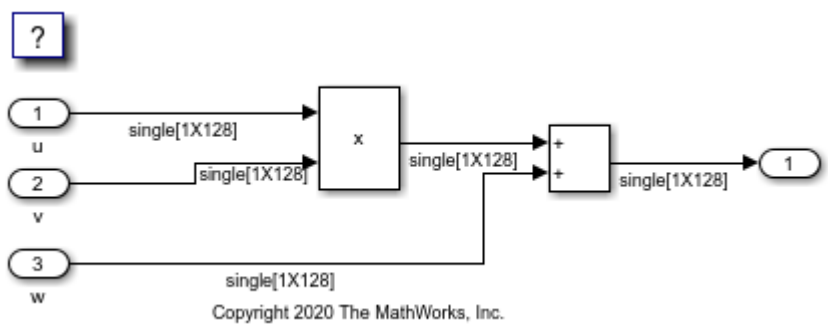
This example shows how to generate optimized single instruction, multiple data (SIMD) code that performs fused multiply-add operations. A fused multiply-add operation is performed in one step with a single rounding rather than performing a multiplication operation followed by an addition, with multiple roundings.

For processors that support fused multiply-add (FMA) instructions, when you set the **Leverage target hardware instruction set** to `FMA`, the code generator generates code that includes FMA intrinsics to perform fused multiply-add operations. Using this optimization improves the execution speed of the generated SIMD code. Because of the single rounding behavior of fused multiply-add operations, this optimization might introduce a mismatch between simulation and code generation results.

Example Model

Open the example model `rtwdemo_simd_with_fma`. The model contains a multiplication operation followed by an addition.

```
model = 'rtwdemo_simd_with_fma';
open_system(model)
```



Generate SIMD Code Without FMA Optimization

1. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

2. On the **Optimization** pane, set the **Leverage target hardware instruction set extensions** to `SSE2`. Alternatively, you can use the command-line API:

```
set_param(model, 'InstructionSetExtensions', 'SSE2');
```

3. Build the model.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_simd_with_fma
### Successful completion of build procedure for: rtwdemo_simd_with_fma
```

```
Build Summary

Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
rtwdemo_simd_with_fma  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 20.933s
```

4. Inspect the generated `rtwdemo_simd_with_fma_step` step function in the `rtwdemo_simd_with_fma.c`.

```
file = fullfile('rtwdemo_simd_with_fma_ert_rtw','rtwdemo_simd_with_fma.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void rtwdemo_simd_with_fma_step(void)
{
  int32_T i;
  for (i = 0; i <= 124; i += 4) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/u'
     *  Inport: '<Root>/v'
     *  Inport: '<Root>/w'
     */
    _mm_storeu_ps(&rtwdemo_simd_with_fma_Y.Out1[i], _mm_add_ps(_mm_mul_ps
      (_mm_loadu_ps(&rtwdemo_simd_with_fma_U.single1X128[i]), _mm_loadu_ps
       (&rtwdemo_simd_with_fma_U.single1X128_f[i])), _mm_loadu_ps
      (&rtwdemo_simd_with_fma_U.single1X128_k[i])));
  }
}
```

The generated code contains a vectorized loop that performs the multiplication and addition operations sequentially, with two roundings.

Generate SIMD Code with FMA Optimization

1. To enable FMA optimization, on the **Optimization** pane, set the **Leverage target hardware instruction set extensions** to `FMA` Alternatively, you can use the command-line API:

```
set_param(model, 'InstructionSetExtensions', 'FMA');
```

2. Build the model again.

```
[status, message, messageid] = rmdir('slprj', 's');
currentDir = pwd;
[~,cgDir] = rtwdemodir();
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_simd_with_fma
### Successful completion of build procedure for: rtwdemo_simd_with_fma

Build Summary
```

```
Top model targets built:

Model                   Action                        Rebuild Reason
========================================================================================
rtwdemo_simd_with_fma  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 40.814s
```

3. Inspect the generated `rtwdemo_simd_with_fma_step` step function in the `rtwdemo_simd_with_fma.c`.

```
file = fullfile('rtwdemo_simd_with_fma_ert_rtw','rtwdemo_simd_with_fma.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void rtwdemo_simd_with_fma_step(void)
{
  int32_T i;
  for (i = 0; i <= 120; i += 8) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/u'
     *  Inport: '<Root>/v'
     *  Inport: '<Root>/w'
     */
    _mm256_storeu_ps(&rtwdemo_simd_with_fma_Y.Out1[i], _mm256_fmadd_ps
                     (_mm256_loadu_ps(&rtwdemo_simd_with_fma_U.single1X128[i]),
                      _mm256_loadu_ps(&rtwdemo_simd_with_fma_U.single1X128_f[i]),
                      _mm256_loadu_ps(&rtwdemo_simd_with_fma_U.single1X128_k[i])));
  }
}
```

The vectorized loop contains the intrinsic function `mm256_fmadd_ps` that performs multiplication-addition in one step, with a single rounding. This optimization improves the execution speed of the generated code. It is possible that the execution of the generated code does not produce the same result as the simulation.

The presence of FMA optimization is reported in the code generation report as follows.

Code Information

System Target File	ert.tlc
Hardware Device Type	Intel->x86-64 (Windows64)
Simulink Coder Version	9.6 (R2021b) 14-May-2021
Timestamp of Generated Source Code	Mon Jun 28 16:20:46 2021
Location of Generated Source Code	C:\Users\sferdous\Documents\Examples\ecoder-ex02442880\rtwdemo_simd_with_fma_ert_rtw
Type of Build	Model
Objectives Specified	Unspecified

Additional Information

Code Generation Advisor	Not run
FMA Instruction Set Extensions	FMA optimization is triggered

It is reported that the code generator used FMA instruction set extensions and the FMA optimization is triggered for the `rtwdemo_simd_with_fma` model code. If you are working with a model hierarchy and FMA optimization is triggered only for referenced models, the code generator does not report the presence of FMA optimization in the code generation report created for the top model.

Clean Up Example Folders and Files

Close the model and remove temporary folders and files.

```
bdclose(model);
cd(currentDir);
rtwdemoclean;
```

See Also

Leverage target hardware instruction set extensions

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Generate SIMD Code from Simulink Blocks” on page 76-105
- “Generate SIMD Code for MATLAB Functions” on page 97-27

Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface

When you generate code from MATLAB Function, MATLAB System, and For Each subsystem blocks, by default the code generator produces code that implements `for`-loops in a single thread. If you have Parallel Computing Toolbox, you can specify that these `for`-loops run in parallel by selecting the **Generate parallel for-loops** parameter and by including the `parfor` statement in your MATLAB Function block and MATLAB System block code. Running iterations in parallel can significantly improve the execution speed of the generated code.

The code generator implements the `for`-loops in parallel by using the Open Multiprocessing (OpenMP) application interface to support shared-memory, multicore code generation. By default, the code generator uses as many threads as it finds available. For MATLAB Function and MATLAB System blocks using `parfor`, if you specify the number of threads to use, the code generator uses at most that number of threads, even if additional threads are available. The code generator cannot generate parallel `for`-loops for For Each Subsystem blocks that contain S-functions.

Generate OpenMP Code from MATLAB Function or MATLAB System Blocks

In this example, you generate OpenMP code from a MATLAB Function block that contains the `parfor` statement as part of its code. Use the `parfor` to speed up execution when you have many iterations of a simple calculation or a loop iteration takes a long time to execute. For more on how `parfor`-loops improve execution speed and when to use them, see “Algorithm Acceleration Using Parallel for-Loops (parfor)”.

1. Create a model that contains a MATLAB Function or MATLAB System block. This example uses a MATLAB Function block.
2. Add this code to the MATLAB Function block.

```
function y = openmpex(u) %#codegen

%   Copyright 2019 The MathWorks, Inc.

A = ones(20,50);
t = 0;

parfor (i = 1:10,4)
    A(i,1) = A(i,1) + 1;
end

y = A(1,4) + u + t;
```

   Note the `parfor` input value of 4 specifies the maximum number of threads to use.
3. Connect the blocks.

4. Open the Configuration Parameters dialog box. On the **Code Generation** pane, and set the **System Target File** parameter to `ert.tlc`.
5. On the **Solver** pane, set the **Type** parameter to `Fixed-step`.
6. On the **Optimization** pane, set the **Priority** parameter to `Maximize execution speed`. Now, the **Generate parallel for-loops** parameter is automatically selected. This parameter enables the compiler to compute loops in parallel.
7. Build the model and generate code.

   In the generated code, the pragma instructs the compiler to execute the looping in OpenMP parallel `for`-loops through multithreading:

   ```
   #pragma omp parallel for num_threads(4 >
                   omp_get_max_threads() ? omp_get_max_threads() : 4)
   ```

   The number 4 indicates the number of processing threads.

Because the loop body can execute in parallel on multiple threads, it must conform to certain restrictions. If Embedded Coder detects loops that do not conform to `parfor` specifications, it produces an error. For more information, see “parfor Restrictions”.

See Also

More About

- “Classification of Variables in parfor-Loops”
- “Reduction Assignments in parfor-Loops”
- “Unroll Parallel for-Loop That Has Small Number of Iterations” on page 79-28

Accelerate Pedestrian Detection with SIMD Code

This example shows how to perform automatic detection and tracking of pedestrians in a video from a moving camera. You can generate SIMD code using Intel™ AVX2 technology to increase the number of frames per second in the video. A higher frame rate improves the quality and speed of the detection and tracking system.

Download Pedestrian Detection Video

This example uses the CamVid dataset [1] from the University of Cambridge. This dataset is a collection of images and videos containing street-level views obtained while driving. Download the file from the MathWorks® website:

```
CAMVideoURL= matlab.internal.examples.downloadSupportFile('vision/data',...
    'CamVid_0016E5.mp4');
```

Create and Configure a Monocular Camera Sensor to Detect Pedestrians

Create a `monoCamera` (Automated Driving Toolbox) object that holds the configuration information for a monocular camera sensor. This information includes camera intrinsics such as the focal point, optical center, and image size of the camera and camera extrinsics such as the height of the camera and yaw angle.

```
focalLength=[526.3158, 526.3158];
principalPoint = [319.5 179.5];
imageSize = [360 640];
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
height = 1.2;
yaw =  13;
sensor = monoCamera(intrinsics,height,'Yaw',yaw);
```

Create an `acfObjectDetector` (Computer Vision Toolbox) object for detecting upright people. Configure the detector for use with the camera.

```
detector = peopleDetectorACF('caltech');
pedWidth = [0.5, 1.5];
detectorMonoCam = configureDetectorMonoCamera(detector, sensor, pedWidth);
```

To support code generation, the `acfObjectDetector` object must be in the form of a structure. Use the `toStruct` (Computer Vision Toolbox) function to create a structure that stores the properties of the input `acfObjectDetector` object in `Classifier` and `TraininginOptions` fields. Save the structure to a `.mat` file.

```
sModel = toStruct(detectorMonoCam);
sModel.TrainingOptions.WindowStride = 4;
sModel.TrainingOptions.NumScaleLevels = 8;
save('model.mat','-struct','sModel','TrainingOptions','Classifier');
```

Examine `detectandTrack` Entry-Point Function

The `detectandTrack.m` file is the main entry-point function for code generation. The `detectandTrack` function loads the `model.mat` file that you just created and recreates an `acfObjectDetector` object to detect pedestrians within the input video.

The `detectandTrack` function uses the `vision.VideoFileReader` (Computer Vision Toolbox) system object to read frames from the input video, calls the `setupTracker` and `detectObjects`

functions, and uses the `updateTracks` function of the `multiObjectTracker` object to update the tracker with new detections. The `detectandTrack` function uses a `vision.DeployableVideoPlayer` (Computer Vision Toolbox) system object to display the pedestrian detection video output.

```
type detectAndTrack.m
```

```
function detectAndTrack(CAMVideoURL)
%#codegen

% Create an ACF object detector for code generation.
model = coder.load('model.mat');
detector = acfObjectDetector(model.Classifier,model.TrainingOptions);

% Create a multi-object tracker to track multiple objects with Kalman filters.
[tracker, positionSelector] = setupTracker;


% Set-up the video reader and player.
videoFReader   = vision.VideoFileReader(CAMVideoURL);
videoFWriter   = vision.VideoFileWriter('CamVid_0016E5_result.avi', 'FrameRate', 25);
depVideoPlayer = vision.DeployableVideoPlayer;
sampleFrame = videoFReader();
frameSize   = [size(sampleFrame,1) size(sampleFrame,2)];

% Generate the sample detection from the sample frame.
detectionThresh = 50;
detections = detectObjects(detector, sampleFrame, 0, detectionThresh);
costMatrix = zeros([0 numel(detections)]);
[confirmedTracks,~,allTracks] = updateTracks(tracker, detections, 0, costMatrix);
coder.varsize('allTracks');

currentStep = 0;
totalTime = 0;
while ~isDone(videoFReader)
    % Update frame counters.
    currentStep = currentStep + 1;

    % Read the next frame.
    frame = videoFReader();

    tic;

    % Detect the pedestrian every 2 frames.
    detections = {};
    if rem(currentStep,2)
        detections = detectObjects(detector, frame, currentStep, detectionThresh);
    end

    % Track the pedestrian every frame.
    costMatrix = detectionToTrackCost(allTracks, detections, positionSelector, tracker.Assignmen
    [confirmedTracks,~,allTracks] = updateTracks(tracker, detections, currentStep, costMatrix);

    % Remove the tracks for people that are far away.
    confirmedTracks = removeNoisyTracks(confirmedTracks, positionSelector, frameSize);
    time = toc;
    totalTime = totalTime+time;
```

```
    % Insert tracking and frame rate annotations.
    framerate = currentStep/totalTime;
    frame = insertTrackBoxes(frame, confirmedTracks, positionSelector);
    frame = insertText(frame, [20 20], framerate, 'AnchorPoint', 'LeftBottom');

    % Write to the output file and display the annotated frame.
    videoFWriter(frame);
    depVideoPlayer(frame);
end

release(videoFWriter);
```

Examine Supporting Files

The `setupTracker` function creates a `multiObjectTracker` (Automated Driving Toolbox) to track multiple objects with Kalman filters. The outputs of `setupTracker` are:

- `tracker` —The `multiObjectTracker` that is configured for this case.
- `positionSelector` —A matrix that specifies which elements of the State vector define the position: `position = positionSelector * State`

```
type setupTracker.m
```

```
function [tracker, positionSelector] = setupTracker()
tracker = multiObjectTracker('FilterInitializationFcn', @initBboxFilter, ...
    'AssignmentThreshold', 1, ...
    'NumCoastingUpdates', 5, ...
    'ConfirmationThreshold', [1 5], ...
    'MaxNumTracks', 10, ...
    'HasCostMatrixInput', true);



% Creat a matrix that specifies which state vector elements are the position.
positionSelector = [1 0 0 0 0 0 0 0; ...
                    0 0 1 0 0 0 0 0; ...
                    0 0 0 0 1 0 0 0; ...
                    0 0 0 0 0 0 1 0];
end
```

The `initBboxFilter` function defines a Kalman filter to filter the bounding box measurement.

```
type initBboxFilter.m
```

```
function filter = initBboxFilter(Detection)
% Step 1: Define the motion model and state.
%   Use a constant velocity model for a bounding box on the image.
%   The state is [x; vx; y; vy; w; wv; h; hv]
%   The state transition matrix is:
%       [1 dt 0  0 0  0 0  0;
%        0  1 0  0 0  0 0  0;
%        0  0 1 dt 0  0 0  0;
%        0  0 0  1 0  0 0  0;
%        0  0 0  0 1 dt 0  0;
%        0  0 0  0 0  1 0  0;
%        0  0 0  0 0  0 1 dt;
%        0  0 0  0 0  0 0  1]
%   Assume dt = 1. This example does not consider a time-variant transition
```

```
%   model for the linear Kalman filter.
    dt = 1;
    cvel =[1 dt; 0 1];
    A = blkdiag(cvel, cvel, cvel, cvel);

% Step 2: Define the process noise.
%   The process noise represents the parts of the process that the model
%   does not take into account. For example, in a constant velocity model,
%   the acceleration is neglected.
    G1d = [dt^2/2; dt];
    Q1d = G1d*G1d';
    Q = blkdiag(Q1d, Q1d, Q1d, Q1d);

% Step 3: Define the measurement model.
%   Only the position ([x;y;w;h]) is measured.
%   The measurement model is
    H = [1 0 0 0 0 0 0 0; ...
         0 0 1 0 0 0 0 0; ...
         0 0 0 0 1 0 0 0; ...
         0 0 0 0 0 0 1 0];

% Step 4: Map the sensor measurements to an initial state vector.
%   Because there is no measurement of the velocity, the v components are
%   initialized to 0:
    state = [Detection.Measurement(1); ...
             0; ...
             Detection.Measurement(2); ...
             0; ...
             Detection.Measurement(3); ...
             0; ...
             Detection.Measurement(4); ...
             0];

% Step 5: Map the sensor measurement noise to a state covariance.
%   For the parts of the state that the sensor measures directly, use the
%   corresponding measurement noise components. For the parts that the
%   sensor does not measure, assume a large initial state covariance. That way,
%   future detections can be assigned to the track.
    L = 100; % Large value
    stateCov = diag([Detection.MeasurementNoise(1,1), ...
                     L, ...
                     Detection.MeasurementNoise(2,2), ...
                     L, ...
                     Detection.MeasurementNoise(3,3), ...
                     L, ...
                     Detection.MeasurementNoise(4,4), ...
                     L]);

% Step 6: Create the correct filter.
%   In this example, all the models are linear, so use trackingKF as the
%   tracking filter.
    filter = trackingKF(...
        'StateTransitionModel', A, ...
        'MeasurementModel', H, ...
        'State', state, ...
        'StateCovariance', stateCov, ...
        'MeasurementNoise', Detection.MeasurementNoise, ...
```

```
        'ProcessNoise', Q);
end
```

The `detectObjects` function detects pedestrians in an image.

```
type detectObjects
```

```
function detections = detectObjects(detector, frame, frameCount, thresh)
    % Run the detector and return a list of bounding boxes: [x, y, w, h]
    [bboxes_raw, scores] = detect(detector, frame);

    numDetections = sum(scores>thresh);
    bboxes = bboxes_raw(scores>thresh, :);

    % Define the measurement noise.
    L = 100;
    measurementNoise = [L 0  0  0; ...
                        0 L  0  0; ...
                        0 0 L/2 0; ...
                        0 0  0 L/2];

    % Formulate the detections as a list of objectDetection reports.
    detections = cell(numDetections, 1);
    objAttr = struct('VisionID', uint32(0), 'Size', zeros(1,3,'single'), ...
        'RadarID', uint16(0), 'Status', uint8(0), 'Amplitude', single(0), ...
        'RangeMode', uint8(0));

    for i = 1:numDetections
        detections{i} = objectDetection(frameCount, bboxes(i, :), ...
            'MeasurementNoise', measurementNoise, 'ObjectAttributes', {objAttr});
    end
end
```

The `removeNoisyTracks` function removes noisy tracks. A track is noisy if its predicted bounding box is too small. Typically, this implies the pedestrian is far away.

```
type removeNoisyTracks.m
```

```
function tracks = removeNoisyTracks(tracks, positionSelector, imageSize)

    if isempty(tracks)
        return
    end

    % Extract the positions from all the tracks.
    positions = getTrackPositions(tracks, positionSelector);
    % The track is 'invalid' if the predicted position is about to move out
    % of the image, or the bounding box is too small.
    invalid = ( positions(:, 1) < 1 | ...
                positions(:, 1) + positions(:, 3) > imageSize(2) | ...
                positions(:, 3) <= 5 | ...
                positions(:, 4) <= 10 );
    tracks(invalid) = [];
end
```

The `insertTrackBoxes` function inserts bounding boxes in an image.

```
type insertTrackBoxes.m
```

```
function I = insertTrackBoxes(I, tracks, positionSelector)
    if isempty(tracks)
        return
    end

    bboxes = getTrackPositions(tracks, positionSelector);
    for i=1:size(bboxes,1)
        box = bboxes(i, :);
        I = insertShape(I,'Rectangle',int32(box));
    end
end
```

Configure Code Generation Configuration Object

To generate a standalone executable for the `detectandTrack` entry-point function, use the `coder.config` function to create a `coder.EmbeddedCodeConfig` object for an `exe` target. This object contains the configuration parameters that the `codegen` function uses for generating an executable program with Embedded Coder™.

```
ecfg = coder.config('exe');
```

Specify an example main C function that the code generator compiles to create a test executable.

```
ecfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Optimize the build for faster running executables and specify that the code generator does not produce code to handle integer overflow and produces code to support nonfinite values (`Inf` and `Nan`) only if they are used.

```
ecfg.BuildConfiguration = 'Faster Runs';
ecfg.SaturateOnIntegerOverflow = false;
ecfg.SupportNonFinite = true;
```

Allocate memory dynamically on the heap for variable-size arrays whose size (in bytes) is greater than or equal to the value of the `DynamicMemoryAllocationThreshold` parameters.

```
ecfg.DynamicMemoryAllocation = 'Threshold';
ecfg.DynamicMemoryAllocationThreshold = 2e8;
```

Because this example generates code from Automated Driving Toolbox™ and Computer Vision Toolbox™ functions, the generated code must be portable and not rely on 3rd party libraries. To generate portable code that you can retarget for an Intel device, create a `coder.HardwareImplementation` object and specify a nonhost target. Then, configure the production hardware settings to match those of an Intel device.

```
ecfg.HardwareImplementation.ProdHWDeviceType = "Generic->Custom";
ecfg.HardwareImplementation.ProdBitPerLong = 64;
ecfg.HardwareImplementation.ProdBitPerPointer = 64;
ecfg.HardwareImplementation.ProdBitPerPtrDiffT = 64;
ecfg.HardwareImplementation.ProdBitPerSizeT = 64;
ecfg.HardwareImplementation.ProdEndianess = "LittleEndian";
ecfg.HardwareImplementation.ProdIntDivRoundTo = "Zero";
ecfg.HardwareImplementation.ProdLargestAtomicFloat = "Float";
ecfg.HardwareImplementation.ProdWordSize = 64;
```

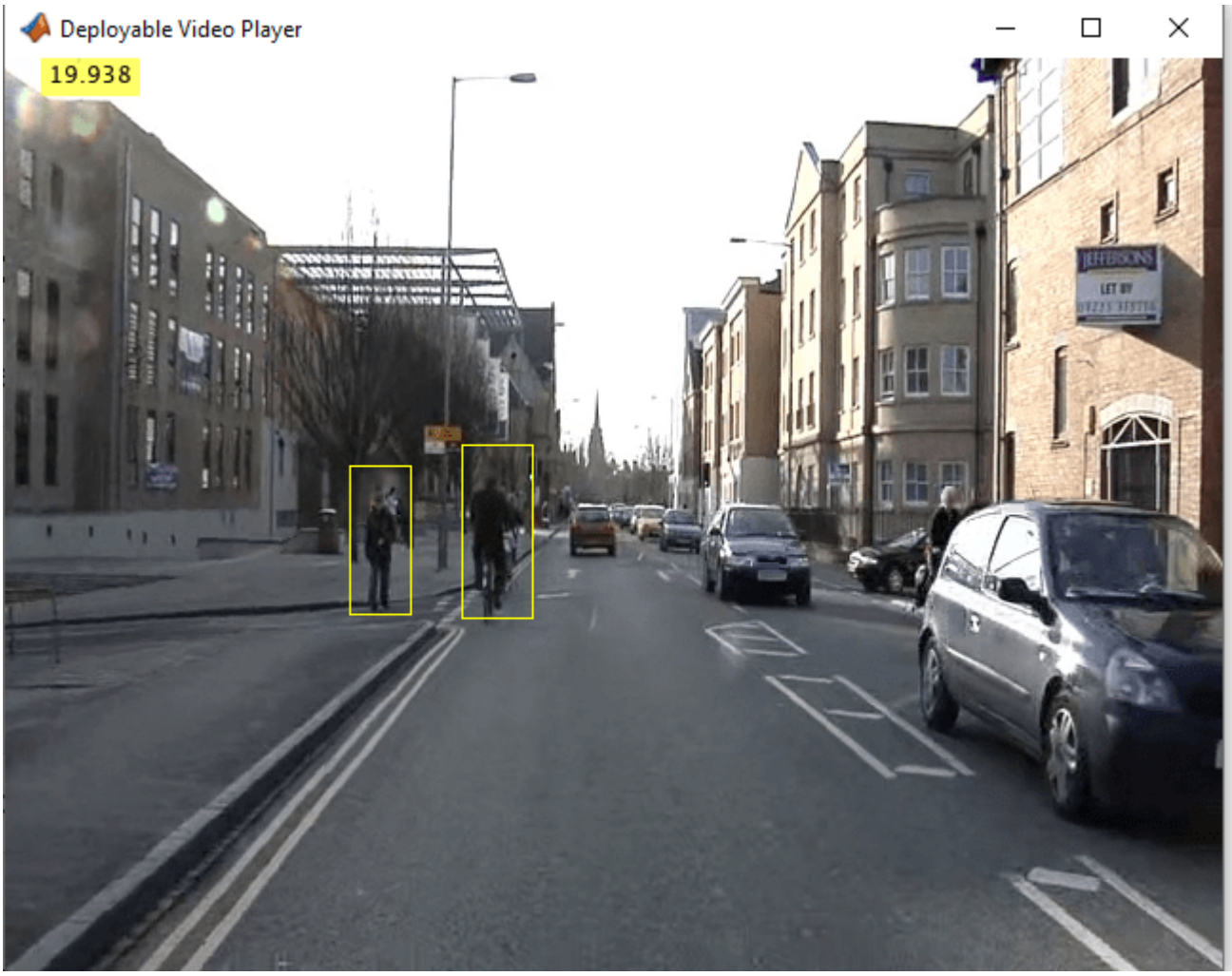
Generate Non-SIMD Code

```
inputs = coder.Constant(CAMVideoURL);
codegen -args inputs -config ecfg detectAndTrack.m
```

```
Code generation successful.
```

Run the executable and observe the frame rate at the top left of the video. This example runs the executable on Windows. To run the executable on Linux, change the command to `!./detectAndTrack`.

```
!detectAndTrack.exe
```



Generate SIMD Code

Configure the code generation configuration object to generate SIMD code using AVX2 technology.

```
ecfg.InstructionSetExtensions = "AVX2";
```

Generate code.

```
codegen -args inputs -config ecfg detectAndTrack.m
```

```
Code generation successful.
```

Run the executable and observe the higher frame rate.

```
!detectAndTrack.exe
```

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic object classes in video: A high-definition ground truth database." Pattern Recognition Letters 30.2 (2009): 88-97.

See Also

Related Examples

- “Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection” on page 83-2
- “Generate SIMD Code for MATLAB Functions” on page 97-27

- “Generate Code for Detecting Objects in Images by Using ACF Object Detector” (Computer Vision Toolbox)

Memory Usage in Embedded Coder

- “Optimize Generated Code Using Minimum and Maximum Values” on page 80-2
- “Reduce Global Variables in Nonreusable Subsystem Functions” on page 80-8
- “Optimize Generated Code by Packing Boolean Data into Bitfields” on page 80-11
- “Optimize Generated Code by Passing Reusable Subsystem Outputs as Individual Arguments” on page 80-14
- “Convert Data Copies to Pointer Assignments” on page 80-18
- “Reuse Buffers of Different Sizes and Dimensions” on page 80-23
- “Remove Reset and Disable Functions from the Generated Code” on page 80-29
- “Control Operator Type in Generated Code” on page 80-32
- “Reduce Memory Usage for Models Containing Referenced Models” on page 80-35
- “Optimize Generated Code Using Code Efficiency Tools and Techniques” on page 80-39

Optimize Generated Code Using Minimum and Maximum Values

To optimize the generated code for your model, you can choose an option to use input range information, also known as design minimum and maximum, that you specify on signals and parameters. These minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

In the Configuration Parameters dialog box, on the **Optimization** pane, when you select the **Optimize using specified minimum and maximum values** check box, the software uses the minimum and maximum values to derive range information for downstream signals in the model. It then uses this derived range information to determine if it is possible to streamline the generated code by:

- Reducing expressions to constants
- Removing dead branches of conditional statements
- Eliminating unnecessary mathematical operations

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks
  - Block outputs
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
  - `Simulink.Signal` objects
- Before generating code, test the minimum and maximum values for signals and parameters. Otherwise, optimization might result in numerical mismatch with simulation. You can simulate your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

  Enable Simulation Range Checking

  1. In the Configuration Parameters dialog box, select **Diagnostics** > **Data Validity**.
  2. On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to `warning` or `error`.
- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream.

Optimize Generated Code Using Specified Minimum and Maximum Values

This example shows how the minimum and maximum values specified on signals and parameters in a model are used to optimize the generated code.

Overview

The specified minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

This optimization uses these values to streamline the generated code. For example, it reduces expressions to constants or removes dead branches of conditional statements.

NOTE: Make sure the minimum and maximum values that you specify are valid limits. Otherwise, this optimization might result in numerical mismatch with simulation.

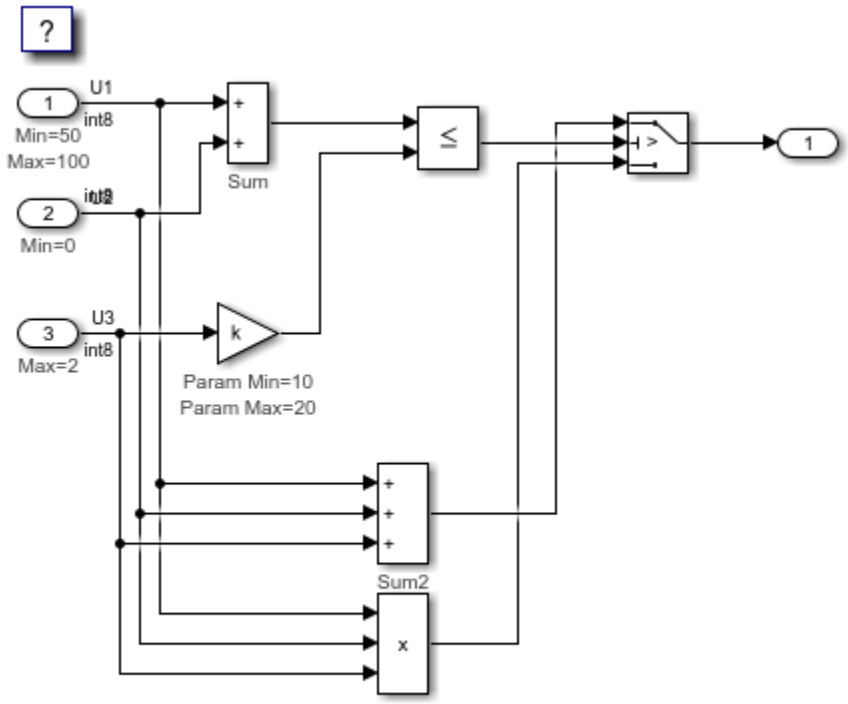
The benefits of optimizing the generated code are:

- Reducing the ROM and RAM consumption.
- Improving the execution speed.

Review Minimum and Maximum Information

Consider the `MinMaxOptimization` model. In this model, there are minimum and maximum values specified on Inports and on the gain parameter of the Gain block.

```
model = 'MinMaxOptimization';
open_system(model);
```



Copyright 2010-2020 The MathWorks, Inc.

Generate Code Without This Optimization

First, generate code for this model without considering the min and max values.

```
slbuild(model)
```

```
### Starting build procedure for: MinMaxOptimization
### Successful completion of build procedure for: MinMaxOptimization

Build Summary

Top model targets built:

Model                Action                        Rebuild Reason
========================================================================================
MinMaxOptimization  Code generated and compiled.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.933s
```

A portion of `MinMaxOptimization.c` is listed below.

```
cfile = fullfile('MinMaxOptimization_ert_rtw','MinMaxOptimization.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

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

```
  int32_T tmp;

  /* Sum: '<Root>/Sum' incorporates:
   *  Inport: '<Root>/U1'
   *  Inport: '<Root>/U2'
   *  Sum: '<Root>/Sum2'
   *  Switch: '<Root>/Switch'
   */
  tmp = U1 + U2;

  /* Switch: '<Root>/Switch' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/U3'
   *  RelationalOperator: '<Root>/Relational Operator'
   *  Sum: '<Root>/Sum'
   */
  if (tmp <= k * U3) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Sum: '<Root>/Sum2'
     */
    rtY.Out1 = tmp + U3;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/U1'
     *  Inport: '<Root>/U2'
     *  Product: '<Root>/Product'
     */
    rtY.Out1 = U1 * U2 * U3;
  }
}
```

Enable This Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Optimize using the specified minimum and maximum values**.

Alternatively, you can enable this optimization by setting the command-line parameter.

```
set_param(model, 'UseSpecifiedMinMax', 'on');
```

Generate Code with This Optimization

In the model, with the specified minimum and maximum values for `U1` and `U2`, the sum of `U1` and `U2` has a minimum value of 50. Considering the range of `U3` and the specified minimum and maximum values for the Gain block parameter, the maximum value of the Gain block's output is 40.

The output of the Relational Operator block remains false, and the output of the Switch block remains the product of the three inputs.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: MinMaxOptimization
### Successful completion of build procedure for: MinMaxOptimization

Build Summary
```

```
Top model targets built:

Model               Action                        Rebuild Reason
========================================================================================
MinMaxOptimization  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.108s
```

View the optimized code from `MinMaxOptimization.c`.

```
cfile = fullfile('MinMaxOptimization_ert_rtw','MinMaxOptimization.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void MinMaxOptimization_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/U1'
   *  Inport: '<Root>/U2'
   *  Inport: '<Root>/U3'
   *  Product: '<Root>/Product'
   *  Switch: '<Root>/Switch'
   */
  rtY.Out1 = U1 * U2 * U3;
}
```

Close the model and cleanup.

```
bdclose(model)
```

Limitations

- This optimization does not take into account minimum and maximum values for:
  - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.
- If you use Polyspace software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace might mark this code orange because it does not have information about the minimum and maximum values for the inputs to the division.

  Polyspace Code Prover automatically captures some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

Polyspace Code Prover stores these values in a Data Range Specification (DRS) file. However, they do not capture all minimum and maximum values in your Simulink model. To provide additional minimum and maximum information to Polyspace, you can manually define a DRS file.

- If you are using double-precision data types and the **non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, code is generated once for the subsystem and shares this code among the multiple instances of the subsystem.
- The Model Advisor check Check safety-related optimization settings generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable.

See Also

Optimize using the specified minimum and maximum values

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Specify Signal Ranges”

Reduce Global Variables in Nonreusable Subsystem Functions

Global variables can increase memory requirements and reduce execution speed. To reduce global RAM for a nonreusable subsystem, you can generate a function interface that passes data through arguments instead of global variables. The Subsystem block parameter **Function interface** provides this option. To compare the outputs for the **Function interface** options, first generate a function for a subsystem with a `void-void` interface, and then generate a function with arguments.

Generate void-void Function

By default, when you configure a Subsystem block as a nonreusable function, it generates a `void-void` interface.

1. Open the example model `RollAxisAutopilot`.

   ```
   openExample('RollAxisAutopilot');
   ```

2. Right-click the subsystem `RollAngleReference`. From the list select `Block Parameter (Subsystem)`.
3. In the Block Parameters dialog box, confirm that the **Treat as atomic unit** check box is selected.
4. Click the **Code Generation** tab and set the **Code interface packaging** parameter to `Nonreusable function`.
5. The **Function interface** parameter is already set to `void_void`.
6. Click **Apply** and **OK**.
7. Repeat steps 2–6, for the other subsystems `HeadingMode` and `BasicRollMode`.
8. Generate code and the static code metrics report for `RollAxisAutopilot`. This model is configured to generate a code generation report and to open the report automatically. For more information, see “Generate Static Code Metrics Report for Simulink Model” on page 54-29.

In the code generation report, in `RollAxisAutopilot.c`, the generated code for subsystem `RollAngleReference` contains a `void-void` function definition:

```
static void RollAngleReference(void)
{
 ...
}
```

In the static code metrics report, navigate to **Global Variables**. With the `void_void` option, the number of bytes for global variables is `47`.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtU	26	14	6
[+] rtDW	17	34	24
[+] rtY	4	1	1
Total	**47**	**49**	

Next, generate the same function with the `Allow arguments (Optimized)` option to compare the results.

Generate Function with Arguments

To reduce global RAM, improve ROM usage and execution speed, generate a function that allows arguments:

1. Open the Subsystem Block Parameter dialog box for `RollAngleReference`.
2. Click the **Code Generation** tab. Set the **Function interface** parameter to `Allow arguments (Optimized)`.
3. Click **Apply** and **OK**.
4. Repeat steps 2 and 3, for the other subsystems `HeadingMode` and `BasicRollMode`.
5. Generate code and the static code metrics report for `RollAxisAutopilot`.

In the code generation report, in `RollAxisAutopilot.c`, the generated code for subsystem `RollAngleReference` now has arguments:

```
static real32_T RollAngleReference(real32_T rtu_Phi,...
                                   boolean_T rtu_AP_Eng,...
                                   real32_T rtu_Turn_Knob)
 {
 ...
 }
```

In the static code metrics report, navigate to **Global Variables**. With the `Allow arguments` option set, the total number of bytes for global variables is now `39` bytes.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtU	26	11	11
[+] rtDW	9	17	14
[+] rtY	4	2	2
Total	**39**	**30**	

In some cases, when generating optimized code, the code generator might not generate a function that has arguments. To generate a predictable function interface that has arguments, set **Function interface** to `Allow arguments (Match graphical interface)`.

See Also

"Function interface"

Related Examples

- "Generate Predictable Function Interface to Match Graphical Block Interface" on page 5-2
- "Design Techniques to Optimize Models for Efficient Code Generation" on page 76-7
- "Generate Subsystem Code as Separate Function and Files" on page 4-7
- "Optimize Generated Code by Passing Reusable Subsystem Outputs as Individual Arguments" on page 80-14

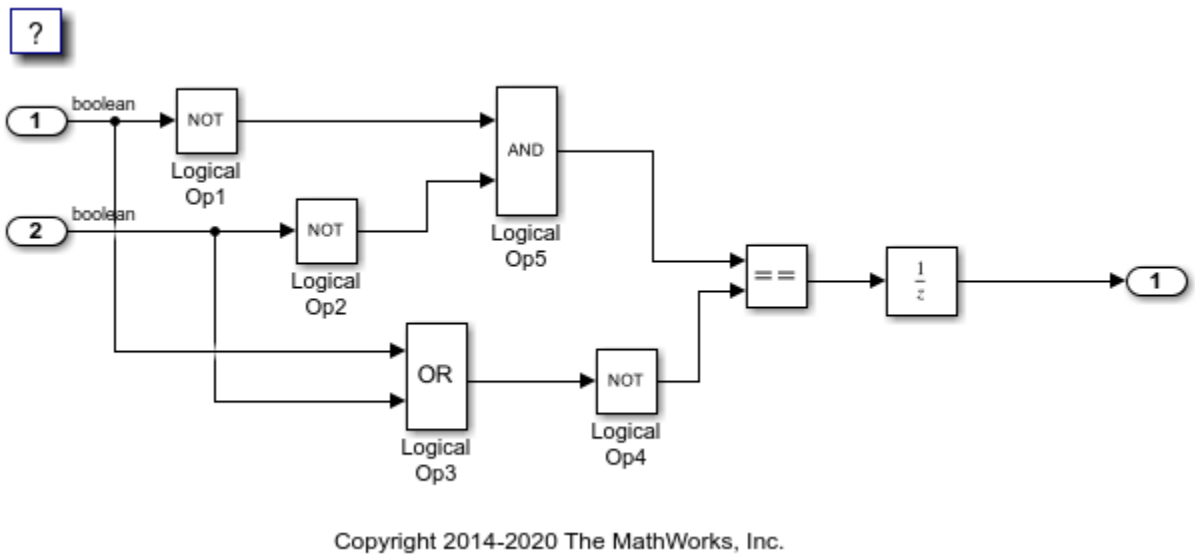
Optimize Generated Code by Packing Boolean Data into Bitfields

This example shows how to optimize the generated code by packing Boolean data into bitfields. When you select the model configuration parameter **Pack Boolean data into bitfields**, Embedded Coder® packs the Boolean signals into 1-bit bitfields, reducing RAM consumption. By default, the optimization is enabled. This optimization reduces the RAM consumption. Be aware that this optimization can potentially increase code size and execution speed.

Example Model

Consider the model `PackBooleanData`.

```
model = 'PackBooleanData';
open_system(model);
```



Disable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, clear **Pack Boolean data into bitfields**.

Alternatively, you can use the command-line API to disable the optimization:

```
set_param(model,'BooleansAsBitfields','off');
```

Generate Code Without Optimization

Build the model using Embedded Coder.

```
slbuild(model)
```

```
### Starting build procedure for: PackBooleanData
### Successful completion of build procedure for: PackBooleanData
```

```
Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
==========================================================================================
PackBooleanData  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.988s
```

View the generated code without the optimization. These lines of code are in `PackBooleanData.h`.

```
hfile = fullfile('PackBooleanData_ert_rtw','PackBooleanData.h');
rtwdemodbtype(hfile,'/* Block signals and states','/* External inputs',1,0);
```

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  boolean_T LogicalOp1;                /* '<Root>/Logical Op1' */
  boolean_T LogicalOp2;                /* '<Root>/Logical Op2' */
  boolean_T LogicalOp5;                /* '<Root>/Logical Op5' */
  boolean_T LogicalOp3;                /* '<Root>/Logical Op3' */
  boolean_T LogicalOp4;                /* '<Root>/Logical Op4' */
  boolean_T RelationalOperator;        /* '<Root>/Relational Operator' */
  boolean_T UnitDelay_DSTATE;          /* '<Root>/Unit Delay' */
} DW;
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Pack Boolean data into bitfields**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'BooleansAsBitfields','on');
```

Generate Code with Optimization

Build the model using Embedded Coder.

```
slbuild(model)
```

```
### Starting build procedure for: PackBooleanData
### Successful completion of build procedure for: PackBooleanData

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
==============================================================================
PackBooleanData  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.067s
```

View the generated code with the optimization. These lines of code are in `PackBooleanData.h`.

```
hfile = fullfile('PackBooleanData_ert_rtw','PackBooleanData.h');
rtwdemodbtype(hfile,'/* Block signals and states','/* External inputs',1,0);
```

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  struct {
    uint_T LogicalOp1:1;               /* '<Root>/Logical Op1' */
    uint_T LogicalOp2:1;               /* '<Root>/Logical Op2' */
    uint_T LogicalOp5:1;               /* '<Root>/Logical Op5' */
    uint_T LogicalOp3:1;               /* '<Root>/Logical Op3' */
    uint_T LogicalOp4:1;               /* '<Root>/Logical Op4' */
    uint_T RelationalOperator:1;       /* '<Root>/Relational Operator' */
    uint_T UnitDelay_DSTATE:1;         /* '<Root>/Unit Delay' */
  } bitsForTID0;
} DW;
```

Selecting **Pack Boolean data into bitfields** enables model configuration parameter **Bitfield declarator type specifier**. To optimize your code further, select `uchar_t`. However, the optimization benefit of the **Bitfield declarator type specifier** setting depends on your choice of target.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
```

See Also

Pack Boolean data into bitfields

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Optimize Generated Code Using Boolean Data for Logical Signals” on page 76-89
- “Replace boolean with Specific Integer Data Type” on page 79-12
- “Data Types Supported by Simulink”

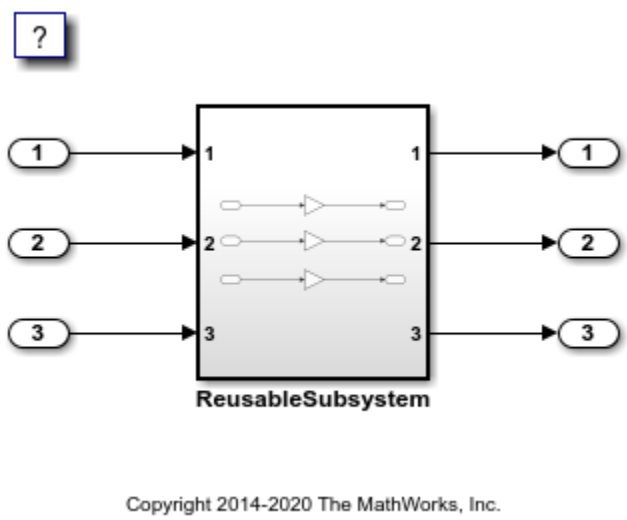
Optimize Generated Code by Passing Reusable Subsystem Outputs as Individual Arguments

This example shows how passing reusable subsystem outputs as individual arguments instead of as a pointer to a structure stored in global memory optimizes the generated code. This optimization conserves RAM consumption and increases code execution speed by reducing global memory usage and eliminating data copies from local variables back to global block I/O structures.

Example Model

Consider the model rtwdemo_reusable_sys_outputs. In this model, the reusable subsystem outputs feed the root outputs of the model.

```
model = 'rtwdemo_reusable_sys_outputs';
open_system(model);
```



Generate Code Without This Optimization

Generate code for this model while passing subsystem outputs as a structure reference. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_reusable_sys_outputs
### Successful completion of build procedure for: rtwdemo_reusable_sys_outputs

Build Summary

Top model targets built:
```

```
Model                          Action                        Rebuild Reason
==========================================================================================
rtwdemo_reusable_sys_outputs  Code generated and compiled.  Code generation information file does

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.537s
```

The code snippet shows portions of `rtwdemo_reusable_sys_outputs.c`. Notice the global block I/O structure and in the model step function a data copy from this structure.

```
cfile = fullfile(cgDir,'rtwdemo_reusable_sys_outputs_ert_rtw',...
'rtwdemo_reusable_sys_outputs.c');
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
'/* Model initialize', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/ReusableSubsystem' */
static void ReusableSubsystem(real_T rtu_In1, real_T rtu_In2, real_T rtu_In3,
  DW_ReusableSubsystem *localDW)
{
  /* Gain: '<S1>/Gain' */
  localDW->Gain = 5.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' */
  localDW->Gain1 = 6.0 * rtu_In2;

  /* Gain: '<S1>/Gain2' */
  localDW->Gain2 = 7.0 * rtu_In3;
}

/* Model step function */
void rtwdemo_reusable_sys_outputs_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/ReusableSubsystem' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   */
  ReusableSubsystem(rtU.In1, rtU.In2, rtU.In3, &rtDW.ReusableSubsystem_d);

  /* End of Outputs for SubSystem: '<Root>/ReusableSubsystem' */

  /* Outport: '<Root>/Out1' */
  rtY.Out1 = rtDW.ReusableSubsystem_d.Gain;

  /* Outport: '<Root>/Out2' */
  rtY.Out2 = rtDW.ReusableSubsystem_d.Gain1;

  /* Outport: '<Root>/Out3' */
  rtY.Out3 = rtDW.ReusableSubsystem_d.Gain2;
}
```

Enable This Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, set **Pass reusable subsystem outputs as** to `Individual arguments`.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'PassReuseOutputArgsAs', 'Individual arguments');
```

Generate Code with This Optimization

With this optimization, the `ReusableSubsystem` function has three output arguments, which are direct references to the external outputs. The `rtDW` global structure no longer exists, and the data copies from this structure to the `rtY` (external outputs) structure are not in the generated code.

Build the model.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_reusable_sys_outputs
### Successful completion of build procedure for: rtwdemo_reusable_sys_outputs

Build Summary

Top model targets built:

Model                          Action                        Rebuild Reason
========================================================================================
rtwdemo_reusable_sys_outputs  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.229s
```

The code snippet below is a portion of `rtwdemo_reusable_sys_outputs.c`. Observe the optimized code.

```
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
'/* Model initialize', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/ReusableSubsystem' */
static void ReusableSubsystem(real_T rtu_In1, real_T rtu_In2, real_T rtu_In3,
  real_T *rty_Out1, real_T *rty_Out2, real_T *rty_Out3)
{
  /* Gain: '<S1>/Gain' */
  *rty_Out1 = 5.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' */
  *rty_Out2 = 6.0 * rtu_In2;

  /* Gain: '<S1>/Gain2' */
  *rty_Out3 = 7.0 * rtu_In3;
}

/* Model step function */
void rtwdemo_reusable_sys_outputs_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/ReusableSubsystem' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   *  Outport: '<Root>/Out1'
```

```
   *  Outport: '<Root>/Out2'
   *  Outport: '<Root>/Out3'
   */
  ReusableSubsystem(rtU.In1, rtU.In2, rtU.In3, &rtY.Out1, &rtY.Out2, &rtY.Out3);

  /* End of Outputs for SubSystem: '<Root>/ReusableSubsystem' */
}
```

Close the model and cleanup.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Pass reusable subsystem outputs as

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Virtualized Output Ports Optimization” on page 78-15
- “Reduce Global Variables in Nonreusable Subsystem Functions” on page 80-8
- “Data Structures in the Generated Code” on page 36-31

Convert Data Copies to Pointer Assignments

The code generator optimizes generated code for vector signal assignments by trying to replace `for` loop controlled element assignments and `memcpy` function calls with pointer assignments. Pointer assignments avoid expensive data copies. Therefore, they use less stack space and offer faster execution speed than `for` loop controlled element assignments and `memcpy` function calls. If you assign large data sets to vector signals, this optimization can result in significant improvements to code efficiency.

Configure Model to Optimize Generated Code for Vector Signal Assignments

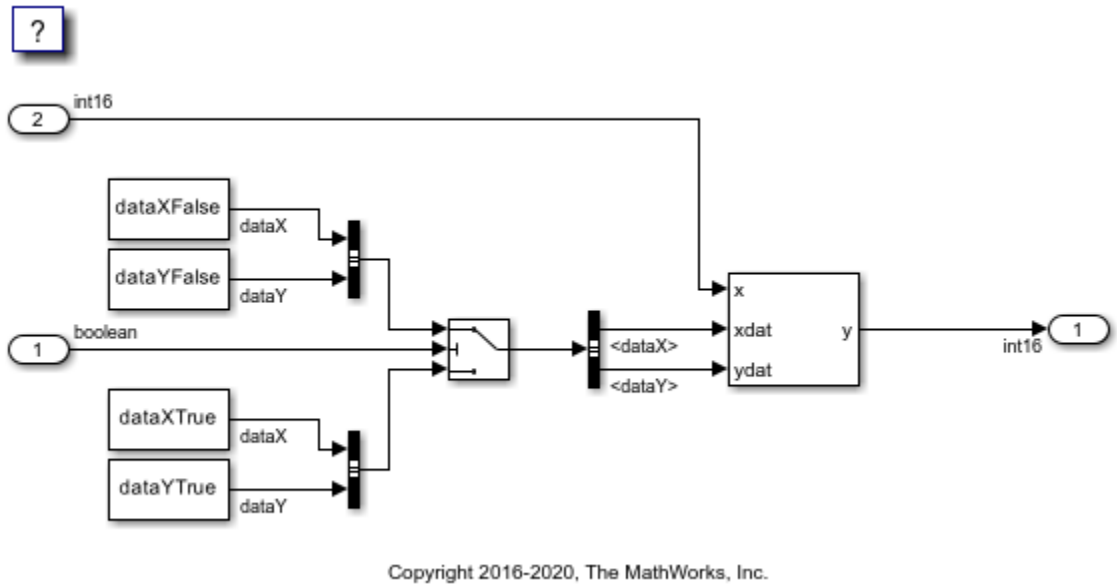
To apply this optimization:

1. Verify that your target supports the `memcpy` function.
2. Determine whether your model uses vector signal assignments (such as `Y=expression`) to move large amounts of data. For example, your model could use a Selector block to select input elements from a vector, matrix, or multidimension signal.
3. On the **Optimization** pane, the **Use memcpy for vector assignment parameter**, which is on by default, enables the associated **Memcpy threshold (bytes)** parameter.
4. Examine the setting of **Memcpy threshold (bytes)**. By default, it specifies 64 bytes as the minimum array size for which `memcpy` function calls or pointer assignments can replace `for` loops in the generated code. Based on the array sizes in your application's vector signal assignments, and target environment considerations on the threshold selection, accept the default value or specify another array size.

Example Model

Open the example model `rtwdemo_pointer_conversion`. This model uses a Switch block to assign data to a vector signal. This signal then feeds into a Bus Selector block.

```
model='rtwdemo_pointer_conversion';
open_system(model);
```



Generate Code Without Optimization

In the Configuration Parameters dialog box, clear the **Use memcpy for vector assignment** parameter. Alternatively, use the command-line.

```
set_param(model, 'EnableMemcpy','off');
```

Press **Ctrl+B** to generate code. Alternatively, use the command-line.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_pointer_conversion
### Successful completion of build procedure for: rtwdemo_pointer_conversion

Build Summary

Top model targets built:

Model                        Action                        Rebuild Reason
==============================================================================================
rtwdemo_pointer_conversion  Code generated and compiled.  Code generation information file does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.392s
```

View the generated code without the optimization. Here is a portion of `rtwdemo_pointer_conversion.c` .

```
cfile = fullfile('rtwdemo_pointer_conversion_ert_rtw','rtwdemo_pointer_conversion.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

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

```
  int32_T i;
  int16_T rtb_dataX[100];
  int16_T rtb_dataY[100];

  /* Switch generated from: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   *  Constant: '<Root>/Constant1'
   *  Constant: '<Root>/Constant2'
   *  Constant: '<Root>/Constant3'
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < 100; i++) {
    if (rtU.In1) {
      rtb_dataX[i] = rtCP_Constant_Value[i];
      rtb_dataY[i] = rtCP_Constant1_Value[i];
    } else {
      rtb_dataX[i] = rtCP_Constant2_Value[i];
      rtb_dataY[i] = rtCP_Constant3_Value[i];
    }
  }

  /* End of Switch generated from: '<Root>/Switch' */

  /* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
   *  Inport: '<Root>/In2'
   *  Outport: '<Root>/Out1'
   */
  /* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
   * Input0  Data Type:  Integer        S16
   * Input1  Data Type:  Integer        S16
   * Input2  Data Type:  Integer        S16
   * Output0 Data Type:  Integer        S16
   * Lookup Method: Linear_Endpoint
   *
   */
  LookUp_S16_S16( &(rtY.Out1), &rtb_dataY[0], rtU.In2, &rtb_dataX[0], 99U);
}
```

Without the optimization, the generated code contains `for` loop controlled element assignments.

Enable Optimization and Generate Code

In the Configuration Parameter dialog box, select the **Use memcpy for vector assignment** parameter.

```
set_param(model, 'EnableMemcpy','on')
```

Generate code.

```
slbuild(model);
```

```
### Starting build procedure for: rtwdemo_pointer_conversion
### Successful completion of build procedure for: rtwdemo_pointer_conversion

Build Summary

Top model targets built:
```

```
Model                        Action                        Rebuild Reason
==========================================================================================
rtwdemo_pointer_conversion  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 11.52s
```

View the generated code without the optimization. Here is a portion of `rtwdemo_pointer_conversion.c`.

```
cfile = fullfile('rtwdemo_pointer_conversion_ert_rtw','rtwdemo_pointer_conversion.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void rtwdemo_pointer_conversion_step(void)
{
  const int16_T *rtb_dataX_0;
  const int16_T *rtb_dataY_0;

  /* Inport: '<Root>/In1' */
  if (rtU.In1) {
    /* Switch generated from: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant'
     */
    rtb_dataX_0 = &rtCP_Constant_Value[0];

    /* Switch generated from: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    rtb_dataY_0 = &rtCP_Constant1_Value[0];
  } else {
    /* Switch generated from: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant2'
     */
    rtb_dataX_0 = &rtCP_Constant2_Value[0];

    /* Switch generated from: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant3'
     */
    rtb_dataY_0 = &rtCP_Constant3_Value[0];
  }

  /* End of Inport: '<Root>/In1' */

  /* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
   *  Inport: '<Root>/In2'
   *  Outport: '<Root>/Out1'
   */
  /* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
   * Input0  Data Type:  Integer        S16
   * Input1  Data Type:  Integer        S16
   * Input2  Data Type:  Integer        S16
   * Output0 Data Type:  Integer        S16
   * Lookup Method: Linear_Endpoint
   *
   */
```

```
  LookUp_S16_S16( &(rtY.Out1), &rtb_dataY_0[0], rtU.In2, &rtb_dataX_0[0], 99U);
}
```

Because the setting of the **Memcpy threshold (bytes)** parameter is below the array sizes in the generated code, the optimized code contains pointer assignments for the vector signal assignments.

```
bdclose(model)
```

See Also

Use memcpy for vector assignment | Memcpy threshold (bytes)

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 76-48
- “Vector Operation Optimization” on page 76-99

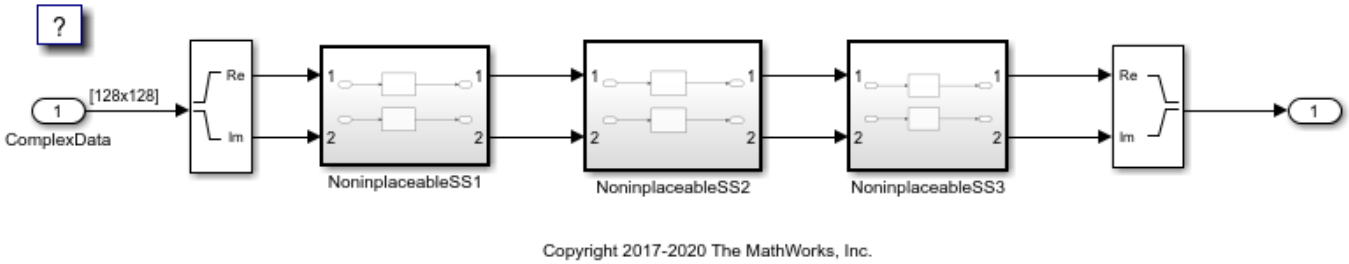
Reuse Buffers of Different Sizes and Dimensions

You can reuse buffers for matrices that have different sizes and shapes. In the Configuration Parameters dialog box, you enable this optimization by selecting **Reuse buffers of different sizes and dimensions**. This optimization conserves RAM and ROM usage and improves code execution speed.

Example Model

The model `DifferentSizeReuse` contains signals of different sizes and dimensions.

```
model='DifferentSizeReuse';
open_system(model);
```



Generate Code Without Optimization

In the Configuration Parameters dialog box, set **Reuse buffers of different sizes and dimension** parameter to `off` or in the MATLAB® Command Window, enter:

```
set_param('DifferentSizeReuse','DifferentSizesBufferReuse','off');
```

Turn off comments and build the model.

```
set_param('DifferentSizeReuse','GenerateComments','off');
slbuild('DifferentSizeReuse');
```

```
### Starting build procedure for: DifferentSizeReuse
### Successful completion of build procedure for: DifferentSizeReuse

Build Summary

Top model targets built:

Model               Action                        Rebuild Reason
========================================================================================
DifferentSizeReuse  Code generated and compiled.  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 17.943s
```

View the generated code without the optimization. The `D_Work` structure is:

```
hfile = fullfile('DifferentSizeReuse_ert_rtw',...
    'DifferentSizeReuse.h');
rtwdemodbtype(hfile,'typedef struct {','} D_Work',1,1);
```

```
typedef struct {
  real_T ComplextoRealImag_o1[16384];
  real_T ComplextoRealImag_o2[16384];
  real_T z[3969];
  real_T z_n[3969];
  real_T z_j[4032];
  real_T z_m[4032];
  real_T z_g[4096];
  real_T z_e[4096];
} D_Work;
```

The portion of `DifferentSizeReuse.c` is:

```
cfile = fullfile('DifferentSizeReuse_ert_rtw',...
    'DifferentSizeReuse.c');
rtwdemodbtype(cfile,'#include "DifferentSizeReuse.h"',...
    'void DifferentSizeReuse_initialize(void)',1,0);
```

```
#include "DifferentSizeReuse.h"
#include "rtwtypes.h"
#include <math.h>

D_Work rtDWork;
ExternalInputs rtU;
ExternalOutputs rtY;
static RT_MODEL rtM_;
RT_MODEL *const rtM = &rtM_;
static void Downsample(const real_T rtu_u[16384], real_T rty_z[4096]);
static void DeltaX(const real_T rtu_u[4096], real_T rty_z[4032]);
static void DeltaY(const real_T rtu_u[4032], real_T rty_z[3969]);
static void NoninplaceableSS1(void);
static void NoninplaceableSS2(void);
static void NoninplaceableSS3(void);
static void Downsample(const real_T rtu_u[16384], real_T rty_z[4096])
{
  int32_T tmp;
  int32_T tmp_0;
  int32_T tmp_1;
  int32_T x;
  int32_T y;
  for (x = 0; x < 64; x++) {
    for (y = 0; y < 64; y++) {
      tmp_0 = (y + 1) << 1;
      tmp_1 = (x + 1) << 1;
      tmp = ((tmp_0 - 2) << 7) + tmp_1;
      tmp_0 = ((tmp_0 - 1) << 7) + tmp_1;
      rty_z[x + (y << 6)] = (((rtu_u[tmp - 2] + rtu_u[tmp - 1]) + rtu_u[tmp_0 -
        2]) + rtu_u[tmp_0 - 1]) / 4.0;
    }
  }
}

static void NoninplaceableSS1(void)
{
  Downsample(rtDWork.ComplextoRealImag_o1, rtDWork.z_e);
  Downsample(rtDWork.ComplextoRealImag_o2, rtDWork.z_g);
```

```
}

static void DeltaX(const real_T rtu_u[4096], real_T rty_z[4032])
{
  int32_T tmp;
  int32_T x;
  int32_T y;
  for (x = 0; x < 63; x++) {
    for (y = 0; y < 64; y++) {
      tmp = (y << 6) + x;
      rty_z[x + 63 * y] = fabs(rtu_u[tmp] - rtu_u[tmp + 1]);
    }
  }
}

static void NoninplaceableSS2(void)
{
  DeltaX(rtDWork.z_e, rtDWork.z_m);
  DeltaX(rtDWork.z_g, rtDWork.z_j);
}

static void DeltaY(const real_T rtu_u[4032], real_T rty_z[3969])
{
  int32_T i;
  for (i = 0; i < 3969; i++) {
    rty_z[i] = fabs(rtu_u[i] - rtu_u[i + 63]);
  }
}

static void NoninplaceableSS3(void)
{
  DeltaY(rtDWork.z_m, rtDWork.z_n);
  DeltaY(rtDWork.z_j, rtDWork.z);
}

void DifferentSizeReuse_step(void)
{
  int32_T i;
  for (i = 0; i < 16384; i++) {
    rtDWork.ComplextoRealImag_o1[i] = rtU.ComplexData[i].re;
    rtDWork.ComplextoRealImag_o2[i] = rtU.ComplexData[i].im;
  }

  NoninplaceableSS1();
  NoninplaceableSS2();
  NoninplaceableSS3();
  for (i = 0; i < 3969; i++) {
    rtY.Out1[i].re = rtDWork.z_n[i];
    rtY.Out1[i].im = rtDWork.z[i];
  }
}
```

The `D_work` structure contains eight global variables for holding the inputs and outputs of `Downsample`, `DeltaX`, and `DeltaY`. These variables have different sizes.

Generate Code with Optimization

1. In the Configuration Parameters dialog box, verify that **Signal storage reuse** is selected.

2 Set the **Reuse buffers of different sizes and dimensions** parameter to `on` or in the MATLAB Command Window, enter:

```
set_param('DifferentSizeReuse','DifferentSizesBufferReuse','on');
```

Build the model.

```
set_param('DifferentSizeReuse','GenerateComments','off');
slbuild(model);
```

```
### Starting build procedure for: DifferentSizeReuse
### Successful completion of build procedure for: DifferentSizeReuse

Build Summary

Top model targets built:

Model               Action                        Rebuild Reason
========================================================================================
DifferentSizeReuse  Code generated and compiled.  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 14.084s
```

View the generated code without the optimization. The `D_Work` structure is:

```
rtwdemodbtype(hfile,'typedef struct {','} D_Work',1,1);
```

```
typedef struct {
  real_T z[4096];
  real_T z_n[16384];
  real_T z_a[16384];
} D_Work;
```

The portion of `DifferentSizeReuse.c` is:

```
cfile = fullfile('DifferentSizeReuse_ert_rtw',...
    'DifferentSizeReuse.c');
rtwdemodbtype(cfile,'#include "DifferentSizeReuse.h"',...
    'void DifferentSizeReuse_initialize(void)',1,0);
```

```
#include "DifferentSizeReuse.h"
#include "rtwtypes.h"
#include <math.h>

D_Work rtDWork;
ExternalInputs rtU;
ExternalOutputs rtY;
static RT_MODEL rtM_;
RT_MODEL *const rtM = &rtM_;
static void Downsample(const real_T rtu_u[16384], real_T rty_z[4096]);
static void DeltaX(const real_T rtu_u[4096], real_T rty_z[4032]);
static void DeltaY(const real_T rtu_u[4032], real_T rty_z[3969]);
static void NoninplaceableSS1(void);
static void NoninplaceableSS2(void);
static void NoninplaceableSS3(void);
static void Downsample(const real_T rtu_u[16384], real_T rty_z[4096])
```

```
{
  int32_T tmp;
  int32_T tmp_0;
  int32_T tmp_1;
  int32_T x;
  int32_T y;
  for (x = 0; x < 64; x++) {
    for (y = 0; y < 64; y++) {
      tmp_0 = (y + 1) << 1;
      tmp_1 = (x + 1) << 1;
      tmp = ((tmp_0 - 2) << 7) + tmp_1;
      tmp_0 = ((tmp_0 - 1) << 7) + tmp_1;
      rty_z[x + (y << 6)] = (((rtu_u[tmp - 2] + rtu_u[tmp - 1]) + rtu_u[tmp_0 -
        2]) + rtu_u[tmp_0 - 1]) / 4.0;
    }
  }
}

static void NoninplaceableSS1(void)
{
  Downsample(rtDWork.z_n, rtDWork.z);
  Downsample(rtDWork.z_a, &rtDWork.z_n[0]);
}

static void DeltaX(const real_T rtu_u[4096], real_T rty_z[4032])
{
  int32_T tmp;
  int32_T x;
  int32_T y;
  for (x = 0; x < 63; x++) {
    for (y = 0; y < 64; y++) {
      tmp = (y << 6) + x;
      rty_z[x + 63 * y] = fabs(rtu_u[tmp] - rtu_u[tmp + 1]);
    }
  }
}

static void NoninplaceableSS2(void)
{
  DeltaX(rtDWork.z, &rtDWork.z_a[0]);
  DeltaX(&rtDWork.z_n[0], &rtDWork.z[0]);
}

static void DeltaY(const real_T rtu_u[4032], real_T rty_z[3969])
{
  int32_T i;
  for (i = 0; i < 3969; i++) {
    rty_z[i] = fabs(rtu_u[i] - rtu_u[i + 63]);
  }
}

static void NoninplaceableSS3(void)
{
  DeltaY(&rtDWork.z_a[0], &rtDWork.z_n[0]);
  DeltaY(&rtDWork.z[0], &rtDWork.z_a[0]);
}

void DifferentSizeReuse_step(void)
```

```
{
  int32_T i;
  for (i = 0; i < 16384; i++) {
    rtDWork.z_n[i] = rtU.ComplexData[i].re;
    rtDWork.z_a[i] = rtU.ComplexData[i].im;
  }

  NoninplaceableSS1();
  NoninplaceableSS2();
  NoninplaceableSS3();
  for (i = 0; i < 3969; i++) {
    rtY.Out1[i].re = rtDWork.z_n[i];
    rtY.Out1[i].im = rtDWork.z_a[i];
  }
}
```

The `D_work` structure now contains three global variables instead of eight global variables for holding the inputs and outputs of `Downsample`, `DeltaX`, and `DeltaY`. The generated code uses these variables to hold the differently-sized inputs and outputs.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
```

Limitations

- If you use a `Reusable` custom storage class to specify reuse on signals that have different sizes and shapes, you must select the **Reuse buffers of different sizes and dimensions** parameter. Otherwise, the model does not build.
- The code generator does not replace a buffer with a lower priority buffer that has a smaller size.
- The code generator does not reuse buffers that have different sizes and symbolic dimensions.

See Also

Reuse buffers of different sizes and dimensions

Related Examples

- “Specify Buffer Reuse for Signals in a Path” on page 78-17

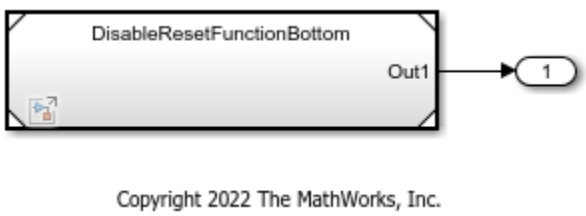
Remove Reset and Disable Functions from the Generated Code

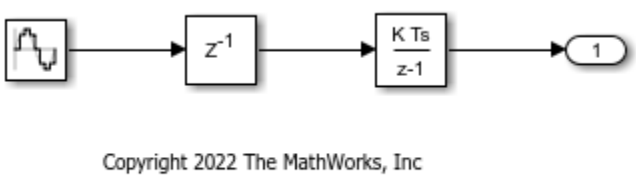
This example shows how the code generator removes unreachable (dead code) instances of the reset and disable functions from the generated code for ERT-based systems that include model referencing hierarchies. Optimizing the generated code to remove unreachable code is a requirement for safety-critical systems. This optimization also improves execution speed and reduces ROM consumption.

If a model contains blocks with states, the generated code contains reset and disable functions. If the model is not part of a conditionally executed system, such as an enabled subsystem, the code generator can remove the disable function because the generated code does not call it. If the model is not part of a conditionally executed system that can reset states when a control input enables it, the code generator can remove the reset function because the generated code does not call it.

Example Model

A referenced model, `DisableResetFunctionBottom`, is in `DisableResetFunctionTop`. The referenced model contains two blocks with states, a Delay block and a Discrete-Time Integrator block.





Generate Code Without This Optimization

Open the `DisableResetFunctionTop` model and generate code.

```
modelTop = 'DisableResetFunctionTop';
open_system(modelTop);
evalc('slbuild(modelTop)');
```

The `DisableResetFunctionBottom.c` file contains these reset and disable functions.

```
cfileReset = fullfile('slprj','ert','DisableResetFunctionBottom','DisableResetFunctionBottom.c')
rtwdemodbtype(cfileReset,'/* System reset for referenced model:', '/* Enable for referenced mode
```

```
/* System reset for referenced model: 'DisableResetFunctionBottom' */
void DisableResetFunctionBottom_Reset(DisableResetFunctionBottom_DW_f *localDW)
{
```

```
  /* InitializeConditions for Delay: '<Root>/Delay' */
  localDW->Delay_DSTATE = 0.0;

  /* InitializeConditions for DiscreteIntegrator: '<Root>/Discrete-Time Integrator' */
  localDW->DiscreteTimeIntegrator_DSTATE = 3.0;
}
```

```
cfileDisable = fullfile('slprj','ert','DisableResetFunctionBottom','DisableResetFunctionBottom.c
rtwdemodbtype(cfileDisable,'/* Disable for referenced model', '/* Output and update for reference
```

```
/* Disable for referenced model: 'DisableResetFunctionBottom' */
void DisableResetFunctionBottom_Disable(real_T *rty_Out1,
  DisableResetFunctionBottom_DW_f *localDW)
{
  /* Disable for DiscreteIntegrator: '<Root>/Discrete-Time Integrator' */
  localDW->DiscreteTimeIntegrator_DSTATE = *rty_Out1;
}
```

The `DisableResetFunctionTop_step` function does not call the `DisableResetFunctionBottom_Disable` function because the model is not part of a conditionally executed system. The `DisableResetFunctionTop_step` function does not call the `DisableResetFunctionBottom_Reset` function because the model is not part of a conditionally executed system that can reset states when a control input enables it.

Generate Code With This Optimization

Open the `DisableResetFunctionBottom` model.

```
modelBottom = 'DisableResetFunctionBottom';
open_system(modelBottom);
```

Open the Configuration Parameters dialog box for `DisableResetFunctionBottom`. On the **Interface** pane, in the **Advanced parameters** section, select the **Remove Disable Function** and **Remove Reset Function** parameters.

Alternatively, use the command-line API to enable the optimization:

```
set_param(modelBottom, 'RemoveResetFunc','on');
set_param(modelBottom, 'RemoveDisableFunc','on');
```

Generate code for the `DisableResetFunctionTop` model.

```
evalc('slbuild(modelTop)');
```

Navigate to the `ert` folder inside the `slprj` folder and open the `DisableResetFunctionBottom.c` file. The code does not contain the DisableResetFunctionBottom_`Reset` function or the DisableResetFunctionBottom_`Disable` function.

Close the models and the code generation report.

```
bdclose(modelTop)
bdclose(modelBottom)
```

See Also

Remove reset function | Remove disable function

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

Control Operator Type in Generated Code

This example shows how to optimize the generated code by controlling the operator type in the code. You can generate code with Bitwise or Logical operators or a combination of both for inputs with Boolean data type. Selecting bitwise operators in the code might improve ROM efficiency when the code contains bitwise operators.

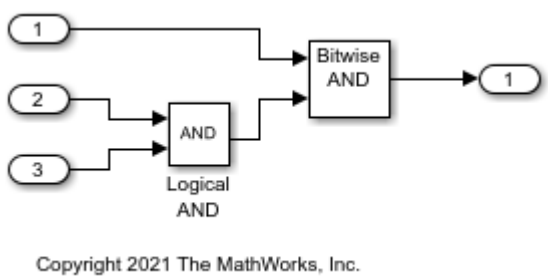
When you select a value for the **Operator to represent Bitwise and Logical Operator blocks** parameter, the generated code maintains the operators as modelled or converts all operators into either bitwise or logical operators independent of the modeling blocks.

NOTE: This example requires Embedded Coder®.

Example Model

In the model matlab:logicalandbitwise, a Logical AND block connects to a Bitwise AND block.

```
model = 'logicalandbitwise';
open_system(model);
```



Generate Code as Modeled

Use your system temporary folder for building and inspection process.

```
currentDir = pwd;
```

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, set the **Operator to represent Bitwise and Logical Operator blocks** parameter to `Same as modeled`.

Alternatively, use the command-line API to set the parameter value:

```
set_param(model, 'BitwiseOrLogicalOp', 'Same as modeled');
evalc('slbuild(model)');
```

View the generated code with operators in the code that matches the modeled blocks. Here is a portion of `logicalandbitwise.c`.

```
cfile = fullfile(currentDir,'logicalandbitwise_ert_rtw','logicalandbitwise.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void logicalandbitwise_step(void)
```

```
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/Input'
   *  Inport: '<Root>/Input1'
   *  Inport: '<Root>/Input2'
   *  Logic: '<Root>/Logical AND'
   *  S-Function (sfix_bitop): '<Root>/Bitwise AND'
   */
  logicalandbitwise_Y.Out1 = (logicalandbitwise_U.Input1 &&
    logicalandbitwise_U.Input2) & logicalandbitwise_U.Input;
}
```

The generated code contains both a logical && operator and a bitwise & operator representing the Logical AND block and Bitwise AND block in the code.

Generate Code with Logical Operators

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, set the **Operator to represent Bitwise and Logical Operator blocks** parameter to `Logical operator`.

Alternatively, use the command-line API to set the parameter value:

```
set_param(model, 'BitwiseOrLogicalOp', 'Logical operator');
evalc('slbuild(model)');
```

View the generated code with only logical operators. Here is a portion of `logicalandbitwise.c`.

```
cfile = fullfile(currentDir,'logicalandbitwise_ert_rtw','logicalandbitwise.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void logicalandbitwise_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/Input'
   *  Inport: '<Root>/Input1'
   *  Inport: '<Root>/Input2'
   *  Logic: '<Root>/Logical AND'
   *  S-Function (sfix_bitop): '<Root>/Bitwise AND'
   */
  logicalandbitwise_Y.Out1 = (logicalandbitwise_U.Input &&
    (logicalandbitwise_U.Input1 && logicalandbitwise_U.Input2));
}
```

The generated code contains only the logical && operator. The Bitwise AND block in the model is converted into the logical && operator in the code.

Generate Code with Bitwise Operators

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, set the **Operator to represent Bitwise and Logical Operator blocks** parameter to `Bitwise operator`.

Alternatively, use the command-line API to set the parameter value:

```
set_param(model, 'BitwiseOrLogicalOp', 'Bitwise operator');
evalc('slbuild(model)');
```

View the generated code with only bitwise operators. Here is a portion of `logicalandbitwise.c`.

```
cfile = fullfile(currentDir,'logicalandbitwise_ert_rtw','logicalandbitwise.c');
rtwdemodbtype(cfile,'/* Model step function ','/* Model initialize function',1, 1);
```

```
/* Model step function */
void logicalandbitwise_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/Input'
   *  Inport: '<Root>/Input1'
   *  Inport: '<Root>/Input2'
   *  Logic: '<Root>/Logical AND'
   *  S-Function (sfix_bitop): '<Root>/Bitwise AND'
   */
  logicalandbitwise_Y.Out1 = (boolean_T)(logicalandbitwise_U.Input1 &
    logicalandbitwise_U.Input2) & logicalandbitwise_U.Input;
}
```

The generated code contains only the bitwise & operator. The Logical AND block in the model is converted into the bitwise & operator in the code.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

Operator to represent Bitwise and Logical Operator blocks | “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7

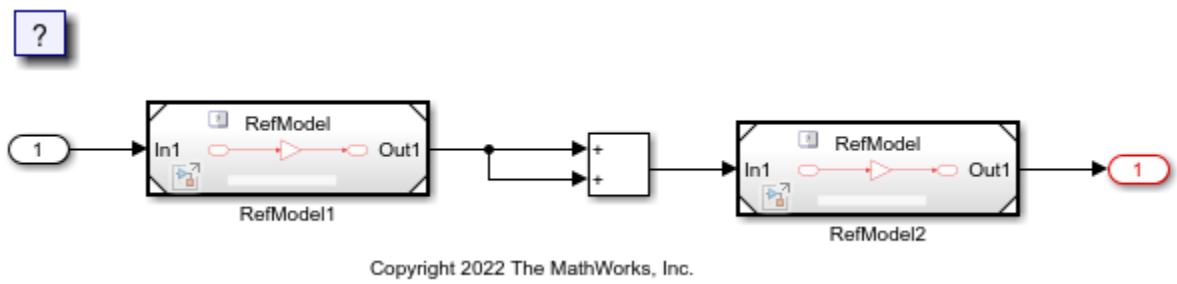
Reduce Memory Usage for Models Containing Referenced Models

This example shows how you can generate RAM efficient code for models containing referenced models that have no function prototype control specifications. When you select the configuration parameter **Reuse output buffers of Model blocks**, the code generator analyzes the referenced model contents and interacts with other optimizations to determine if buffer reuse is possible. If buffer reuse is possible, the code generator tries to reuse existing signal memory or generates reusable temporary buffers to hold referenced model outputs. Reusing buffers can significantly reduce RAM consumption.

Open Example Model

Open the example model `RefBufferReuse` containing two instances of the referenced model `RefModel`, which does not have a function prototype control specification.

```
model = 'RefBufferReuse';
open_system(model)
Rmodel= 'RefModel';
load_system(Rmodel);
```



Generate Code Without This Buffer Reuse Optimization

Open the configuration parameters for the model. Clear the **Reuse output buffers of Model blocks** parameter check box. Alternatively, use the command line.

```
set_param(model, 'ReuseModelBlockBuffer', 'off');
```

Open the configuration parameters for the referenced model. Select the **Reuse output buffers of Model blocks** parameter check box and save the changes. Alternatively, use the command line.

```
set_param(Rmodel, 'ReuseModelBlockBuffer', 'off');
save_system(Rmodel);
```

Build the model.

```
slbuild(model);
```

```
### Starting serial model reference code generation build.
### Starting build procedure for: RefModel
### Successful completion of build procedure for: RefModel
### Starting build procedure for: RefBufferReuse
### Successful completion of build procedure for: RefBufferReuse
```

```
Build Summary

Code generation targets built:

Model     Action                        Rebuild Reason
========================================================================
RefModel  Code generated and compiled.  RefModel.c does not exist.

Top model targets built:

Model           Action                        Rebuild Reason
========================================================================================================
RefBufferReuse  Code generated and compiled.  Code generation information file does not exist.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 24.794s
```

Inspect the generated code.

```
file = fullfile('RefBufferReuse_ert_rtw','RefBufferReuse.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void RefBufferReuse_step(void)
{
  /* local block i/o variables */
  real_T rtb_RefModel1[100];
  real_T rtb_Add[100];
  real_T rtb_RefModel1_0;
  int32_T i;

  /* ModelReference: '<Root>/RefModel1' incorporates:
   *  Inport: '<Root>/In1'
   */
  RefModel(&rtU.In1[0], &rtb_RefModel1[0]);

  /* Sum: '<Root>/Add' */
  for (i = 0; i < 100; i++) {
    rtb_RefModel1_0 = rtb_RefModel1[i];
    rtb_Add[i] = rtb_RefModel1_0 + rtb_RefModel1_0;
  }

  /* End of Sum: '<Root>/Add' */

  /* ModelReference: '<Root>/RefModel2' incorporates:
   *  Outport: '<Root>/Out1'
   */
  RefModel(&rtb_Add[0], &rtY.Out1[0]);
}
```

The generated code generates the unique `rtb_Model1` buffer to hold the output of `RefModel1`.

Generate Code With This Buffer Reuse Optimization

Open the configuration parameters for the model. Select the **Reuse output buffers of Model blocks** parameter check box. Alternatively, use the command line.

```
set_param(model, 'ReuseModelBlockBuffer', 'on');
```

Open the configuration parameters for the referenced model. Select the **Reuse output buffers of Model blocks** parameter check box and save the changes. Alternatively, use the command line.

```
set_param(Rmodel, 'ReuseModelBlockBuffer', 'on');
save_system(Rmodel);
```

Build the model.

```
slbuild(model);
```

```
### Starting serial model reference code generation build.
### Starting build procedure for: RefModel
### Successful completion of build procedure for: RefModel
### Starting build procedure for: RefBufferReuse
### Successful completion of build procedure for: RefBufferReuse

Build Summary

Code generation targets built:

Model     Action                        Rebuild Reason
==============================================================================
RefModel  Code generated and compiled.  Model or library RefModel changed.

Top model targets built:

Model           Action                        Rebuild Reason
================================================================================
RefBufferReuse  Code generated and compiled.  Referenced models were updated.

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.047s
```

Inspect the generated code.

```
file = fullfile('RefBufferReuse_ert_rtw','RefBufferReuse.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function',1,1);
```

```
/* Model step function */
void RefBufferReuse_step(void)
{
  real_T rtb_Add[100];
  real_T rtb_Add_0;
  int32_T i;

  /* ModelReference: '<Root>/RefModel1' incorporates:
   *  Inport: '<Root>/In1'
   */
  RefModel(&rtU.In1[0], &rtb_Add[0]);

  /* Sum: '<Root>/Add' */
  for (i = 0; i < 100; i++) {
    rtb_Add_0 = rtb_Add[i];
    rtb_Add[i] = rtb_Add_0 + rtb_Add_0;
  }
```

```
  /* End of Sum: '<Root>/Add' */

  /* ModelReference: '<Root>/RefModel2' incorporates:
   *  Outport: '<Root>/Out1'
   */
  RefModel(&rtb_Add[0], &rtY.Out1[0]);
}
```

The generated code reuses the `rtb_Add` buffer to hold the output of `RefModel1`.

Clean Up Example Folders and Files

Close the model and remove generated files.

```
bdclose(model);
rtwdemoclean;
```

See Also

Reuse output buffers of Model blocks

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Reuse Global Block Outputs in the Generated Code” on page 78-12

Optimize Generated Code Using Code Efficiency Tools and Techniques

When generating production code from your model to deploy into a real-time embedded system, code efficiency is an important consideration. Based on your target hardware and application, you might need to deploy code that is faster and memory efficient. To accommodate your efficiency needs, Embedded Coder provides optimization parameters that you can set to implement certain optimizations in the generated code.

However, if the efficiency of the generated code does not meet your requirements, review the tips and techniques discussed below and choose an approach for your model.

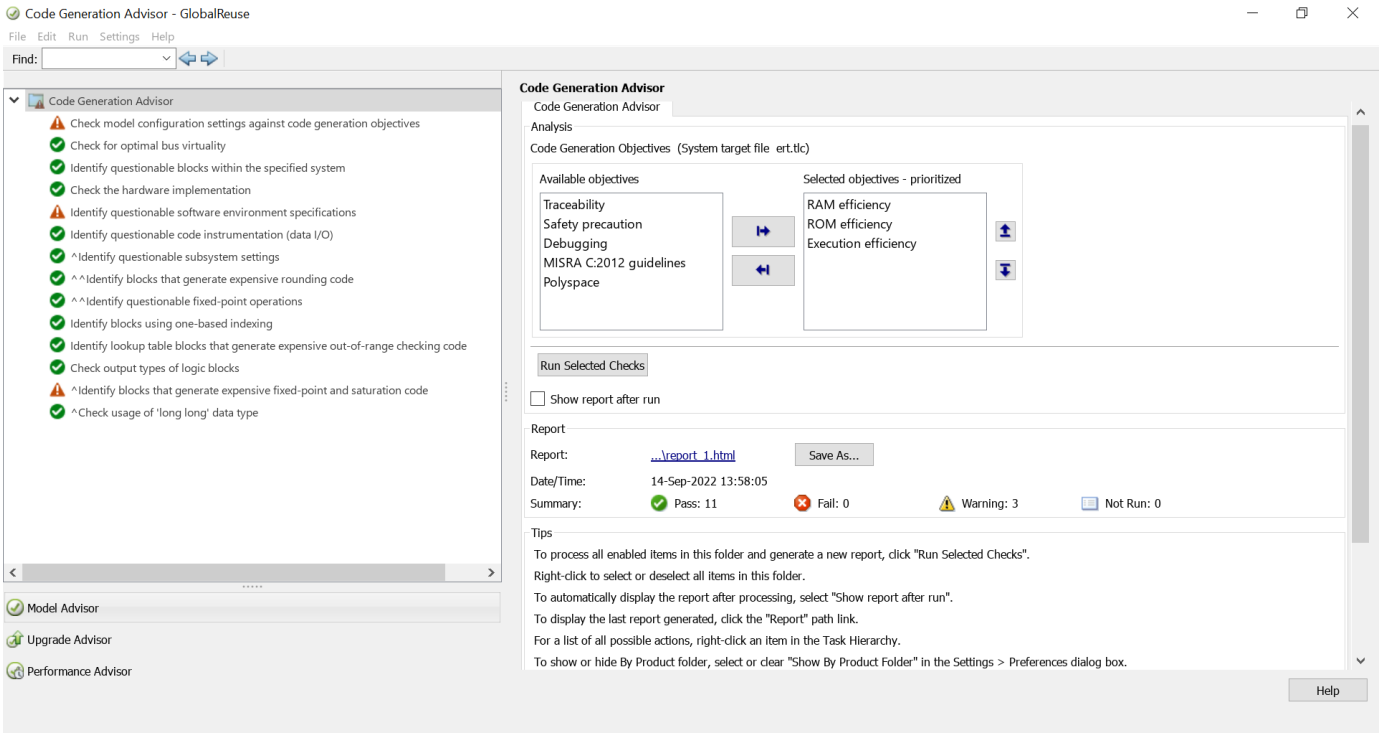
- “Techniques to Refine Models for Efficient Code Generation” on page 80-39
- “Techniques to Identify Areas for Optimizing Memory Usage” on page 80-42
- “Techniques to Optimize Code for Memory Efficiency” on page 80-44
- “Techniques to Identify Areas for Optimizing Code Execution-Time” on page 80-45
- “Explore Parallelization Opportunities” on page 80-46

Techniques to Refine Models for Efficient Code Generation

Optimal parameter settings, efficient modeling patterns, and specifying an appropriate optimization goal are prerequisites for efficient code generation. Use the design techniques discussed in “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7 while designing your model. After developing your model, refine it focusing on code efficiency goal by using the following techniques.

Refine Parameter Settings Based on Code Efficiency Objectives

Use the Code Generation Advisor to analyze your model for code efficiency objectives and identify aspects of the model that do not impede the objectives. You can select multiple objectives and can prioritize them. The Code Generation Advisor checks the model and provides recommended actions for configuration and block parameter settings that can limit efficiency objectives.

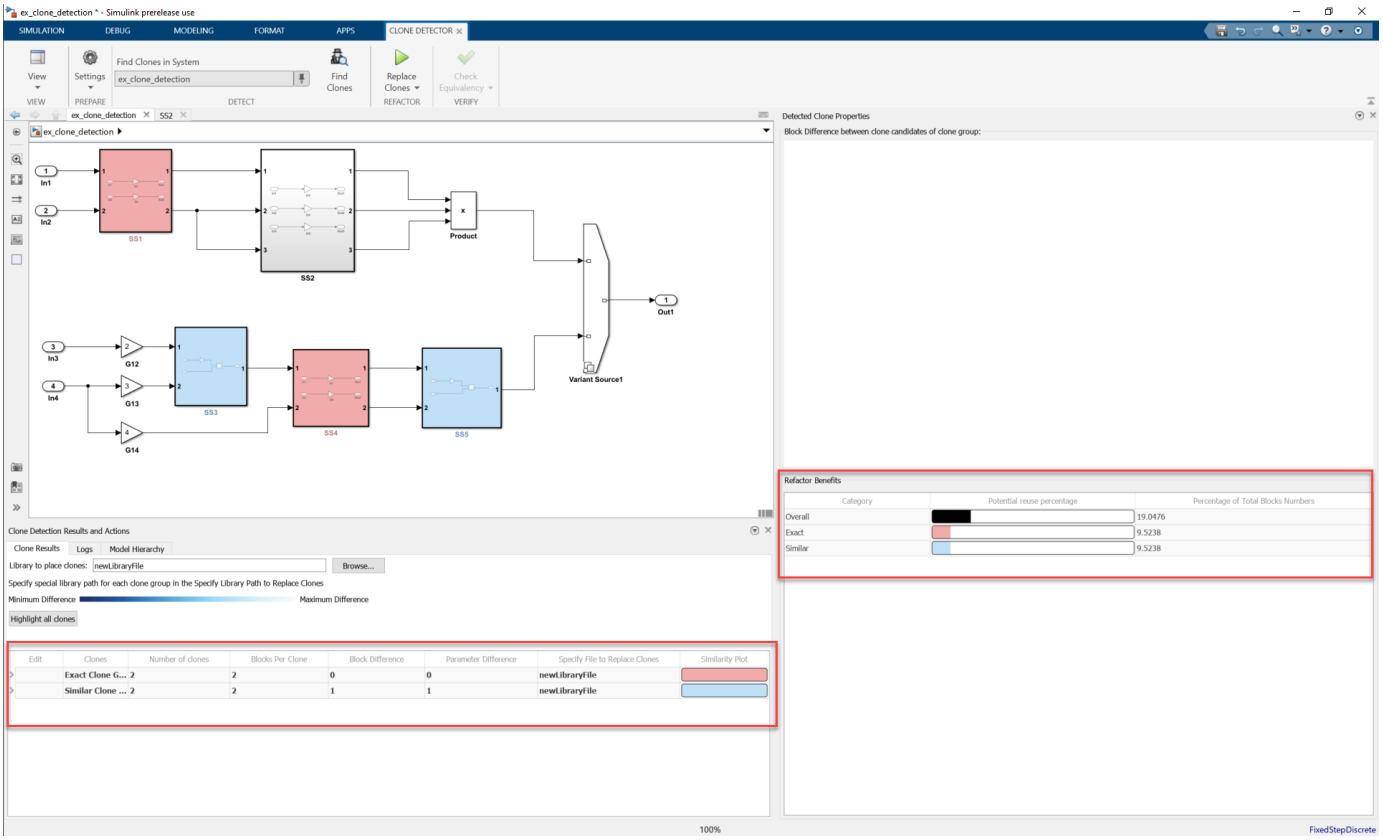


For more information, see “Configure Model for Code Generation Objectives by Using Code Generation Advisor” on page 49-2.

Identify and Refactor Inefficient Modeling Patterns

Inefficient modeling patterns in your model can lead to inefficient code generation. Consider using the refactoring tools **Clone Detector** and **Model Transformer** to identify the inefficient modeling patterns that qualify for automatic replacement with more efficient patterns.

To identify clones in your model that have identical block types and connections, use **Clone Detector**. After analysis, the tool displays the percentage of clones present in the model. If your model contains a high percentage of clones, refactoring can be beneficial from the efficiency standpoint. Using the tool, you can automatically create library blocks for repeated clone patterns and then replace the clones with links to the library blocks.



For example, the `ex_clone_detection` model contains 19.047% overall clones and refactoring the model for all clone groups can reduce 19.047% of the model reuse. For more information, see “Enable Component Reuse by Using Clone Detection” (Simulink Check).

Similarly, using **Model Transformer**, you can identify and:

- Transform qualified modeling patterns into Variant Source or Variant Subsystem blocks. Using Variant Subsystem and Variant Source blocks improves the reusability of a model for different variant choices. For more information, see “Transform Model to Variant System” (Simulink Check).
- Replace qualified Data Store blocks with either a direct signal line, a Delay block, or a Merge block to reduce model complexity. For more information, see “Improve Model Readability by Eliminating Local Data Store Blocks” (Simulink Check).
- Convert qualified multiple n-D Lookup Table blocks into shared Prelookup blocks and multiple Interpolation blocks. Eliminating the redundant Prelookup blocks can improve code efficiency. For more information, see “Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks” (Simulink Check).
- Convert qualified multiple Interpolation Using Prelookup blocks into a single Interpolation Using Prelookup block. Reducing the number of Interpolation Using Prelookup blocks reduces the number of variable assignments in the code, which improves code efficiency. For more information, see “Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks” (Simulink Check).

The refactored model has the same functionality as the original model. To verify the equivalence, use Simulink Test.

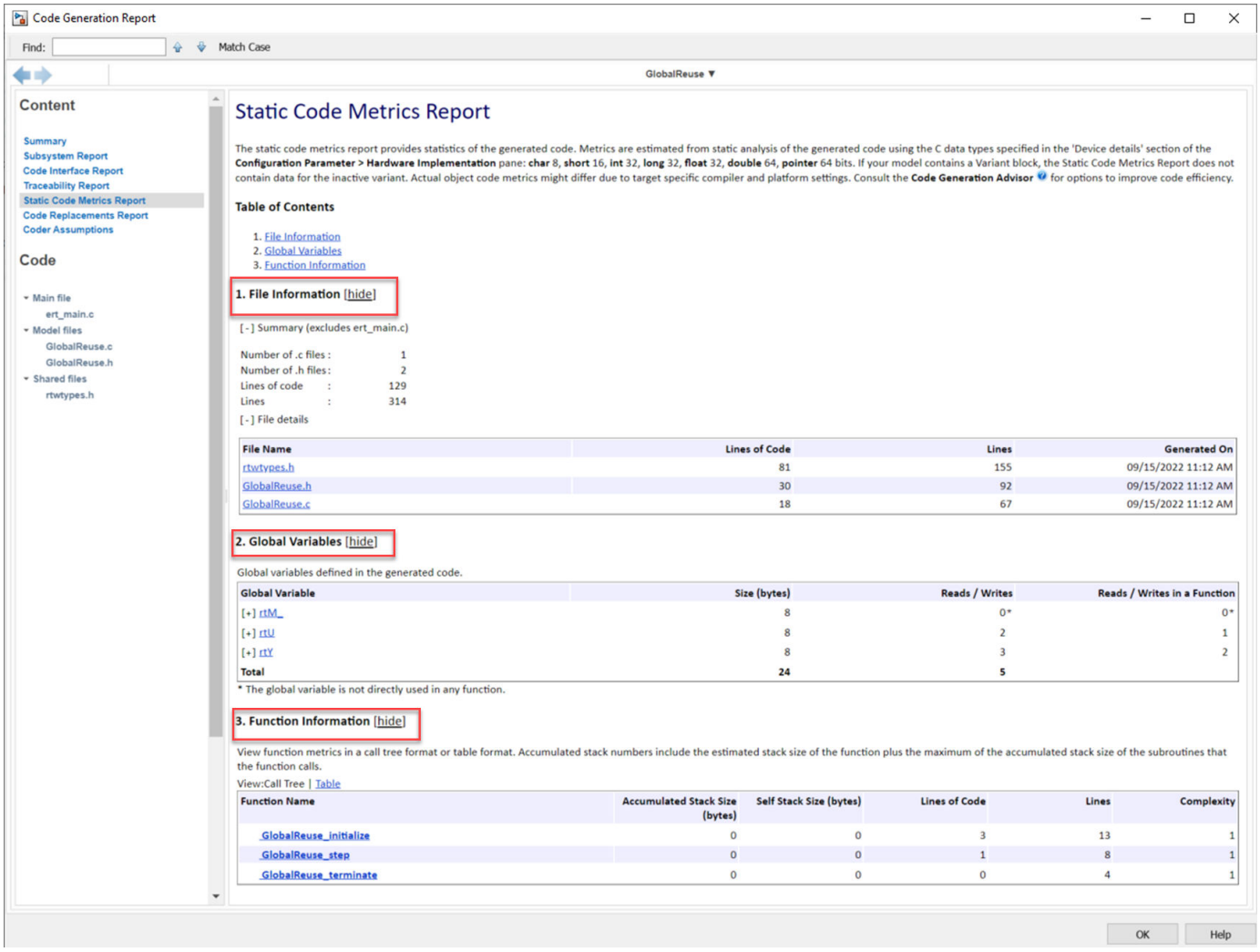
Specify Optimization Level and Priority

By setting the **Level** and **Priority** configuration parameters, you can optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between execution efficiency and RAM consumption. To implement the desired optimization specified in the **Priority** and **Level**, the code generator automatically sets values for the parameters in the **Details** section. For more information on how the parameters in **Details** are configured for different optimization goals, see “Tips”.

Techniques to Identify Areas for Optimizing Memory Usage

If the generated code does not meet your memory requirements or contains unnecessary data copies, you can identify areas where further memory optimization is possible by using these techniques:

- Generate code with bidirectional traceability: Studying the generated code using “Model-to-Code Traceability” on page 85-10 and “Code-to-Model Traceability” on page 85-8 can help you identifying the root cause of the code inefficiency. Also, you can investigate whether a data copy is necessary or whether you can remove it from the generated code by using one of the techniques discussed in “Techniques to Optimize Code for Memory Efficiency” on page 80-44.
- Generate a static metric report: The code generator performs static analysis on the generated code. When you select the **Generate static code metrics** parameter, the code generator provides the static metrics in the **Static Code Metrics Report** section of the HTML code generation report.



From the report, you can:

- Estimate the number of lines of code and stack usage per function
- Review the global variables, their sizes, and the number accesses.
- Determine possible performance slow points, such as the largest global variables or the costliest call path in terms of stack usage.
- View the function call tree. To confirm whether the generated code meets your execution-time requirements, generate execution-time profiling. For more information, see “Techniques to Identify Areas for Optimizing Code Execution-Time” on page 80-45.
- Navigate from the report to the source code.

Reviewing the report, you can determine which global variables and function call paths affect code efficiency. For more information on the static analysis of the generated code, see “Static Code Metrics” on page 54-26 and “Generate Static Code Metrics Report for Simulink Model” on page 54-29.

Techniques to Optimize Code for Memory Efficiency

After studying the generated code and the static code metrics report and identifying areas where you think buffer reuse is possible, consider using these techniques to implement the reuse in the code.

Fine-Tune Configuration Settings

If setting the **Level** and **Priority** parameters for increased execution efficiency does not satisfy your RAM efficiency requirements, select these optimization parameters in the **Details** section after selecting the **Specify custom optimizations** check box.

Parameter	Optimization	More Information
Reuse buffers of different sizes and dimensions	Reuse buffers to store data of different sizes and dimensions.	“Reuse Buffers of Different Sizes and Dimensions” on page 80-23
Optimize block operation order in the generated code	Reorder block operations in the generated code.	“Remove Data Copies by Reordering Block Operations in the Generated Code” on page 78-27
Reuse output buffers of Model blocks	Reuse referenced model output buffers.	“Reduce Memory Usage for Models Containing Referenced Models” on page 80-35

You can also select these **Advanced parameters** to improve ROM efficiency:

- **Remove code from floating-point to integer conversions with saturation that maps NaN to zero**
- **Remove code from floating-point to integer conversions that wraps out-of-range values**
- **Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values**
- **Remove code that protects against division arithmetic exceptions**
- **Optimize using the specified minimum and maximum values**
- **Operator to represent Bitwise and Logical Operator blocks**

The first four advanced parameters are for controlling the generation of protective code that guards against a possible mismatch between simulation and code generation results. Investigate the tradeoffs while configuring those parameters.

Optimize Buffers at Block Boundaries

If you see unnecessary buffers generated at block boundaries, use these optimization techniques to create instances of buffer reuse.

Optimization Techniques	Purpose	More Information
Use function prototype control specifications for model reference interfaces.	Customize referenced model function interfaces to use the same variable for input and output.	“Configure Name and Arguments for Individual Step Functions” on page 33-120

Optimization Techniques	Purpose	More Information
Use the bus data type as block input and output.	Use inplace operations at block boundaries to create instances of buffer reuse.	“Reduced data copies for blocks with bus inputs and outputs”
Use the `Reusable` storage class across block boundaries.	Specify buffer reuse on signals across block boundaries.	“Specify Buffer Reuse for Signals in a Path” on page 78-17
Use the same variable name for the input and output arguments of MATLAB Function blocks.	Specify buffer reuse across MATLAB Function blocks.	“Specify Buffer Reuse for MATLAB Function Blocks in a Path” on page 78-25
Use inplace interface for S-function blocks.	Specify buffer reuse across S-function blocks.	“S-Functions That Specify Port Scope and Reusability” on page 13-73

Creating instances of buffer reuse across block boundaries eliminates the generation of unnecessary buffers and data copies, which reduces RAM consumptions.

Optimize to Reuse Signal Buffers

If your model has the optimal parameter settings for reusing buffers and removing data copies, you might be able to create additional instances of signal buffer reuse by using these optimization techniques:

Optimization Techniques	Purpose	More Information
Use signal labels to guide buffer reuse.	Add labels to signal lines for which you think buffer reuse is possible. The code generator reorders block operations to implement the reuse specification. This optimization improves execution speed and might reduce RAM consumption.	“Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse” on page 78-39
Use `Reusable` storage class to specify buffer reuse for multiple signals in a path.	Specify buffer reuse on signals. This optimization reduces ROM and RAM consumption because there are less global variables and data copies in the generated code.	“Specify Buffer Reuse for Signals in a Path” on page 78-17

Techniques to Identify Areas for Optimizing Code Execution-Time

If you are running SIL and PIL simulations for the generated code, configure the simulations to produce execution-time metrics for tasks and functions in your code. Using the execution-time profiling, you can:

- Determine whether the generated code meets execution-time requirements for real-time deployment on your target hardware.
- If code execution overruns, look for ways to reduce execution time. Identify the tasks that require the most time. For those tasks, investigate whether tradeoffs between functionality and speed are

possible. If your target is a multicore processor, consider parallelizing your code.. For more information, see “Explore Parallelization Opportunities” on page 80-46.

For more information about generating execution-time profiling, see “Create Execution-Time Profile for Generated Code” on page 81-2.

Explore Parallelization Opportunities

When you generate code from a model containing a MATLAB Function, a MATLAB System, and the For Each Subsystem block, by default, the code generator produces code that implements `for`-loops in a single thread. If you are using a compiler that supports the Open Multiprocessing (OpenMP) application interface and have the Parallel Computing Toolbox™ installed, you can use the **Generate parallel for loops** parameter to generate the for-loops as parallel for-loops. The iterations of the `parfor`-loop run in parallel on multiple cores in the target hardware, which improves the execution speed of the generated code. However, generating `parfor`-loops for small loops can be inefficient, as parallelization introduces overheads, which include time for thread creation, data synchronization between threads, and thread deletion. To make parallelization effective, control the **Loop unrolling threshold**. For more information, see “Unroll Parallel for-Loop That Has Small Number of Iterations” on page 79-28.

Generating single instruction, multiple data (SIMD) code from certain Simulink blocks by using Intel SSE and Intel AVX technology can further improve code efficiency because a single instruction of the SIMD code processes multiple data. For computationally intensive operations on SIMD supported blocks, SIMD intrinsics can significantly improve the execution performance of the generated code on Intel platforms. For more information, see “Generate SIMD Code from Simulink Blocks” on page 76-105.

See Also

Related Examples

- “Design Techniques to Optimize Models for Efficient Code Generation” on page 76-7
- “Generate Static Code Metrics Report for Simulink Model” on page 54-29
- “Create Execution-Time Profile for Generated Code” on page 81-2

Code Execution and Stack Usage Profiling in Embedded Coder

- “Create Execution-Time Profile for Generated Code” on page 81-2
- “View and Compare Code Execution Times” on page 81-6
- “Code Execution Profiling Report” on page 81-19
- “Analyze Code Execution Data” on page 81-29
- “Code Execution Profiling From the Command Line” on page 81-31
- “Remove Instrumentation Overheads from Execution Time Measurements” on page 81-32
- “Reduce Bandwidth Usage During Code Execution Profiling” on page 81-37
- “Visualize Task Scheduling in XCP External Mode Simulation” on page 81-40
- “Tips and Limitations” on page 81-50
- “Stack Usage Profiling for Code Generated from Simulink Models” on page 81-53

Create Execution-Time Profile for Generated Code

You can configure a software-in-the-loop (SIL), processor-in-the-loop (PIL) on page 88-2, or XCP-based external mode on page 60-8 simulation to produce execution-time metrics for tasks and functions in your generated code. The software calculates execution times from data that is obtained through code instrumentation added to the target application or the generated code under test. You can use the execution-time metrics to determine whether the generated code meets the requirements for real-time deployment on your target hardware.

For example, you can perform the following analysis:

1. Identify tasks that require the most time. Tasks are main entry points into the generated code. For example, the step function for a sample rate or the `model_initialize` function.
2. In these tasks, investigate code sections that require the most time.
3. Identify variations in execution time over time steps.

If you are trying to reduce execution times, the analysis results help you to focus on the most critical code sections. To observe performance changes for an updated model, rerun the SIL, PIL, or external mode simulation and compare the new metrics against previous metrics.

When the SIL, PIL, or external mode simulation is complete, you can:

- View execution-time metrics through a display panel, report, or in the Code view.
- Use the Simulation Data Inspector to view and compare the variation of execution times over a simulation.
- Analyze the measurements within the MATLAB environment.

Configure Code Execution Profiling

This table lists system target files that are supported for code execution profiling.

Simulation Mode	System Target File
SIL or PIL	GRT, ERT, or ERT-based
XCP-based external	ERT or ERT-based

To configure code execution profiling for a SIL, PIL, or XCP-based external mode simulation:

1. In your top model, open the Configuration Parameters dialog box.
2. Select the **Measure task execution time** check box.
3. For function execution times, from the **Measure function execution times** drop-down list, select one of these options:
   - `Coarse (referenced models and subsystems only)` –– if you want to analyse generated function code for the main model components.
   - `Detailed (all function call sites)` –– if you want to analyse generated function code for all blocks in the model.
4. In the **Workspace variable** field, specify a name. When you run the simulation, the software generates a variable with this name in the MATLAB base workspace. The variable contains the execution-time measurements, and is an object of type `coder.profile.ExecutionTime`.

If the **Data Import/Export** > **Single simulation output** check box is selected, the software creates the variable in the `Simulink.SimulationOutput` object that you specify.

5 From the Save options drop-down list, select one of these options:

- `Summary data only` — Stream execution times to the Simulation Data Inspector during the simulation. Store minimal information at the end of the simulation. Use this option if you want to generate only a report and reduce memory usage, for example, in a long simulation.
- `All data` — Stream execution times to the Simulation Data Inspector during the simulation. Store execution-time profiles in the base workspace and generate a report. After the simulation, you can use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes to retrieve execution-time measurements for every call to each profiled section of code that occurs during the simulation.

  If you select this option, the metrics display window and generated report display the Simulation Data Inspector icon . When you click the icon, the software imports simulation results into the Simulation Data Inspector. You can then plot execution times and manage and compare plots from various simulations.
- `Metrics only` –– Reduces bandwidth usage for the communication channel between Simulink and the target application. During the simulation, the software does not stream execution times to the Simulation Data Inspector. On the target hardware, the software stores maximum execution times, average execution times, and number of calls. At the end of the simulation, Simulink uploads the data from the target hardware to your development computer.

6 Click **OK**.

For a PIL simulation:

- You must configure a hardware-specific timer. When you set up target connectivity, create a timer object. This action is not required for a SIL simulation. For more information, see “Set Up PIL Connectivity by Using target Package” on page 88-48 or “Create PIL Target Connectivity Configuration for Simulink” on page 88-38.
- You can remove instrumentation overheads from execution times. For more information, see “Remove Instrumentation Overheads from Execution Time Measurements” on page 81-32.

Control Profiling Granularity

You can control the granularity of execution-time profiling, that is, prevent the addition of code instrumentation to specific function-call sites. Through the control of profiling granularity, you can:

- Focus on the performance of model components that require improvement. For example, after an initial run, disable profiling for blocks that require little processing time. This action reduces the number of items displayed by the profiling report.
- Reduce the code instrumentation overhead. For example, for simple functions, the code instrumentation overhead can be greater than the execution time of the function code.

To generate execution-time metrics for tasks only, on the **Code Generation** > **Verification** pane of the Configuration Parameters dialog box, select the **Measure task execution time** check box and set **Measure function execution times** to `Off`.

To generate function execution data for referenced models and atomic subsystems in the top model, on the **Code Generation** > **Verification** pane, select the **Measure task execution time** check box

and set **Measure function execution times** to `Coarse (referenced models and subsystems only)`.

The generation of function execution data requires the insertion of measurement probes into the generated code. The software inserts measurement probes for an atomic subsystem only if you set the **Function packaging** field (on the **Code Generation** tab of the Function Block Parameters dialog box) to either `Nonreusable function` or `Reusable function`. If the field is set to `Auto`, then the insertion of probes depends on the packaging choice that results from the `Auto` setting. If the field is set to `Inline`, the software does not insert probes.

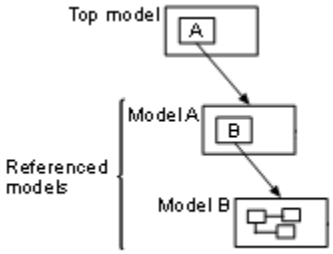
Note In the generated code, the software wraps each function call with measurement probes except when:

- The call site cannot be wrapped because of expression folding (see “Minimize Computations and Storage for Intermediate Results at Block Outputs”).
- The call site is located in the shared utility code (see “Generate Shared Utility Code” on page 8-52).

To generate function execution times for model reference hierarchies:

1. In the top model, open the Configuration Parameters dialog box, and select the **Code Generation** > **Verification** pane.
2. Select the **Measure task execution time** check box
3. For each Model block that you want to profile, specify **Measure function execution times** only at the reference level for which you require function execution times.

For example, consider a top model that has Model block A, which in turn contains Model block B.



If you want to generate execution times for functions from model B, select **Measure task execution time** for the top model and specify **Measure function execution times** for model B.

These parameters of the top model override the corresponding parameters of referenced models:

- **Measure task execution time**. If you disable this parameter for the top model, you also disable function profiling for referenced models.
- **Workspace variable**
- **Save options**

To control code execution profiling for a block in a model, for example, a subsystem block, use the `CodeProfilingOverride` block parameter:

1. In the Simulink Editor, select the block.
2. In the Command Window, run:

   ```
   set_param(gcb, 'CodeProfilingOverride', blockParameterValue)
   ```

   Use one of these values for *`blockParameterValue`*:

   - `'off'` –– Disables profiling for the block.
   - `'on'` –– Enables profiling for the block if profiling is enabled for the parent model.
   - `'inherit'` (default) –– Applies profiling settings of parent block.

Changing the block profiling configuration does not cause the regeneration of production code.

If your top model has a PIL block, the execution profiling settings that apply to the PIL block are the settings from the original model that you used to create the PIL block. See “Simulation with Subsystem Blocks” on page 88-17. You cannot use `CodeProfilingOverride` to control profiling for a PIL block.

See Also

Save options | Measure function execution times | Workspace variable | Measure task execution time

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “View and Compare Code Execution Times” on page 81-6
- “Analyze Code Execution Data” on page 81-29
- “Specify Hardware Timer for Simulink” on page 88-45
- “View and Analyze Simulation Results”

View and Compare Code Execution Times

You can run a software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulation that produces execution-time metrics for your generated code.

During the simulation, you can use the Simulation Data Inspector to observe streamed execution times. In an XCP-based external mode simulation, you can also visualize task scheduling and related target hardware activity.

At the end of the simulation, you can:

- Use the Code Profile Analyzer to analyze execution-time metrics for profiled model components. You can view, for example, the profiled code section and the execution-time distribution for each profiled code section.
- Open a report of execution-time metrics for profiled components.
- Use the Simulation Data Inspector to plot and compare execution times from various simulations.

Run Simulation That Generates Execution-Time Metrics

In this section, to generate execution-time metrics, you run a SIL simulation. For an example that runs an XCP-based external mode simulation, see “Visualize Task Scheduling in XCP External Mode Simulation” on page 81-40.

Open the `SILTopModel` model, which has two subsystems `CounterTypeA` and `CounterTypeB`. At the command line, enter:

```
openExample('ecoder/SILPILVerificationExample', ...
            supportingFile='SILTopModel.slx')
```

To run a SIL simulation that generates execution-time metrics, on the **SIL/PIL** tab:

1. In the **Mode** section, select **SIL/PIL Simulation Only**.
2. In the **Prepare** section, specify these settings:
   a. **System Under Test** — `Top model`
   b. **SIL/PIL Mode** — `Software-in-the-Loop (SIL)`
3. To measure code execution times for the subsystems, in the **Prepare** section:
   a. Click **Settings**.
   b. Under **Time Profiling**:
      - Click the **Task Profiling** button on. This action selects the **Measure task execution time** configuration parameter, which provides execution-time metrics for the task generated from the top model `SILTopModel`.
      - Click the **Save Options** button until it displays `ALL DATA`. This action sets the **Save options** configuration parameter to `All data`.
      - Click the **Functions** button until it displays `COARSE`. This action sets the **Measure function execution times** configuration parameter to `Coarse (referenced models and subsystems only)`, which provides execution-time metrics for the functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.

c Under **Coverage**, click the **Coverage Collection** button off, which disables code coverage analysis. You collect code coverage data separately from profiling data.

d Click the **Settings** button to open the Configuration Parameters dialog box. On the **Data Import/Export** pane, clear **Single simulation output** and then click **OK**.

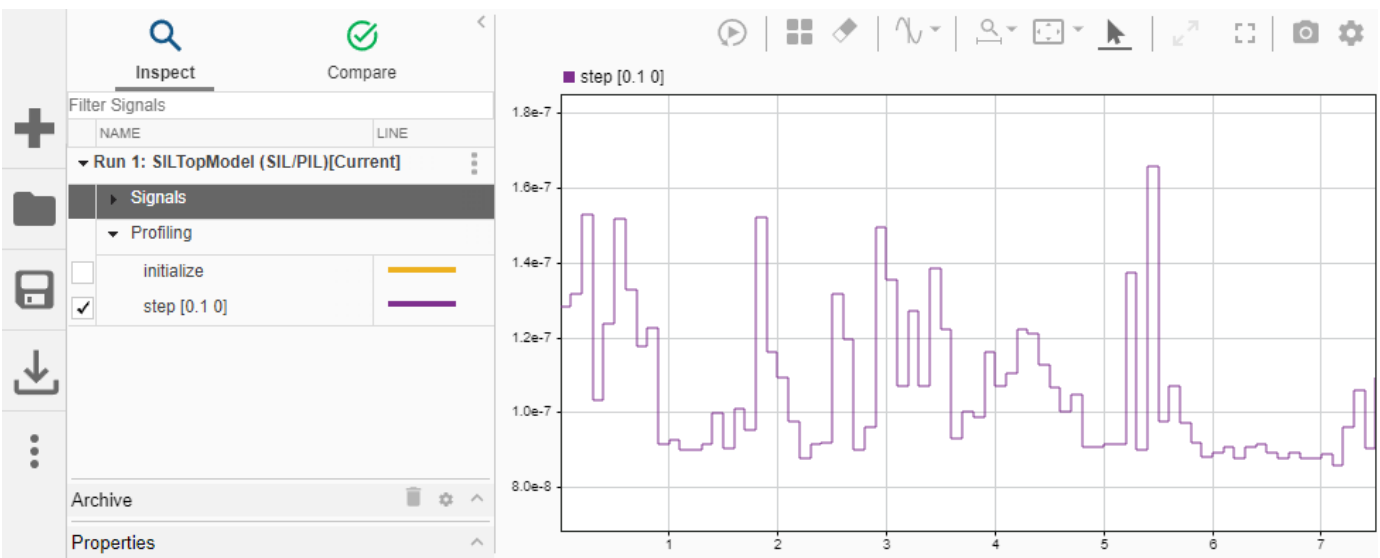
4 To view streamed execution times during the simulation, open the Simulation Data Inspector. In the **Results** section, click .

5 In the **Run** section, click **Run SIL/PIL**.

In the MATLAB base workspace, the simulation generates a variable with the default name, `executionProfile`, which stores the execution profiling data. You can specify an alternative name through the **Workspace variable** configuration parameter.

Note If the **Data Import/Export** > **Single simulation output** check box is selected, the simulation creates the variable in your specified `Simulink.SimulationOutput` object.

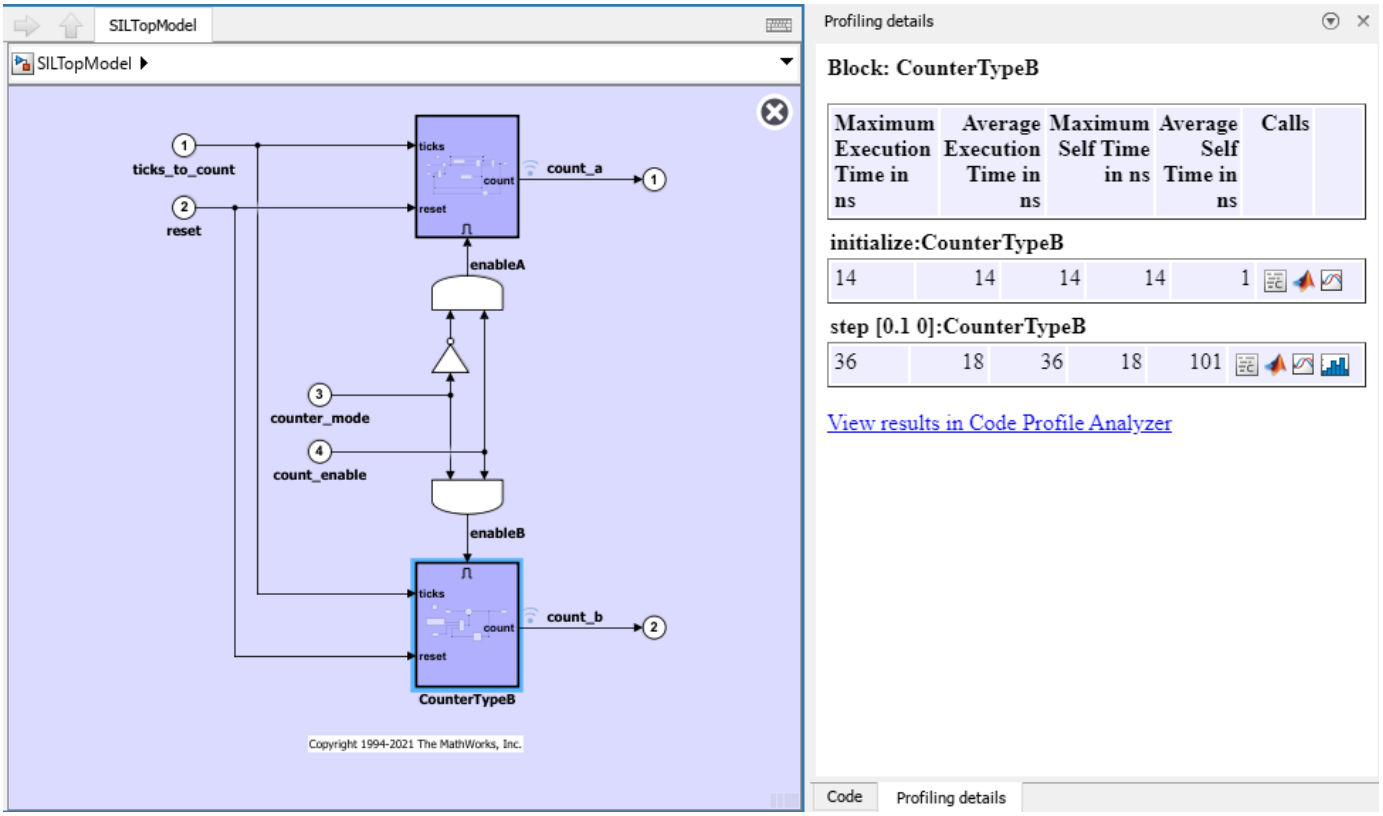
During the simulation, use the Simulation Data Inspector to observe, for example, the variation of execution times for the `step` function.



When the simulation is complete, the profiled model components are colored blue. For top-model SIL or PIL simulations, the Simulink Editor background is also colored blue. The **SIL/PIL** tab displays two additional panels:

- **Profiling details** — Provides links to the code execution profiling report or the Code Profile Analyzer.
- **Code** — Displays the generated code. For example, `SILTopModel.c`.

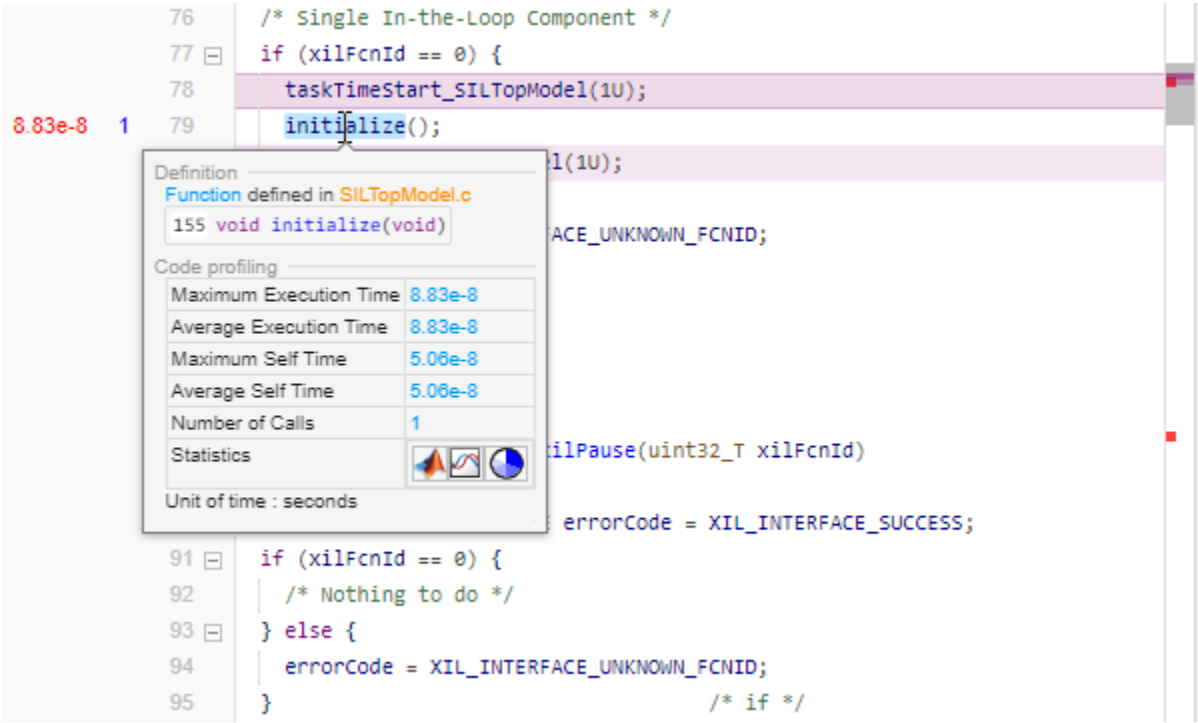
To view execution-time metrics for top-model tasks, click the blue-shaded background. Or, for a profiled component, click a blue-shaded block. For example, subsystem `CounterTypeB`.



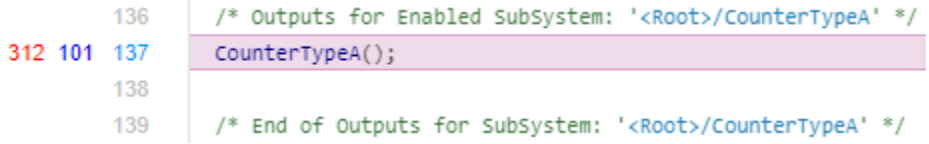
In addition to execution-time metrics, the **Profiling details** panel provides links to:

- The Code Profile Analyzer, which enables you to view and analyze execution-time metrics for all profiled code sections.
- The Simulation Data Inspector, which allows you to plot and compare execution-time measurements for the profiled code section.
- The execution-time distribution for the profiled code section.
- Pie charts that show the relative execution times of caller and called functions

You can trace from a model component to the profiled code section in the Code view. In the **Profiling details** panel, on a code section row, click the icon . For example, if you click the model background and then click the icon for the `SILTopModel_initialize` task, you see the measurement probes around the call site in the SIL application. To see the profiling metrics for the component, place your cursor on the call site.



If you click the icon for a function, the call site is highlighted. The Code view displays the average execution time and number of function calls to the left of the code line. In this example, the maximum and average times are the same because the initialization function was called only one time.



If you close the model or display window, you can reopen the colored model and display window with this line command:

```
annotate(executionProfile)
```

Analyze Task and Function Execution Times

To open the Code Profile Analyzer:

- Click the **SIL/PIL** tab.
- In the **Results** gallery, under **Execution Profiling Results**, click **Code Profile Analyzer**.

Or, in the Command Window, enter:

```
coder.profile.show(executionProfile)
```

The app displays information in the **Task Execution** and **Summary** panels.

View Task Metrics

In the **Task Execution** panel, the **Task Execution Times** view shows execution-time metrics for these generated code sections:

- The model initialization function `initialize`.
- The task represented by the step function `step [0.1 0]`.

Task Execution Times

Section	Minimum Execution Time	Average Execution Time	Maximum Execution Time	Average Self Time	Maximum Self Time	Calls	Execution Time Standard Deviation	Execution Time Variance
initialize	63.3333	63.3333	63.3333	30.0000	30.0000	1	0	0
step [0.1 0]	83.3333	123.8999	271.1111	75.5281	220.5556	101	37.9380	1.4393e+03

For each code section, the table provides this information:

- **Section** — Name of profiled code section.
- **Minimum Execution Time** — Shortest time between start and end of code section.
- **Average Execution Time** — Average time between start and end of code section.
- **Maximum Execution Time** — Longest time between start and end of code section.
- **Average Self Time** — Average execution time, excluding time in child sections.
- **Maximum Self Time** — Maximum execution time, excluding time in child sections.
- **Calls** — Number of calls to the code section.
- **Execution Time Standard Deviation** — A measure of the spread of task execution-times about the average value.
- **Execution Time Variance** — A measure of the dispersion of execution-time measurements. The execution-time standard deviation is the square root of this value.

By default, the app displays time in nanoseconds (10^{-9} seconds). To change the time unit, in the **Settings** section, from the **Time unit** drop-down list, select another unit, for example, `Microseconds (us)`.

The table displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known:

- On a Windows computer, the software determines this value for a SIL simulation.
- On a Linux computer, calibrate the timer manually. In the **Settings** section, in the **CPU Clock (Hz)** field, specify your processor speed in Hz.

The **Results** section of the toolstrip provides features that you can use in your analysis of tasks. For example, to view the `step [0.1 0]` call in generated code:

1. Click the row that contains `step [0.1 0]`.
2. In the **Results** section, click **Highlight Code**. The Simulink Editor displays the call in the Code view.

Code Profiling 1 / 2

```
                222    return XIL_INTERFACE_UNKNOWN_FCNID;
                223  }
                224
                225  switch (xilTID) {
                226   case 1:
                227    taskTimeStart_SILTopModel(2U);
1.00e-7   101   228    step();
                229    taskTimeEnd_SILTopModel(2U);
                230    break;
                231
                232   default:
                233    return XIL_INTERFACE_UNKNOWN_TID;
                234  }
                235
```

View CPU Workload

The CPU Utilization view provides information about the CPU workload, which enables you to determine whether you can run the generated code on your target hardware.

CPU Utilization

Task	Average CPU Utilization	Maximum CPU Utilization
step [0.1 0]	0.0001239%	0.0002711%
Overall CPU Utilization	**0.0001239%**	**0.0002711%**

The table provides this information:

- **Task** — Task name
- **Average CPU Utilization** — Average CPU time required by each task during a sample period
- **Maximum CPU Utilization** — Maximum CPU time required by each task during a sample period
- **Overall CPU Utilization** — Total CPU usage in a sample period.

The software computes the percentage of CPU time assigned to a task by dividing the task execution time by sample time.

The software generates **CPU Utilization** metrics only if the number of timer ticks per second is known, that is, a value is specified in the **CPU Clock (Hz)** field.

Summary Panel (Execution-Time)

Summary

Profiled component:	SILTopModel
Workflow:	SIL/PIL
Environment:	Simulink
Save options:	All data
Time created:	10-Oct-2022 13:57...

The **Summary** panel provides miscellaneous simulation information:

- **Profiled component** — Simulink model that is the source of the generated code

- **Workflow** — Type of code execution
- **Environment** — Simulation environment
- **Save options** — Value of **Save options** configuration parameter
- **Time created** — Date and time when workspace variable with execution-time metrics was created

View Function Metrics

On the toolstrip, in the **Analysis** section, click **Function Execution**.

The **Function Execution Times** view provides a function-call tree and execution-time metrics for the tasks and their child functions, that is, the functions generated from subsystems `CounterTypeA` and `CounterTypeB`.

Function Execution Times

- initialize
  - CounterTypeA
  - CounterTypeB
- step [0.1 0]
  - CounterTypeA
  - CounterTypeB

Section	Minimum Execution Time	Average Execution Time	Maximum Execution Time	Average Self Time	Maximum Self Time	Calls	Execution Time Standard Deviation	Execution Time Variance
initialize	63.3333	63.3333	63.3333	30.0000	30.0000	1	0	0
CounterTypeA	19.4444	19.4444	19.4444	19.4444	19.4444	1	0	0
CounterTypeB	13.8889	13.8889	13.8889	13.8889	13.8889	1	0	0
step [0.1 0]	83.3333	123.8999	271.1111	75.5281	220.5556	101	37.9380	5.1814e+03
CounterTypeA	22.7778	30.4125	100.5556	30.4125	100.5556	101	9.3283	313.2590
CounterTypeB	10.5556	17.9593	35.5556	17.9593	35.5556	101	6.8759	170.2024

The **Relative Function Execution Times** view provides function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code.

Relative Function Execution Times

Section	Self Time / Caller (%)	Self Time / Task (%)	Self Time / Simulation (%)	Function / Simulation (%)
initialize	47.3684	47.3684	0.2385	0.5036
CounterTypeA	30.7018	30.7018	0.1546	0.1546
CounterTypeB	21.9298	21.9298	0.1104	0.1104
step [0.1 0]	60.9589	60.9589	60.6520	99.4964
CounterTypeA	24.5461	24.5461	24.4225	24.4225
CounterTypeB	14.4950	14.4950	14.4220	14.4220

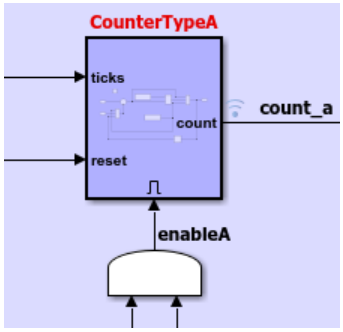
The table contains these metrics.

Column	Comparison Performed
Self Time / Caller (%)	Function self time compared with the total execution time of the caller function.
Self Time / Task (%)	Function self time compared with the total execution time of the task.
Self Time / Simulation (%)	Function self time compared with the total execution time of the simulation.
Function / Simulation (%)	Function execution time, which includes self time and time spent in child functions, compared with the total execution time of the simulation.

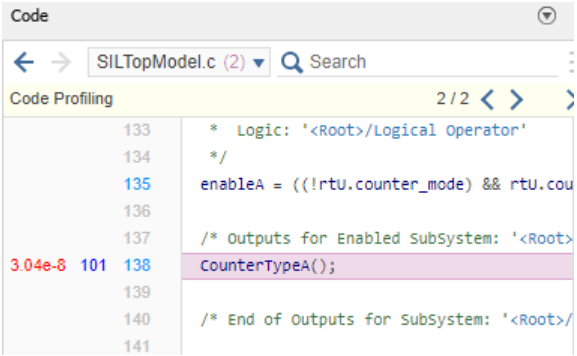
The **Results** section of the toolstrip provides features that you can use in your analysis of functions.

To trace the model component associated with a generated function and its metrics:

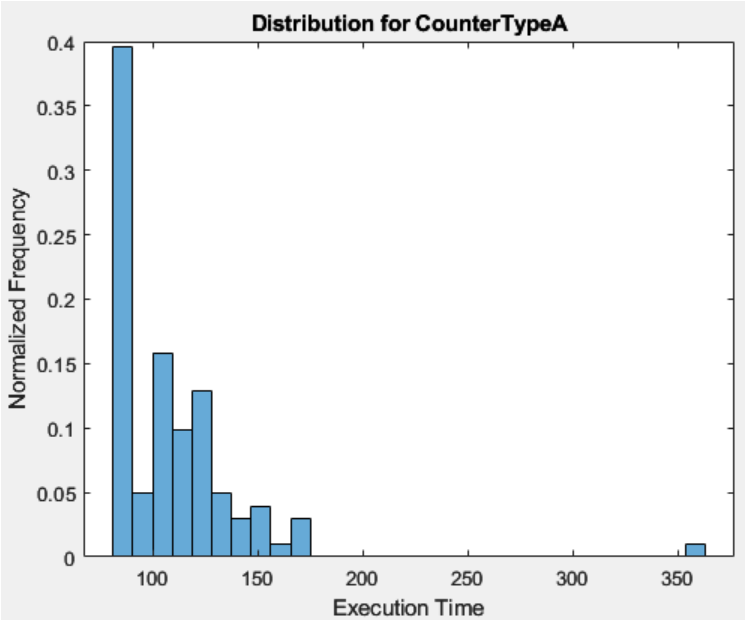
1. In the function-call tree or a metrics table, click the function. For example, `CounterTypeA`, which is called by `step [0.1 0]`.
2. On the toolstrip, in the **Results** section, click **Highlight Source**. The Simulink Editor identifies the subsystem.



To view the function call in generated code, in the **Results** section, click **Highlight Code**. The Simulink Editor displays the call in the Code view.



To display the execution-time distribution for the function, in the **Results** section, click **Generate Distribution**.

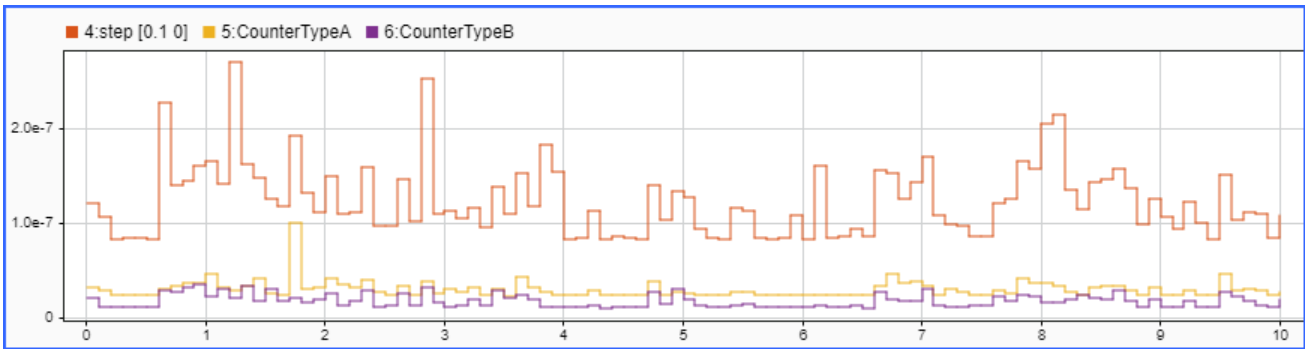


`CounterTypeA` is nested in caller function `step [0.1 0]`. To view the relative execution times of caller and called functions:

1. In the function-call tree or a metrics table, click the caller function step [0.1 0].
2. On the toolstrip, in the **Results** section, click **Function Distribution**, which generates these pie charts.

Use the pie charts to identify functions that are bottlenecks in code execution.

To display measured execution times in the Simulation Data Inspector, in the **Results** section, click **Export to SDI** . The Simulation Data Inspector displays execution times for `CounterTypeA`. Under **Filter Signals**, if you also select the `step [0.1 0]` and `CounterTypeB` check boxes, you can see how execution times for the step and child functions vary over simulation time.



The execution time for the step function `step [0 0.1]` is the sum of its self-time and execution times for the child functions `CounterTypeA` and `CounterTypeB`. You can use the Simulation Data Inspector to manage and compare plots from various simulations.

To view execution-time metrics for the generated code in an HTML report, in the **Results** section, click **Open Report**. For more information, see “Code Execution Profiling Report” on page 81-19.

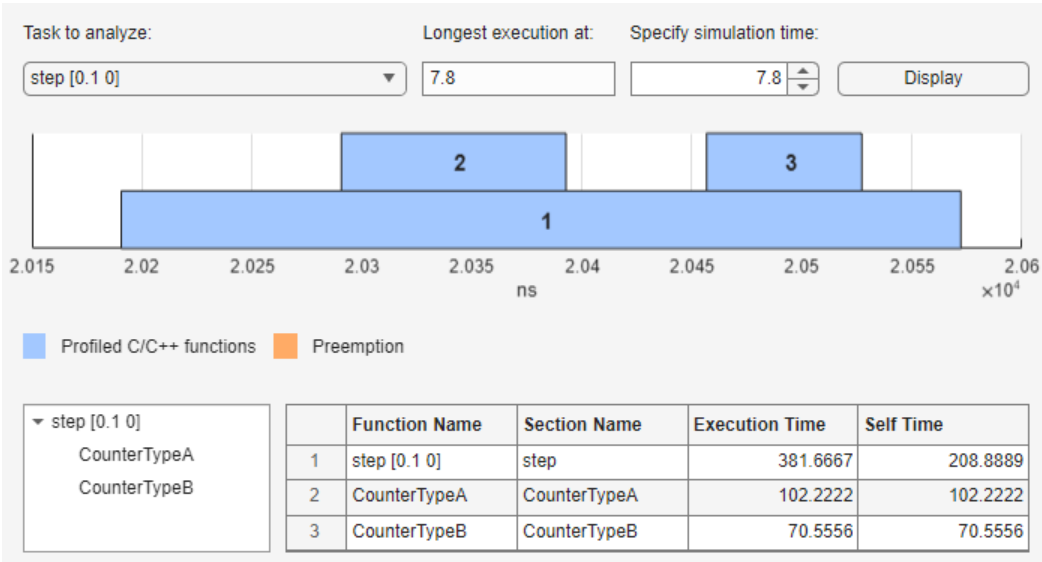
Visualize Function-Call Stack

You can visualize the function call-stack at the simulation step where a task takes longest to execute.

1. On the toolstrip, in the **Analysis** section, click **Function-Call Stack**.
2. From the **Task to analyze** drop-down list, select a code section.
3. Note the **Longest execution at** field value, the simulation time at which execution of the code section is the longest.
4. Use the default value for **Specify simulation time**. Modify the value only if you want to visualize the function-call stack at a different simulation time.
5. Click **Display**.

The panel displays:

- The function-call stack.
- The function-call tree.
- Execution-time and self-time values for profiled code sections.



Compare Execution Times Against Baseline

You can compare execution-time metrics from the current simulation against values from a baseline simulation.

1. On the toolstrip, in the **Analysis** section, click **Comparison**.
2. From the **Compare current data** drop-down list, select the workspace variable that contains results from the baseline simulation. You can update the drop-down list by clicking **Refresh list**.
3. Click **Compare**.

The panel displays:

- The function-call tree
- A percentage comparison of execution-time metrics for the two simulations.

The table cells are colored:

- Green if the current execution-time metric is less than the baseline value.
- Yellow if the current execution-time metric is greater than the baseline value.

If the workspace variable for the baseline simulation does not contain data for a code section, the panel displays `NaN`.

View Code Execution Profiles for Multiple Model Blocks

If you run a top-model simulation that executes multiple Model blocks in SIL or PIL mode with code execution profiling enabled, you can use the Code Profile Analyzer to view execution-time metrics for code generated from the blocks.

1. Open the top model that contains Model blocks. For example, at the command line, enter:

```
openExample('ecoder/SILPILVerificationExample', ...
            supportingFile='SILModelBlock.slx')
```

2. On the **SIL/PIL** tab, in the **Mode** section, specify **SIL/PIL Simulation Only**.
3. In the **Prepare** section, specify these settings:
   - **System Under Test** –– `Model blocks in SIL/PIL mode`
   - **Top Model Mode** –– `Normal`
4. Model block `Counter A` is already configured to run in SIL mode. For Model block `Counter B`, specify these parameters:
   - **Simulation mode** –– `Software-in-the-loop (SIL)`
   - **Code interface** –– `Normal`
5. Both Model blocks reference the model `SILCounter`. To generate execution times for the Model blocks:
   a. Select `CounterA` or `CounterB`.
   b. On the **Model Block** tab, in the **Actions** section, click **Open as Top Model**.
   c. On the **SIL/PIL** tab, in the **Settings** gallery, click **Functions** to `COARSE`, which sets the **Measure function execution times** configuration parameter for `SILCounter` to `Coarse (referenced models and subsystems only)`.
6. To run a top-model simulation:
   a. Select the `SILModelBlock` window.
   b. On the **SIL/PIL** tab, in the **Run** section, click **Run SIL/PIL**.

   Since the **Single simulation output** check box is selected, the simulation creates the variable `executionProfile` in `out`, a `Simulink.SimulationOutput` object.
7. When the simulation is complete, in the **Results** gallery, under **Execution Profiling Results**, click **Code Profile Analyzer**. In the **Task Execution** panel, the **Task Execution Times** view shows execution-time metrics for the generated code sections.

Task Execution Times

Section	Minimum Execution Time	Average Execution Time	Maximum Execution Time	Average Self Time	Maximum Self Time	Calls	Execution Time Standard Deviation	Execution Time Variance
SILCounter_initialize	75.5556	75.5556	75.5556	75.5556	75.5556	1	0	0
SILCounter_step	19.4444	52.3487	175.5556	52.3487	175.5556	101	22.9398	526.2338
SILCounter_initialize	76.6667	76.6667	76.6667	76.6667	76.6667	1	0	0
SILCounter_step	21.1111	56.4521	428.8889	56.4521	428.8889	101	48.1433	2.3178e+03

For information about generating:

- Function execution times for model reference hierarchies, see “Control Profiling Granularity” on page 81-3.
- Code execution profiling reports for referenced models, see “Profiling Reports for Multiple Model Blocks” on page 81-26

Visualize Task Scheduling

You can use the Simulation Data Inspector to visualize task scheduling and related target hardware activity.

Simulation Mode	What You Can Observe	How
SIL or PIL	Task scheduling and the order of function calls	When the SIL or PIL simulation is complete, perform one of these actions: • In the **Results** gallery of the Code Profile Analyzer, click **Generate Schedule**. See "Visualize Task Scheduling in SIL Simulation". • Run the `schedule` function. See "Visualize Task Scheduling in SIL Simulation". • From the **SIL/PIL** tab, in the **Results** gallery, click **Generate Schedule**. See "Analyze Results and Export Test Cases" on page 88-79.
XCP-based external	Task activity, task diagnostics (run-time only), CPU activity, and CPU utilization	During the XCP external mode simulation, use the Simulation Data Inspector to view displays for task scheduling and related activity. After the simulation, use the `schedule` function to regenerate displays (except for task diagnostics). For more information, see "Visualize Task Scheduling in XCP External Mode Simulation" on page 81-40.

See Also

`annotate` | `report` | `schedule` | **Code Profile Analyzer**

Related Examples

- "Create Execution-Time Profile for Generated Code" on page 81-2
- "Analyze Code Execution Data" on page 81-29
- "External Mode Simulation by Using XCP Communication" on page 60-8
- Simulation Data Inspector

More About

- "Tips and Limitations" on page 81-50

Code Execution Profiling Report

You can configure a software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulation to produce execution-time metrics for generated code. The simulation stores data in a workspace variable, for example, `executionProfile`. When the simulation is complete, you can open a code execution profiling report by:

- Using the Code Profile Analyzer. In the **Results** section, under **Overall**, click **Open Report**.
- Running `report(executionProfile)` in the Command Window.

The report contains multiple sections.

Summary

This section gives information about the creation of the report.

1. Summary

Total time	19232
Unit of time	ns
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	3.6e+09
Profiling data created	30-Nov-2021 19:04:25

Feasibility Checks

If the number of timer ticks per second is known, the simulation performs feasibility checks and displays results in this section. The checks are shown in this table.

Check	Comments
Overall average CPU utilization lower than threshold	The default threshold value is 70%. To change the threshold value, for example, to 85%, use this command: `executionProfile.CPUUtilization=0.85;` For information about CPU utilization, see section 5.
Overall maximum CPU utilization lower than threshold	
All average execution times are shorter than their task period	Use the results of these checks to verify that task overrun does not occur.
All maximum execution times are shorter than their task period	

Note An external mode simulation does not produce this section.

In the **Results** column, the report displays Passed or Failed.

2. Feasibility Checks [hide]

Check Description	Results
Overall average CPU utilization lower than threshold (70%)	Passed
Overall maximum CPU utilization lower than threshold (70%)	Passed
All average execution times are shorter than their task period	Passed
All maximum execution times are shorter than their task period	Passed

Profiled Sections of Code

This section contains information about profiled code sections. In the **Section** column, expand and collapse profiled sections by clicking [+] and [–] respectively. This graphic shows fully expanded sections.

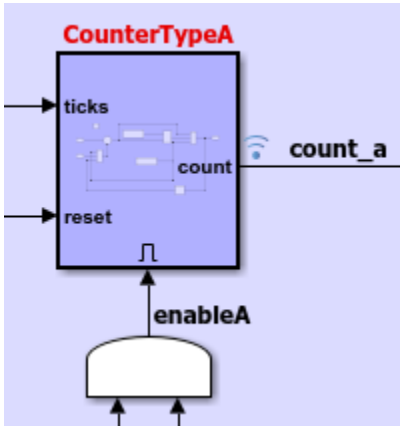
3. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls
[-] initialize	72	72	42	42	1
CounterTypeA	19	19	19	19	1
CounterTypeB	11	11	11	11	1
[-] step [0.1 0]	973	190	309	103	101
CounterTypeA	353	53	353	53	101
CounterTypeB	649	33	649	33	101

In this example, the report contains time measurements for:

- The model initialization function `initialize`.
- A task represented by the step function `step [0.1 0]`.
- Functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.

From the report, you can trace the model component that produces a set of metrics. For example, in the **Section** column, if you click the `CounterTypeA` hyperlink, the Simulink Editor identifies the subsystem.



By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the following command:

```
report(executionProfile,...
         'Units', 'Seconds', ...
         'ScaleFactor', '1e-06', ...
         'NumericFormat', '%0.3f')
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known. On a Windows machine, the software determines this value for a SIL simulation. On a Linux machine, calibrate the timer manually. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:

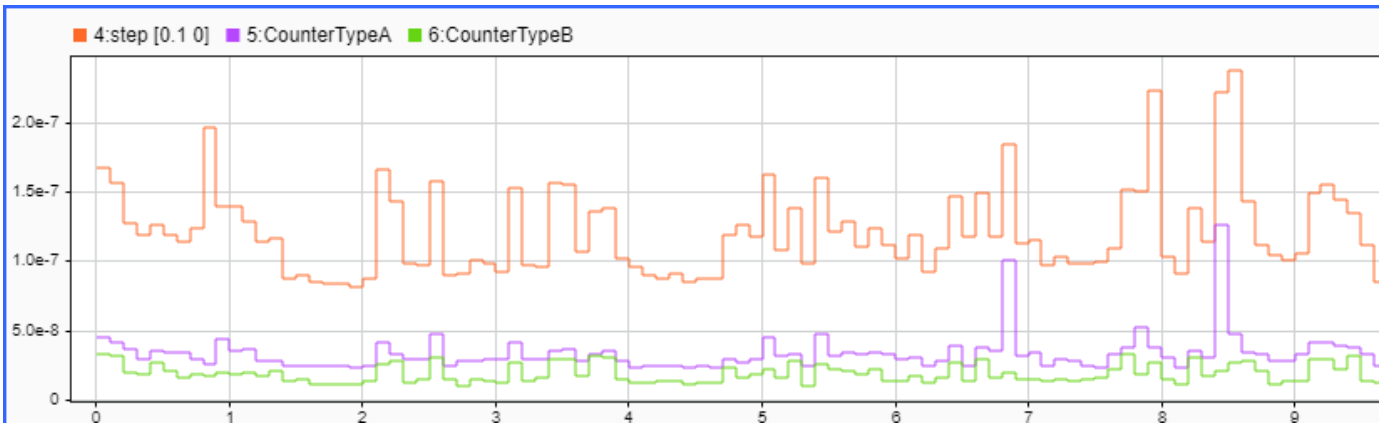
```
executionProfile.TimerTicksPerSecond = 1e9;
```

To view execution-time metrics for a task or function in the Command Window, on the corresponding row, click .

```
>> executionProfile.Sections(5)
  ExecutionTimeCodeSection with properties:

                            Name: 'CounterTypeA'
                          Number: 5
            ExecutionTimeInTicks: [122 102 100 86 88 88 88 86 88 88 156 122  … ]
                 SelfTimeInTicks: [122 102 100 86 88 88 88 86 88 88 156 122  … ]
           TurnaroundTimeInTicks: [122 102 100 86 88 88 88 86 88 88 156 122  … ]
       TotalExecutionTimeInTicks: 19436
            TotalSelfTimeInTicks: 19436
      TotalTurnaroundTimeInTicks: 19436
     MaximumExecutionTimeInTicks: 1270
     MaximumExecutionTimeCallNum: 27
          MaximumSelfTimeInTicks: 1270
          MaximumSelfTimeCallNum: 27
    MaximumTurnaroundTimeInTicks: 1270
    MaximumTurnaroundTimeCallNum: 27
                        NumCalls: 101
          ExecutionTimeInSeconds: [3.3889e-08 2.8333e-08 2.7778e-08 2.3889e-08 … ]
                            Time: [101×1 double]
```

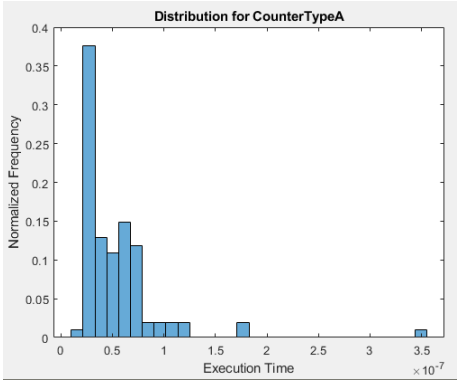
To display measured execution times in the Simulation Data Inspector, click the icon . Then, select the `step [0.1 0]` and `CounterTypeB` check boxes.

You can see how execution times for the step and child functions vary over simulation time.



The execution time for the step function `step [0 0.1]` is the sum of its self-time and execution times for the child functions `CounterTypeA` and `CounterTypeB`. You can use the Simulation Data Inspector to manage and compare plots from various simulations.

To display the execution-time distribution, click the icon .

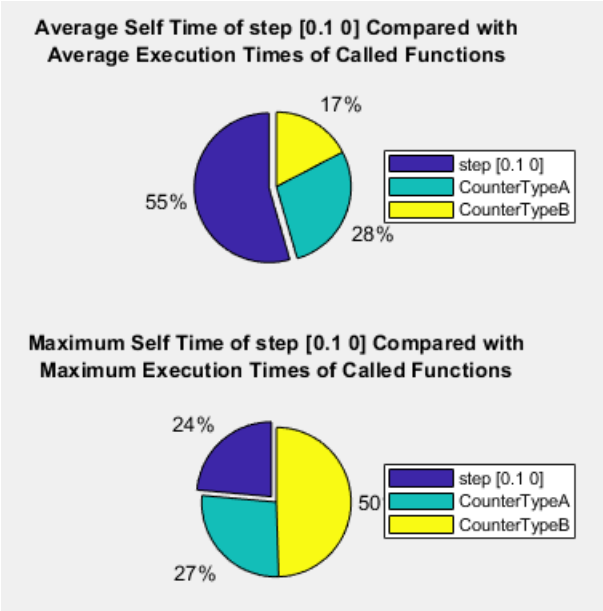


In this example, to create the histogram, the software uses these commands:

```
section=executionProfile.Sections(5);
data=section.ExecutionTimeInSeconds;
histogram(data, 30,'Normalization','probability');
```

The software creates the histogram only if the number of calls is greater than one.

If the generated code for your model contains nested functions, the simulation generates pie charts that show the relative execution times of caller and called functions. To display the pie charts, click the icon for the caller function. For example, `step [0.1 0]`.



The pie charts can help you to identify functions that are bottlenecks in code execution.

This table describes the information that **Profiled Sections of Code** provides.

Column	Description
Section	Name of task, top model, subsystem, or Model block. Click the link to go to the model. With a task, the sample period and sample offset are listed next to the task name. For example, *`SILTopModel_step [0.1 0]`* indicates that the sample period is 0.1 seconds and the sample offset is 0.
Maximum Turnaround Time (Simulink Real-Time and support packages in real-time mode)	Longest time between start and end of code section. Includes preemption time.
Average Turnaround Time (Simulink Real-Time and support packages in real-time mode)	Average time between start and end of code section. Includes preemption time.
Maximum Execution Time	Longest time between start and end of code section.
Average Execution Time	Average time between start and end of code section.
Maximum Self Time	Maximum execution time, excluding time in child sections.
Average Self Time	Average execution time, excluding time in child sections.
Calls	Number of calls to the code section.

Column	Description
	Icon that you click to see the profiled code section in the Code view of the SIL/PIL app. The code section can be a task or a function. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display the profiled code section in the Command Window. Equivalent to executing the command `executionProfile.Sections(i)`. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display measured execution times with Simulation Data Inspector. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display the execution-time distribution for the profiled code section. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display pie charts that show the relative execution times of caller and called functions in the generated code.

Execution Times in Percentages

This section provides function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code.

4. Execution Times in Percentages [hide]

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
[-] initialize	58.14%	58.14%	0.21665%	0.37264%
CounterTypeA	27.132%	27.132%	0.1011%	0.1011%
CounterTypeB	14.729%	14.729%	0.054885%	0.054885%
[-] step [0.1 0]	54.501%	54.501%	54.298%	99.627%
CounterTypeA	28.177%	28.177%	28.072%	28.072%
CounterTypeB	17.321%	17.321%	17.257%	17.257%

This table describes the information that each column provides.

Column	Comparison Performed
Self Time / Caller Function	Function self time compared with the total execution time of the caller function.

Column	Comparison Performed
Self Time / Task	Function self time compared with the total execution time of the task.
Self Time / Simulation	Function self time compared with the total execution time of the simulation.
Function / Simulation	Function execution time, which includes self time and time spent in child functions, compared with the total execution time of the simulation.

Note An external mode simulation does not produce this section.

CPU Utilization

To help you to decide whether you can run the generated code on your target hardware, this section of the code execution profiling report provides CPU workload information:

- Average and maximum CPU time required by each task during a sample period.
- Total CPU usage in a sample period.

The software computes the percentage of CPU time assigned to a task by dividing task execution time by sample time.

5. CPU Utilization [hide]

Task	Average CPU Utilization	Maximum CPU Utilization
step [0.1 0]	0.0001897%	0.0009728%
Overall CPU Utilization	0.0001897%	0.0009728%

The software generates **CPU Utilization** only if the number of timer ticks per second is known.

Definitions

This section provides descriptions of some metrics.

6. Definitions

CPU Utilization: Percentage of CPU time assigned to a task. Computed by dividing task execution time by sample time.

CPU Utilization Threshold: Maximum CPU utilization that allows task scheduling without deadline misses or overuns. To override the default (70%), use the CPUUtilization parameter.

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

Comparison with Baseline Performance

You can create a code execution profiling report that compares the performance of the generated code against the performance of a reference version. For example, suppose you want to use the execution-time metrics in `executionProfile` as the baseline for `SILTopModel`.

1. In the Workspace browser, rename `executionProfile` to `executionProfileBaseline`.
2. Run another SIL simulation.
3. To create a report that compares the latest performance against the baseline, in the Command Window, run:

   ```
   report(executionProfile,'baseline', executionProfileBaseline)
   ```

In the report, the **Comparison with Baseline Performance** and **CPU Utilization** sections provide comparisons of execution-time metrics. The differences in execution-time metrics are color-coded:

- Red — An increase in execution times or CPU usage.
- Green — A decrease in execution times or CPU usage.
- Black — No change.

5. Comparison with Baseline Performance [hide]

Section	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time
[-] initialize	-22.75%	-22.75%	-20.86%	-20.86%
CounterTypeA	-26.47%	-26.47%	-26.47%	-26.47%
CounterTypeB	-23.53%	-23.53%	-23.53%	-23.53%
[-] step [0.1 0]	+53.35%	+11.26%	+52.67%	+7.55%
CounterTypeA	+125.25%	+20.19%	+125.25%	+20.19%
CounterTypeB	-69.86%	+9.09%	-69.86%	+9.09%

6. CPU Utilization [hide]

Task	Average CPU Utilization Baseline in brackets	Maximum CPU Utilization Baseline in brackets
step [0.1 0]	0.0001181% (0.0001062%)	0.0003689% (0.0002406%)
Overall CPU Utilization	0.0001181% (0.0001062%)	0.0003689% (0.0002406%)

Profiling Reports for Multiple Model Blocks

You can configure a top-model simulation that runs multiple Model blocks in SIL or PIL mode with code execution profiling enabled. When you run the simulation, it generates separate code execution profiling reports for the referenced models.

1. Open the top model that contains Model blocks. For example, at the command line, enter:

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILModelBlock.slx')
   ```
2. On the **SIL/PIL** tab, in the **Mode** section, specify **SIL/PIL Simulation Only**.
3. In the **Prepare** section, specify these settings:
   - **System Under Test** -- `Model blocks in SIL/PIL mode`
   - **Top Model Mode** -- `Normal`
4. Model block `Counter A` is already configured to run in SIL mode. For Model block `Counter B`, specify these parameters:

- **Simulation mode** -- `Software-in-the-loop (SIL)`
- **Code interface** -- `Normal`

5. Both Model blocks reference the model `SILCounter`. To generate execution times for the Model blocks:

   a. Select `CounterA` or `CounterB`.

   b. On the **Model Block** tab, in the **Actions** section, click **Open as Top Model**.

   c. On the **SIL/PIL** tab, in the **Settings** gallery, click **Functions** to `COARSE`, which sets the **Measure function execution times** configuration parameter for `SILCounter` to `Coarse (referenced models and subsystems only)`.

6. To run a top-model simulation:

   a. Select the `SILModelBlock` window.

   b. On the **SIL/PIL** tab, in the **Run** section, click **Run SIL/PIL**.

   Since the **Single simulation output** check box is selected, the simulation creates the variable `executionProfile` in `out`, a `Simulink.SimulationOutput` object.

7. When the simulation is complete, to show the number of Model blocks that are profiled, in the Command Window, enter:

```
NumOfReports=out.executionProfile.getModuleNumber()
```

```
NumOfReports =
     2
```

8. To open the report for `CounterA`, the first Model block, run:

```
report(out.executionProfile.getModule(1))
```

1. Summary

Total time	4955
Unit of time	ns
Command	report(, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	3.6e+09
Profiling data created	03-Dec-2021 12:24:45

The simulation ran multiple model blocks in SIL or PIL mode with code execution profiling enabled. To view execution-time metrics for a model block, click the corresponding report link:

rtwdemo_sil_modelblock/CounterA	This report
rtwdemo_sil_modelblock/CounterB	Open report

2. Profiled Sections of Code

9. In the **Summary** section, the report provides a link to the second Model block report. To open the report for `CounterB`, click Open report or run:

```
report(out.executionProfile.getModule(2))
```

For more information about generating function execution times for model reference hierarchies, see “Control Profiling Granularity” on page 81-3.

See Also

Code Profile Analyzer

Related Examples

- “Create Execution-Time Profile for Generated Code” on page 81-2
- “View and Compare Code Execution Times” on page 81-6
- Simulation Data Inspector

Analyze Code Execution Data

After a software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulation, you can analyze execution-time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

1. Open the model `SILTopModel`.

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILTopModel.slx')
   ```

2. In the **Mode** section, select **SIL/PIL Simulation Only**.
3. In the **Settings** gallery, specify options to enable profiling:
   - Under **SIL/PIL Settings**, click **Portable Word Sizes** on.
   - Under **Time Profiling**, click **Task Profiling** on, click **Save Options** to `ALL DATA`, and click **Functions** off.
   - Under **Coverage**, click **Coverage Collection** off.
4. In the **Run** section, click **Run SIL/PIL**.

The software generates out, a `Simulink.SimulationOutput` object that contains `executionProfile`, a `coder.profile.ExecutionTime` object.

To get the total number of code sections that have profiling data, use the `Sections` method:

```
no_of_Sections = out.executionProfile.Sections
```

```
no_of_Sections =

  1×2 ExecutionTimeTaskSection array with properties:

    Name
    Number
    ExecutionTimeInTicks
    SelfTimeInTicks
    TurnaroundTimeInTicks
    TotalExecutionTimeInTicks
    TotalSelfTimeInTicks
    TotalTurnaroundTimeInTicks
    MaximumExecutionTimeInTicks
    MaximumExecutionTimeCallNum
    MaximumSelfTimeInTicks
    MaximumSelfTimeCallNum
    MaximumTurnaroundTimeInTicks
    MaximumTurnaroundTimeCallNum
    NumCalls
    ExecutionTimeInSeconds
    Time
```

To get the `coder.profile.ExecutionTimeSection` object for a profiled code section, use the method `Sections`.

```
FirstSectionProfile = out.executionProfile.Sections(1)
SecondSectionProfile = out.executionProfile.Sections(2)
```

```
FirstSectionProfile =

  ExecutionTimeTaskSection with properties:

                            Name: 'initialize'
                          Number: 1
            ExecutionTimeInTicks: 154
```

```
                SelfTimeInTicks: 154
           TurnaroundTimeInTicks: 154
       TotalExecutionTimeInTicks: 154
            TotalSelfTimeInTicks: 154
      TotalTurnaroundTimeInTicks: 154
     MaximumExecutionTimeInTicks: 154
     MaximumExecutionTimeCallNum: 1
          MaximumSelfTimeInTicks: 154
          MaximumSelfTimeCallNum: 1
    MaximumTurnaroundTimeInTicks: 154
    MaximumTurnaroundTimeCallNum: 1
                       NumCalls: 1
          ExecutionTimeInSeconds: 4.2778e-08
                            Time: 0


SecondSectionProfile =

  ExecutionTimeTaskSection with properties:

                            Name: 'step [0.1 0]'
                          Number: 2
            ExecutionTimeInTicks: [262 216 176 174 174 174 178 172 … ]
                 SelfTimeInTicks: [262 216 176 174 174 174 178 172 … ]
           TurnaroundTimeInTicks: [262 216 176 174 174 174 178 172 … ]
       TotalExecutionTimeInTicks: 22366
            TotalSelfTimeInTicks: 22366
      TotalTurnaroundTimeInTicks: 22366
     MaximumExecutionTimeInTicks: 748
     MaximumExecutionTimeCallNum: 68
          MaximumSelfTimeInTicks: 748
          MaximumSelfTimeCallNum: 68
    MaximumTurnaroundTimeInTicks: 748
    MaximumTurnaroundTimeCallNum: 68
                       NumCalls: 101
          ExecutionTimeInSeconds: [7.2778e-08 6.0000e-08 4.8889e-08 4.8333e-08 … ]
                            Time: [101×1 double]
```

Use `coder.profile.ExecutionTimeSection` methods to extract profiling information for a particular code section. For example, use `Name` to obtain the name of a profiled task.

```
name_of_section = SecondSectionProfile.Name
```

```
name_of_section =

    'step [0.1 0]'
```

If the timer is uncalibrated and you know the timer rate, for example 2.2 GHz, you can use the `coder.profile.ExecutionTime` method `TimerTicksPerSecond` to calibrate the timer.

```
out.executionProfile.TimerTicksPerSecond = 2.2e9;
SecondSectionProfile = out.executionProfile.Sections(2);
```

See Also

Related Examples

- “Create Execution-Time Profile for Generated Code” on page 81-2
- “View and Compare Code Execution Times” on page 81-6
- “Tips and Limitations” on page 81-50

Code Execution Profiling From the Command Line

This example shows how you can use line commands to produce execution-time metrics for generated code.

1. Open `SILTopModel`, which has two subsystems and is configured to run a software-in-the-loop (SIL) simulation.

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILTopModel.slx')
   ```

2. Configure code execution time profiling:

   a. Produce execution-time metrics for the task generated from the top model.

      ```
      set_param(gcs, 'CodeExecutionProfiling', 'on');
      ```

   b. Produce execution-time metrics for functions generated from the subsystems in the model.

      ```
      set_param(gcs, 'CodeProfilingInstrumentation', 'coarse');
      ```

   c. Save all code profiling data to an object in the base workspace.

      ```
      set_param(gcs,...
                'CodeProfilingSaveOptions', 'AllData');
      ```

3. Disable Simulink Coverage and third-party code coverage analysis.

   ```
   set_param(gcs,...
             'CovEnable', 'off');
   covSettings = get_param(gcs, 'CodeCoverageSettings');
   covSettings.CoverageTool = 'None';
   set_param(gcs, 'CodeCoverageSettings', covSettings);
   ```

4. Run the simulation.

   ```
   simOut = sim(gcs);
   ```

   The simulation generates the variable `executionProfile` (default) in the object `simOut`.

5. When the simulation is complete, open the code execution report.

   ```
   report(simOut.executionProfile)
   ```

6. To visualize how the tasks are scheduled and the generated code is executed, run:

   ```
   schedule(simOut.executionProfile)
   ```

See Also

`report` | `schedule`

Related Examples

- “Create Execution-Time Profile for Generated Code” on page 81-2
- “View and Compare Code Execution Times” on page 81-6
- “Analyze Code Execution Data” on page 81-29

Remove Instrumentation Overheads from Execution Time Measurements

To improve execution-time profiling of generated code that runs on deterministic hardware, you can run processor-in-the-loop (PIL) simulations that automatically filter the overheads associated with code instrumentation. You can use the target hardware to estimate the average overhead value or you can specify the value manually.

Note By default, the software filters the execution times of AUTOSAR Runtime Environment (RTE) code and functions that run within Simulink, for example, Function Caller blocks.

Approach for Estimating and Removing Instrumentation Overhead

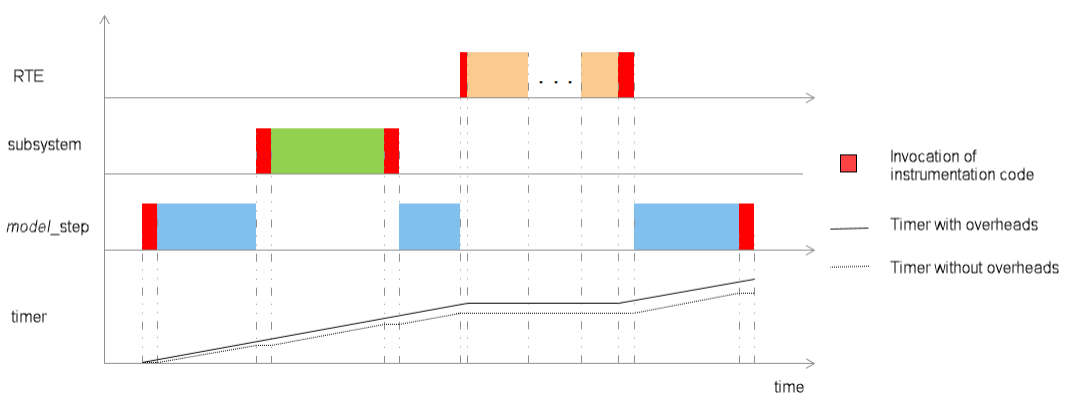
When you set up target connectivity for PIL simulations, you can specify objects or functions for estimating and removing instrumentation overheads from execution-time measurements.

You can:

- Set up a benchmark program that Simulink runs a specified number of times at the start of a PIL simulation. The benchmark program estimates the overheads for the timer API and for timer freezing and unfreezing, task, and function instrumentation code. Simulink obtains average values for the instrumentation overheads.
- Manually specify instrumentation overhead values.

During the PIL simulation, Simulink subtracts the instrumentation overhead values from the execution-time measurements. The Simulation Data Inspector and the code execution profiling report display the corrected measurement values.

This figure shows how the uncorrected and corrected timer values correspond to invocations of code sections.



The corrected timer value excludes execution time associated with instrumentation code, RTE code, and function code that runs within Simulink.

To estimate and remove instrumentation overheads from execution-time measurements:

- If PIL connectivity is set up by using the `target` package, use overhead classes from the package.
- If PIL connectivity is set up by using the `rtw.connectivity.Config` class, use functions from the class.

Remove Instrumentation Overheads by Using target Package

To provide the software with instrumentation overhead values, use one of these methods:

- Set up the benchmark program that estimates overhead values.
- Using overhead classes, manually specify overhead values.

Set Up Benchmark Program

When you set up target connectivity on page 88-48, implement these steps:

1. Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', myProcessorObjectId);
timer = target.get('Timer', myTimerObjectId);
```

2. Create a `target.ProfilingTaskOverhead` object.

```
taskOverhead = target.create('ProfilingTaskOverhead', ...
                             'Name', 'Timer Task Overhead');
taskOverhead.Counter = timer;
taskOverhead.MinimumBenchmarkIterations = 150;
```

3. Associate the overhead values with the processor.

```
processor.Overheads = taskOverhead;
```

Specify Overhead Values

When you set up target connectivity on page 88-48, create overhead class objects and use the `Value` property to specify your value for the corresponding instrumentation overhead.

1. Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', myProcessorObjectId);
timer = target.get('Timer', myTimerObjectId);
```

2. Create a `target.ProfilingTaskOverhead` object.

```
taskOverhead = target.create('ProfilingTaskOverhead',...
                             'Name', 'Timer Task Overhead');
taskOverhead.Value = 10;
taskOverhead.Counter = timer;
```

3. Create a `target.ProfilingFunctionOverhead` object.

```
functionOverhead = target.create('ProfilingFunctionOverhead',...
                            'Name', 'Timer Function Overhead');
functionOverhead.Value = 20;
functionOverhead.Counter = timer;
```

4. Create a `target.ProfilingFreezingOverhead` object.

```
freezingOverhead = target.create('ProfilingFreezingOverhead',...
                                 'Name', 'Timer Freezing Overhead');
```

```
freezingOverhead.Value = 30;
freezingOverhead.Counter = timer;
```

5 Associate the overhead values with the processor.

```
processor.Overheads = ...
 [taskOverhead, functionOverhead, freezingOverhead];
```

Remove Instrumentation Overheads by Using Functions from rtw.connectivity.Config

The `rtw.connectivity.Config` class provides these functions for the removal of instrumentation overheads.

Function	Description
`activateOverheadFiltering`	`obj.activateOverheadFiltering (value);` activates filtering of instrumentation overheads if value is `true`. Deactivates overhead filtering if value is `false`.
`isOverheadFilteringActive`	`obj.isOverheadFilteringActive` returns `true` if instrumentation overhead filtering is activated. Otherwise, returns `false`.
`runOverheadBenchmark`	`obj.runOverheadBenchmark(value);` runs a benchmark program if value is `true`. Does not run the benchmark program if value is `false`.
`isOverheadBenchmarkExecuted`	`obj.isOverheadBenchmarkExecuted;` returns `true` if execution of benchmark program is complete. Otherwise, returns `false`.
`setOverheadBenchmarkSteps`	`obj.setOverheadBenchmarkSteps(steps);` specifies number of steps for benchmark program.
`getOverheadBenchmarkSteps`	`steps = obj.getOverheadBenchmarkSteps` returns the number of specified steps for the benchmark program.
`setOverheads`	`obj.setOverheads(taskInstrumentationOverhead, functionInstrumentationOverhead, freezingInstrumentation);` specifies the overhead values for task, function, and freezing instrumentation respectively.
`getOverheads`	`[taskInstrumentationOverhead, functionInstrumentationOverhead, freezingInstrumentation] = obj.getOverheads;` returns overhead values for task, function, and freezing instrumentation respectively.

When you set up target connectivity on page 88-38 for PIL simulations, configure the removal of the instrumentation overheads through an `rtw.connectivity.Config` subclass. For example, this code shows how you can run the benchmark program.

```
classdef overheadConnectivityConfig < rtw.connectivity.Config
  methods
    function this = customConnectivityConfig(componentArgs)

      % Create builder
      targetApplicationFramework = ...
        mypil.TargetApplicationFramework(componentArgs);
      builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
        targetApplicationFramework, '');

      % Create launcher
      launcher = mypil.Launcher(componentArgs, builder);

      % Set up communication
      hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                         componentArgs, ...
                         launcher, ...
                         rtiostreamLibTCPIP);

      % Call super class constructor to register components
        this@rtw.connectivity.Config(componentArgs,...
                                     builder,...
                                     launcher,...
                                     hostCommunicator);

      % Register hardware-specific timer, which enables
      % code execution profiling. This example uses a
      % timer for the host platform.
      timer = coder.profile.crlHostTimer();
      this.setTimer(timer);

      % Specify removal of profiling instrumentation overheads
      this.activateOverheadFiltering(true);
      this.runOverheadBenchmark(true);
      this.setOverheadBenchmarkSteps(50);

    end
  end
end
```

Instead of running the benchmark program, you can use `setOverheads` to provide your values for instrumentation overheads.

```
    ...

    % Remove instrumentation overhead using manually specified values
    this.activateOverheadFiltering(true);
    this.setOverheads (taskInstrumentationOverhead, ...
      functionInstrumentationOverhead, freezingInstrumentation);

    ...
```

Retrieve Benchmark Results

When a PIL simulation is complete, you can retrieve benchmark results from the variable that you specify in the **Code Generation** > **Verification** > **Workspace variable** field (`CodeExecutionProfileVariable`), for example, `myExecutionProfile`.

```
benchmarkResults = myExecutionProfile.getOverheadBenchmarkData();
```

benchmarkResults is a structure that contains code instrumentation overhead values.

```
>> benchmarkResults
 ans =
 struct with fields:
 TimestampOverhead: [1×1 struct]
 TaskOverhead: [1×1 struct]
 FunctionOverhead: [1×1 struct]
 FreezingOverhead: [1×1 struct]

>> benchmarkResults.TaskOverhead
 ans =
 struct with fields:
 Data: [1×100 uint64]
 NumSamples: 100
 Average: 45.2800
```

See Also

`rtw.connectivity.Config` | `target` | `target.ProfilingTaskOverhead` | `target.ProfilingFunctionOverhead` | `target.ProfilingFreezingOverhead`

More About

- "Set Up PIL Connectivity by Using target Package" on page 88-48
- "Create PIL Target Connectivity Configuration for Simulink" on page 88-38
- "Create Execution-Time Profile for Generated Code" on page 81-2
- "View and Compare Code Execution Times" on page 81-6

Reduce Bandwidth Usage During Code Execution Profiling

This topic shows how you can reduce communication channel bandwidth usage when running software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations with code execution profiling enabled.

Capture Main Profiling Metrics on Target Hardware

For code execution profiling, to reduce the communication channel bandwidth usage during a software-in-the-loop (SIL) simulation, processor-in-the-loop simulation (PIL), or XCP-based external mode simulation, you can capture and store only these profiling metrics on the target hardware:

- Maximum execution time of code section
- Average execution time of code section
- Number of calls to code section

Enable the capture of the main profiling metrics by setting the configuration parameter **Save options** to `Metrics only`. At the end of the simulation, Simulink uploads the metrics from the target hardware to your development computer.

To run a SIL simulation that generates only the main execution-time metrics:

1. Open the model `SILTopModel`, which has two subsystems `CounterTypeA` and `CounterTypeB`.

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILTopModel.slx')
   ```

2. On the SIL/PIL tab, in the **Mode** section, select **SIL/PIL Simulation Only**.
3. In the **Prepare** section, specify these settings:
   a. **System Under Test** — `Top model`
   b. **SIL/PIL Mode** — `Software-in-the-Loop (SIL)`
4. To measure code execution times for the subsystems:
   a. Click .
   b. In the Configuration Parameters dialog box, select **Measure task execution time**, which provides execution-time metrics for the task generated from the top model `SILTopModel`.
   c. Set **Measure function execution times** to `Coarse (referenced models and subsystems only)`, which provides execution-time metrics for the functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.
   d. Specify a **Workspace variable**, for example, `executionProfile`. If the **Data Import/Export** > **Single simulation output** check box is selected, the software creates the variable in the `Simulink.SimulationOutput` object that you specify.
   e. From the **Save options** drop-down list, select `Metrics only`.
   f. Click **OK**.
5. In the **Prepare** section, open the **Settings** gallery. Under **Coverage**, click the **Coverage Collection** button off.
6. In the **Run** section, click **Run SIL/PIL**.

To open the code execution profiling report, in the **Profiling details** panel, click the displayed link. In section 2, the report displays the profiled code sections as a flattened list.

2. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Calls
rtwdemo_sil_topmodel_initialize	299	299	1
rtwdemo_sil_topmodel_step [0,1 0]	450	275	101
CounterTypeA (CounterTypeA)	106	78	101
CounterTypeB (CounterTypeB)	72	37	101
CounterTypeA (CounterTypeA_Init)	57	57	1
CounterTypeB (CounterTypeB_Init)	33	33	1

The software cannot extract the hierarchy of calls to code sections because this information is not stored on the target hardware.

Overhead filtering on page 81-32 is not supported when **Save options** is set to `Metrics only`.

Specify Memory Allocation for Code Execution Profiling

You can specify memory allocation for code profiling with XCP-based external mode simulations. If you want to:

- Limit the memory allocated for code profiling because your target hardware has limited resources, use the Simulink parameter `CodeProfilingXCPMaxMemory`.
- Reduce communication channel bandwidth usage, use the Simulink parameter `CodeProfilingMaxBufferSize`.

The parameters control memory allocation only when `CodeProfilingSaveOptions` is set to `'SummaryOnly'` or `'AllData'`.

This table provides details.

Parameter	Details

`CodeProfilingXCPMaxMemory`	Purpose — Specify the maximum amount of memory to use for code profiling.
	
	Range — 0 to 2^{32} - 1. Default is `"0"`.
	
	Data type — `string`, which resolves to `uint32`.
	
	Example:
	
	`set_param(gcs, CodeProfilingXCPMaxMemory="4000")`
	`get_param(gcs, 'CodeProfilingXCPMaxMemory')`

Parameter	Details

`CodeProfilingMaxBufferSize`	Purpose — Specify maximum size of profiling data buffer that is uploaded from target hardware to development computer.
	
	Range — 0 to 255. Default is `"0"`.
	
	Data type — `string`, which resolves to `uint32`.
	
	Example:
	
	`set_param(gcs, CodeProfilingMaxBufferSize="200")`
	`get_param(gcs, 'CodeProfilingMaxBufferSize')`

Display Absolute Time

During an XCP-based external mode simulation, the Simulation Data Inspector displays, by default, streamed signal values and execution-time metrics with respect to simulation time: after the execution of a task, the target application uploads data samples from the profiling buffer and associates all data samples from the profiling buffer with the same simulation time.

There are potential issues with streaming simulation time:

- During data uploading, memory usage can be suboptimal if the target application uploads the whole profiling buffer even when the profiling buffer contains only a few data samples. The maximum size of the profiling buffer is 255 bytes.
- Data uploading can slow down the execution of the target application.

If the communication channel for the external mode simulation has a low bandwidth or the target application runs tasks with high sampling rates, the generation of profiling data can exceed data uploading, which leads to data loss.

To improve bandwidth usage during data uploading, specify the display of absolute time:

```
set_param(modelName, 'CodeProfilingXCPUseAbsoluteTime', 'on')
```

When the external mode simulation runs, the target application uploads data samples only when the profiling buffer is full. The Simulation Data Inspector displays streamed values with respect to absolute time instead of simulation time.

See Also

Save options | Measure function execution times | Workspace variable | Measure task execution time

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “View and Compare Code Execution Times” on page 81-6

Visualize Task Scheduling in XCP External Mode Simulation

If you run an XCP-based external mode simulation with code execution profiling enabled, you can use the Simulation Data Inspector to visualize how the target hardware executes generated code.

This example shows how to configure and run the simulation and display task activity and diagnostics and CPU activity and utilization.

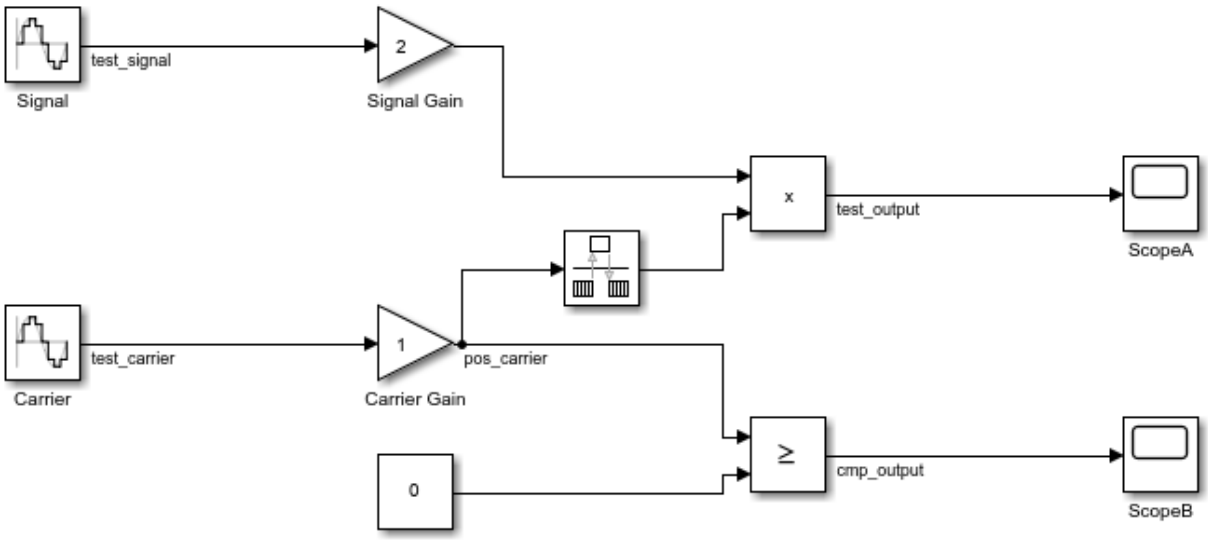
Model Configuration

1 Create and configure a model in a folder.

In the Command Window, enter these commands:

```
% Create work folder
if mkdir('workFolder')
    rmdir 'workFolder' s
    mkdir('workFolder')
end
cd workFolder
open_system(new_system('myModel'));

% Add, configure, and connect blocks
add_block('simulink/Sources/Sine Wave','myModel/Signal',...
                                        'Bias','5',...
                                        'Amplitude','2',...
                                        'SampleTime','0.01');
set_param('myModel/Signal','position',[140,80,180,120]);

add_block('simulink/Math Operations/Gain','myModel/Signal Gain',...
                                        'Gain','2');
set_param('myModel/Signal Gain','position',[340,80,380,120]);
add_line('myModel','Signal/1','Signal Gain/1');

% Label signal
signal=get_param('myModel/Signal','PortHandles');
set_param(signal.Outport(1),'Name','test_signal');

% And so on
add_block('simulink/Math Operations/Product','myModel/Product');
set_param('myModel/Product','position',[540,160,580,200]);
add_line('myModel','Signal Gain/1','Product/1');

add_block('simulink/Sources/Sine Wave','myModel/Carrier',...
                                        'Bias','5',...
                                        'Amplitude','10',...
                                        'Frequency','3',...
                                        'SampleTime','0.02');

set_param('myModel/Carrier','position',[140,240,180,280]);

add_block('simulink/Math Operations/Gain','myModel/Carrier Gain');
set_param('myModel/Carrier Gain','position',[340,240,380,280]);
add_line('myModel','Carrier/1','Carrier Gain/1');

signal=get_param('myModel/Carrier','PortHandles');
set_param(signal.Outport(1),'Name','test_carrier');

add_block('simulink/Signal Attributes/Rate Transition',...
          'myModel/Rate Transition');
set_param('myModel/Rate Transition','position',[440,200,480,240]);
add_line('myModel','Carrier Gain/1','Rate Transition/1');
add_line('myModel','Rate Transition/1','Product/2');

add_block('simulink/Sources/Constant','myModel/Constant',...
          'Value','0');
set_param('myModel/Constant','position',[340,320,380,360]);
```

```
add_block('simulink/Logic and Bit Operations/Relational Operator',...
          'myModel/Relational Operator','Operator','>=');
set_param('myModel/Relational Operator','position',[540,300,580,340]);
add_line('myModel','Carrier Gain/1','Relational Operator/1');
add_line('myModel','Constant/1','Relational Operator/2');

signal=get_param('myModel/Carrier Gain','PortHandles');
set_param(signal.Outport(1),'Name','pos_carrier');

add_block('simulink/Sinks/Scope','myModel/ScopeA');
set_param('myModel/ScopeA','position',[740,160,780,200]);
add_line('myModel','Product/1','ScopeA/1');

signal=get_param('myModel/Product','PortHandles');
set_param(signal.Outport(1),'Name','test_output');

add_block('simulink/Sinks/Scope','myModel/ScopeB');
set_param('myModel/ScopeB','position',[740,300,780,340]);
add_line('myModel','Relational Operator/1','ScopeB/1');

signal=get_param('myModel/Relational Operator','PortHandles');
set_param(signal.Outport(1),'Name','cmp_output');

% Specify the ERT system target file
set_param('myModel', 'SystemTargetFile', 'ert.tlc');

% Configure an XCP-based external mode simulation
set_param('myModel', 'SimulationMode', 'external');

set_param('myModel', 'SolverType', 'Fixed-step');
set_param('myModel', 'ExtMode', 'on');

cs = getActiveConfigSet('myModel');
index = Simulink.ExtMode.Transports.getExtModeTransportIndex(cs, 'XCP on TCP/IP');
set_param('myModel', 'ExtModeTransport', index);

set_param('myModel', 'GenerateSampleERTMain', 'on');
set_param('myModel', 'TargetOS', 'BareBoardExample');

set_param('myModel', 'ReturnWorkspaceOutputs', 'off');

% Save the model
save_system('myModel');
```



2 Treat each discrete rate as a separate task.

```
set_param(gcs, 'EnableMultiTasking', 'on');
```

3 Enable code execution profiling.

```
set_param(gcs, 'CodeExecutionProfiling', 'on');
set_param(gcs, 'CodeProfilingInstrumentation', 'Detailed');
set_param(gcs, 'CodeProfilingSaveOptions', 'AllData');
```

4 Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

5 Start the simulation.

```
set_param(gcs, 'SimulationCommand', 'start');
```

Run-Time Visualization

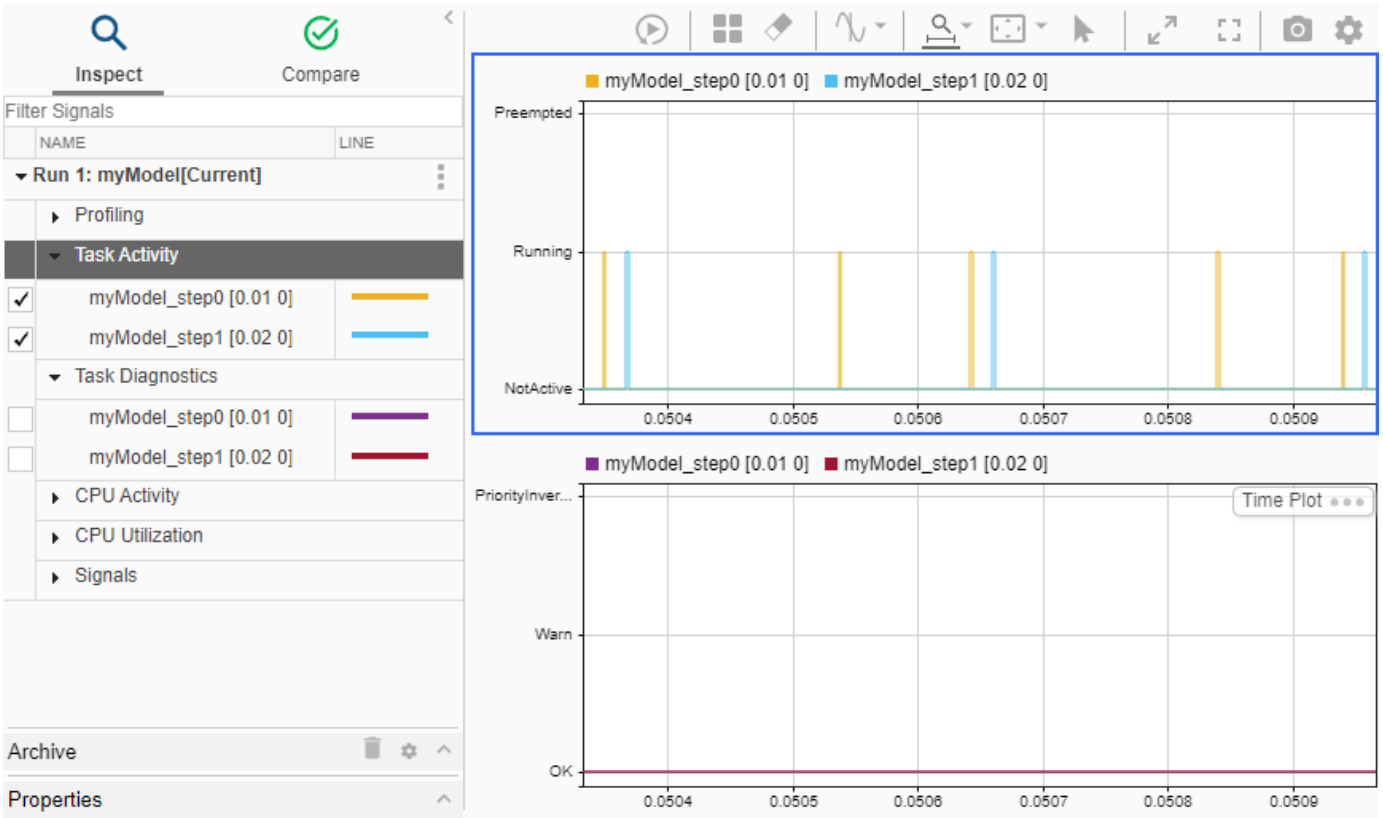
In the Simulation Data Inspector, the **Inspect** pane provides display groups. This table describes the profiling information that is available.

Display Group	Information Available
Profiling	For each task, the variation of execution time over simulation time.
Task Activity	For each task, the variation of its state over time. A task can be in one of these states: • `NotActive` — Task has not started. • `Running` — Task is running on a processor core. • `Preempted` — Execution of task is preempted by another task that has a higher priority.

Display Group	Information Available
Task Diagnostics	For each task, the variation of task diagnostic values over simulation time. A task diagnostic can have one of these values: • `OK` — No scheduling issue detected. • `Warn` — Warning that the task is preempted by a task with lower priority. • `PriorityInversion` — Priority inversion detected The software uses a rate-monotonic analysis to determine task priorities. Task diagnostics are available only for target applications that are generated to run on: • Your development computer • Target devices that are Linux-based.
CPU Activity	For each CPU core, the names of tasks that run on the CPU core over simulation time. At any instant, if no task is running on the CPU core, the value `idle` is displayed.
CPU Utilization	For each CPU core that tasks run on, normalized utilization over simulation time. Utilization values range between 0.0 and 1.0. For example, a value of 0.5 indicates that the CPU core is idle for 50% of the time.

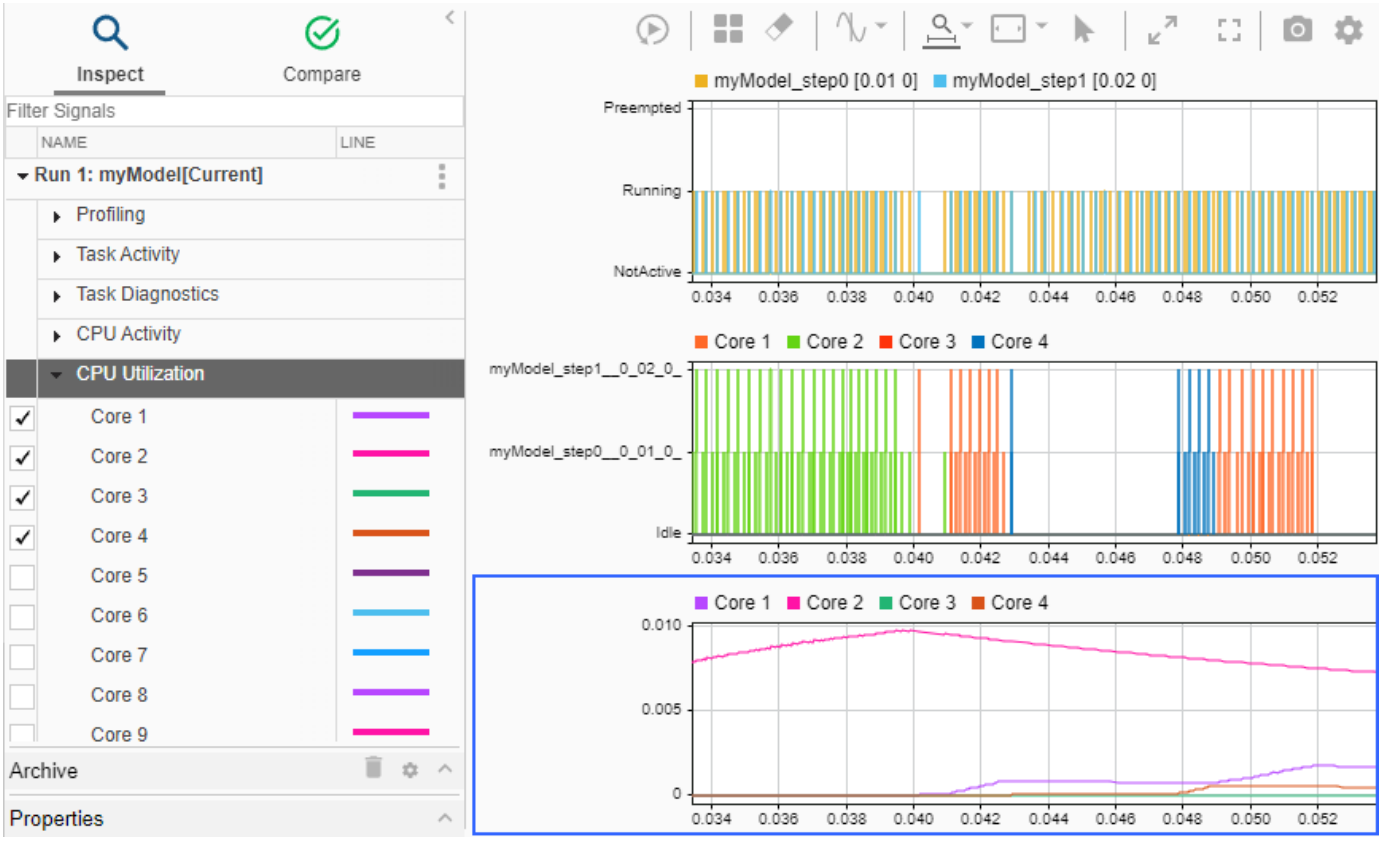
To monitor task preemption and priority inversion:

1. Select a subplot. Then, on the **Inspect** pane, from the **Task Activity** display group, select the **myModel_step0 [0.01 0]** and **myModel_step1 [0.02 0]** check boxes.
2. Select the next subplot. Then, on the **Inspect** pane, from the **Task Diagnostics** display group, select the **myModel_step0 [0.01 0]** and **myModel_step1 [0.02 0]** check boxes.
3. Using the **Zoom in Time** (**Ctrl+Shift+T**) control, view a specific time section.



To monitor CPU core activity and utilization:

1. Select a subplot. On the **Inspect** pane, from the **CPU Activity** display group, select, for example, the **Core 1**, **Core 2**, **Core 3**, and **Core 4** check boxes.
2. Select the next subplot. On the **Inspect** pane, from the **CPU Utilization** display group, select, for example, the **Core 1**, **Core 2**, **Core 3**, and **Core 4** check boxes.
3. Using the **Zoom in Time** (**Ctrl+Shift+T**) control, view a specific time section.

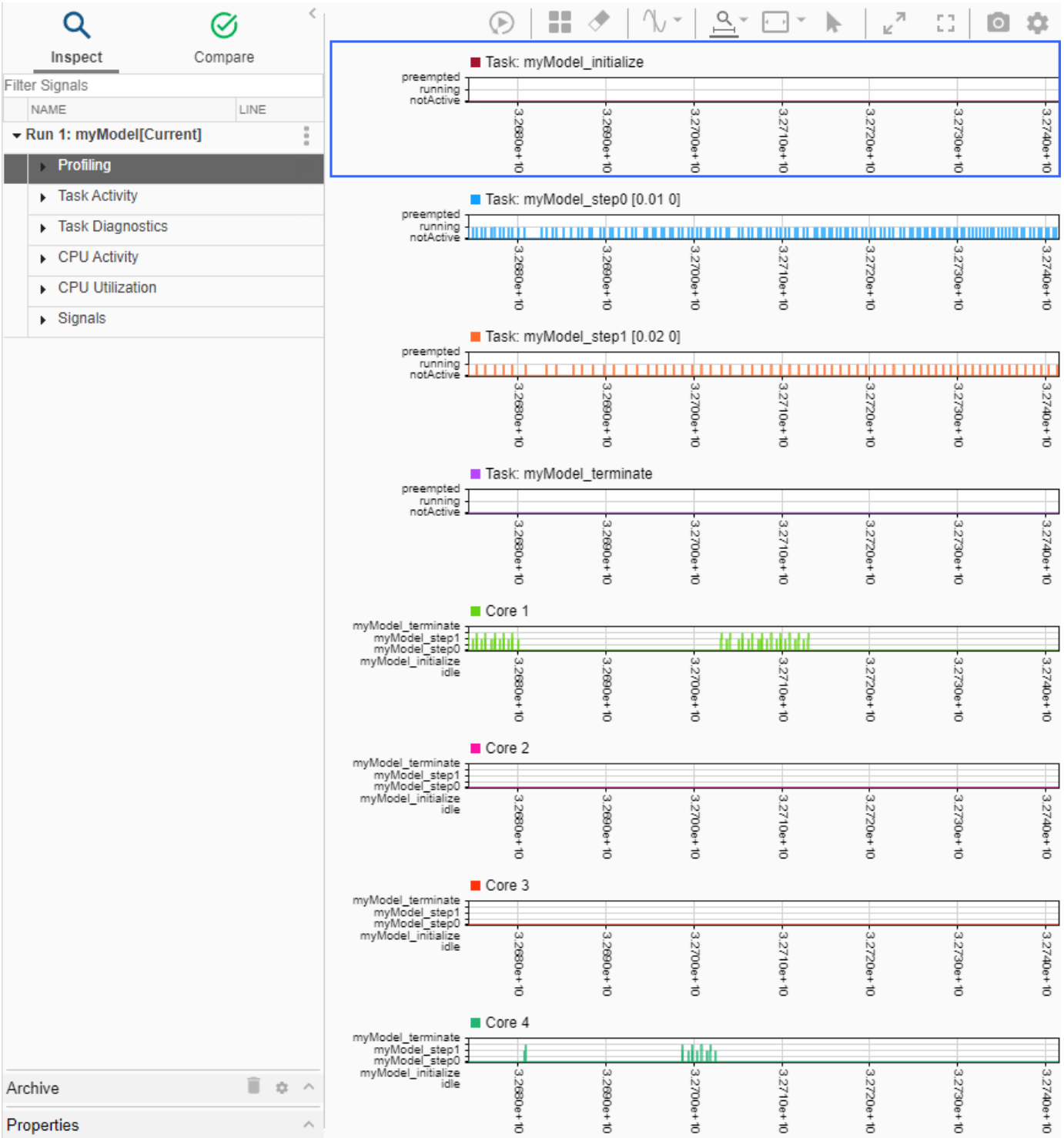


Offline Visualization

After the completion of the simulation, you can use the `schedule` function to regenerate displays (except for task diagnostics). For this example, run:

```
schedule(executionProfile)
```

The function adds the task activity, CPU core activity, and CPU core utilization displays to the **Profiling** group. By default, the function produces eight subplots.

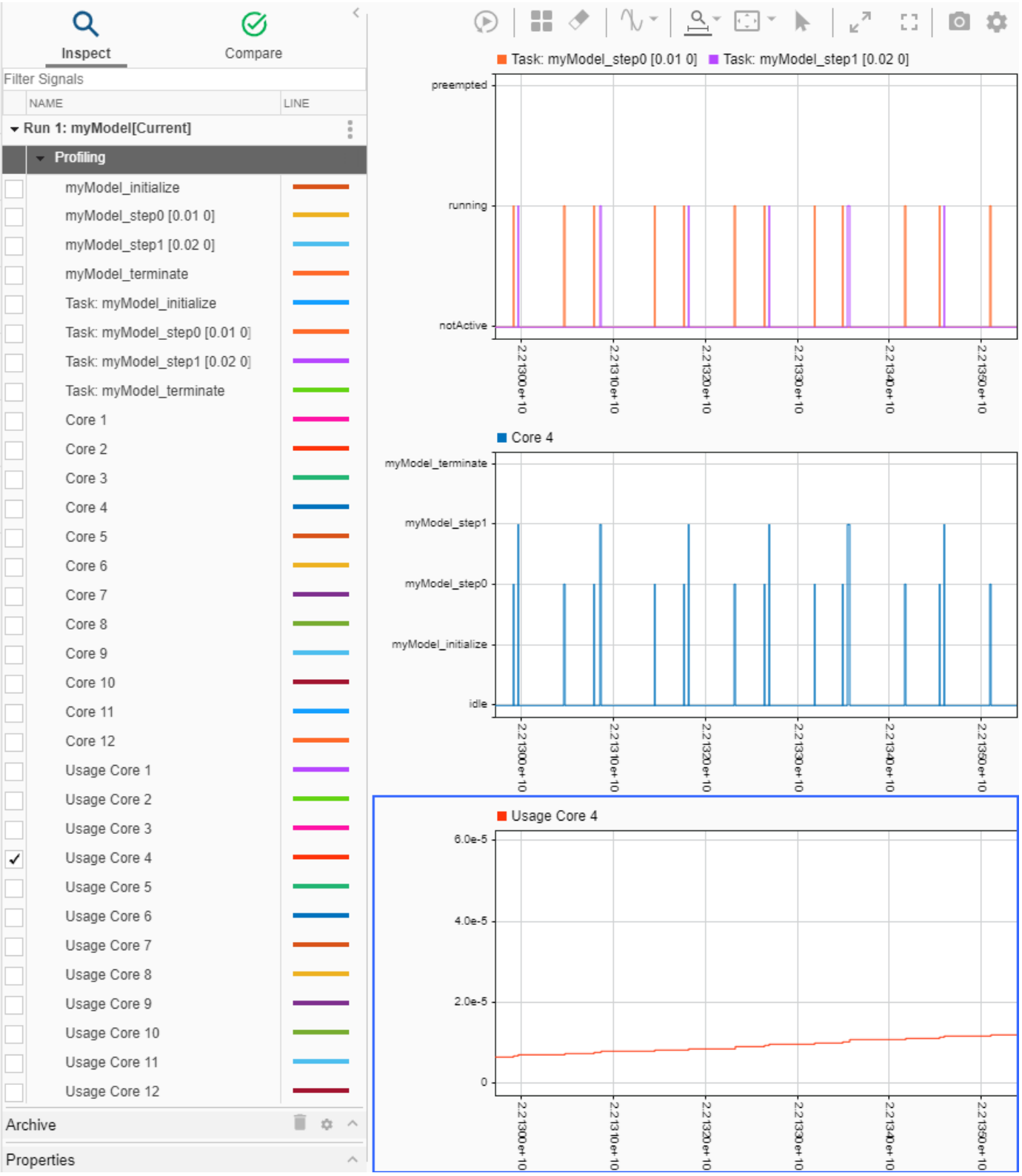


To visualize specific task scheduling and related CPU activity for specific tasks, you can use `Simulink.sdi` functions. For example, to observe task activity for `myModel_step0` and `myModel_step1` and CPU core activity and utilization for `Core 4`:

1. Clear the displays and specify three subplots.

```
Simulink.sdi.clearAllSubPlots;
Simulink.sdi.setSubPlotLayout(3,1);
```

2. In the Simulation Data Inspector, perform these actions:
   - Click subplot 1, and then select the **Task: myModel_step0 [0.01 0]** and **Task: myModel_step1 [0.02 0]** check boxes.
   - Click subplot 2, and then select the **Core 4** check box.
   - Click subplot 3, and then select the **Usage Core 4** check box.
3. Using the **Zoom in Time** (**Ctrl+Shift+T**) control, view activity for a specific time section.


Inspect
Compare
Filter Signals
NAME
LINE
Run 1: myModel[Current]
Profiling
myModel_initialize
myModel_step0 [0.01 0]
myModel_step1 [0.02 0]
myModel_terminate
Task: myModel_initialize
Task: myModel_step0 [0.01 0]
Task: myModel_step1 [0.02 0]
Task: myModel_terminate
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Usage Core 1
Usage Core 2
Usage Core 3
Usage Core 4
Usage Core 5
Usage Core 6
Usage Core 7
Usage Core 8
Usage Core 9
Usage Core 10
Usage Core 11
Usage Core 12
Archive
Properties
Task: myModel_step0 [0.01 0]
Task: myModel_step1 [0.02 0]
preempted
running
notActive
Core 4
myModel_terminate
myModel_step1
myModel_step0
myModel_initialize
idle
Usage Core 4
6.0e-5
4.0e-5
2.0e-5
0

See Also

`schedule`

Related Examples

- “External Mode Simulation by Using XCP Communication” on page 60-8
- “Create Execution-Time Profile for Generated Code” on page 81-2
- Simulation Data Inspector

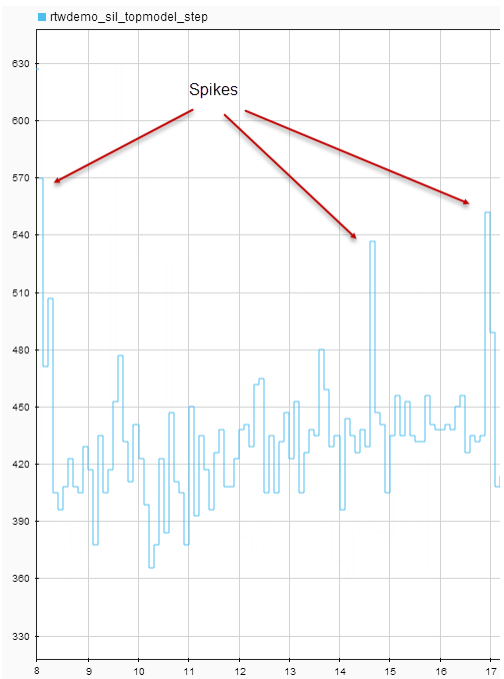
Tips and Limitations

Triggered Model Block

Consider the case where a triggered Model block is configured to run in the SIL or PIL simulation mode. The software generates one execution-time measurement each time the referenced model is triggered to run. If there are multiple triggers in a single time step, the software generates multiple measurements for the triggered Model block. Conversely, if there is no trigger in a given time step, the software generates no time measurements.

Outliers in Execution-Time Profiles

When you run a SIL simulation with execution time profiling enabled, you might see spikes in execution-time measurements.



The spikes are due to process preemption that occurs with a multitasking host operating system. If the operating system preempts the SIL process and runs another process, the measured execution time includes the time during which the SIL process is suspended. With a PIL simulation, you do not see spikes because code execution on the target is not preempted.

Counter wrapping produces execution-time measurements that are smaller than expected. For SIL, the counter wraps when an execution-time period is greater than 2^{64} ticks (2^{32} ticks if the MEX compiler is LCC). For PIL, the wrapping point depends on the timer you specify and can be 2^8, 2^{16}, 2^{32}, or 2^{64} ticks.

Consider a PIL example where the timer frequency is 20 MHz. For a 32-bit timer, wrapping occurs when the execution-time period is greater than `1/(20e6)*(2^32-1)`, that is, 214.7 s. However, for a 16-bit timer, the point at which wrapping occurs is 0.0033 s.

For a real-time, multi-core application, the software accommodates synchronization discrepancies when recording timer values for different cores, which effectively reduces the timer measurement range.

Hardware-Specific Timer

If your target configuration does not already specify a timer, create a timer object that provides details of the hardware-specific timer and associated source files:

- For SIL simulation, the timer word length is 64 bits.
- For PIL simulation, specify an unsigned 8-, 16-, 32-, or 64-bit timer. For more information, see “Specify Hardware Timer for Simulink” on page 88-45.

Effect of Instrumentation Code on Instruction Cycle

Instrumentation code increases the volume of code to run and can result in the CPU requiring more cycles to fetch code blocks from memory. This is one of the effects of instrumentation code on the fetch stage and depends on how the code is aligned in memory. If the instrumented code does not fit in the same number of blocks as code that is not instrumented, the prefetch unit must transfer more blocks.

Task Context Switching Due to Preemption

Profiling instrumentation is intrusive and affects the quantity that it is meant to measure. Therefore, the design goal is to maximize code understanding with a minimum of instrumentation. For example, with a real-time system, there can be task context switches due to preemption. These context switches are not explicitly instrumented. To record the start and end of each task, the software must infer context switches from instrumentation. As a result, the software reports behavior that is an estimate. The estimate is subject to error because of incomplete instrumentation within the kernel.

In some cases, when the software cannot accurately determine behavior, the software generates a warning:

```
Warning: Analysis unsuccessful for one or more profiling data points. ...
```

For example, the software might generate this warning when not all mutex take system calls (associated with rate transitions) are instrumented. In the case of Simulink Real-Time, this situation might arise if you generate code for a model reference hierarchy without enabling function profiling for all referenced models (`set_param(model, 'CodeProfilingInstrumentation', 'on')`). If a mutex take system call is not instrumented, a task context switch might occur that is not visible to the execution profiling analysis.

In other cases, although the software cannot accurately determine behavior, the software does *not* generate a warning.

Subsystem Code Reuse

You cannot generate execution-time profiles for function call sites within subsystem code that is reused across a model or multiple models. For information about subsystem code reuse, see “Generate Reusable Code from Library Subsystems Shared Across Models”.

Cannot Load Execution-Time Measurements from Previous Release

You cannot load execution-time measurements saved with a previous release. For example, using R2014a, you save workspace variables to a MAT-file. One of the workspace variables contains execution-time measurements. In R2015b, if you try to load the MAT-file, you see this error:

```
Format of execution profiling data is invalid. This error can occur if
you load data from a previous release. Loading data from a previous
release is not supported.
```

Stack Usage Profiling for Code Generated from Simulink Models

To determine the size of stack memory that is required to run generated code, you can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation that generates a stack usage profile. The SIL or PIL simulation instruments the generated code and uses the instrumentation to calculate stack usage. When the simulation is complete, you can view a code stack profiling report that shows minimum, average, and maximum memory demand. Using the Simulation Data Inspector, you can observe streamed stack usage information during and after the simulation. The profiles that you generate enable you to observe the effect of compiler optimization and data input on stack usage.

You can generate stack usage profiles by using the top-model or Model block SIL/PIL workflows. Subsystem SIL/PIL workflows do not support stack usage profiling.

Benefits of Dynamic Stack Usage Profiling

Static stack usage analysis has some limitations. For example, it does not consider:

- Memory optimization performed by a compiler. For example, variables that are optimized away or stored in registers.
- Additional memory that the compiler uses in a function call. Compilers store the address to which the execution returns after the function call.
- Memory alignment that the compiler applies.

The dynamic stack usage profiling provided by a SIL or PIL simulation does not have the listed limitations because the simulation calculates the actual memory usage of the compiled code.

Configure Stack Usage Profiling

To run a SIL or PIL simulation that generates stack usage metrics:

1. In the Simulink Editor, open your model.
2. On the **Apps** tab, click **SIL/PIL Manager**.
3. In the **Mode** section, select **SIL/PIL Simulation Only**.
4. In the **Prepare** section, specify **System Under Test** and, if required, **SIL/PIL Mode**. A PIL simulation requires a driver implementation that obtains stack usage data from the target hardware. See “Obtain Stack Usage Data From Target Hardware.”.
5. To enable stack usage profiling, in the **Prepare** section:
   a. Open the **Settings** gallery.
   b. Under **Time Profiling**, click the **Task Profiling** button and **Functions** button off.
   c. Under **Coverage**, click the **Coverage Collection** button off.
   d. Under **Stack Profiling**, click the **Stack Profiling** button on. This action selects the **Measure task stack usage** configuration parameter, which provides stack usage measurements for each generated task. The top model setting for the parameter overrides the corresponding setting in referenced models.

6 Click , which opens the Configuration Parameters dialog box. On the **Data Import/Export** pane, clear the **Single simulation output** check box.

7 To view streamed stack usage during the simulation, open the Simulation Data Inspector. In the **Results** section, click .

8 In the **Run** section, click **Run SIL/PIL**.

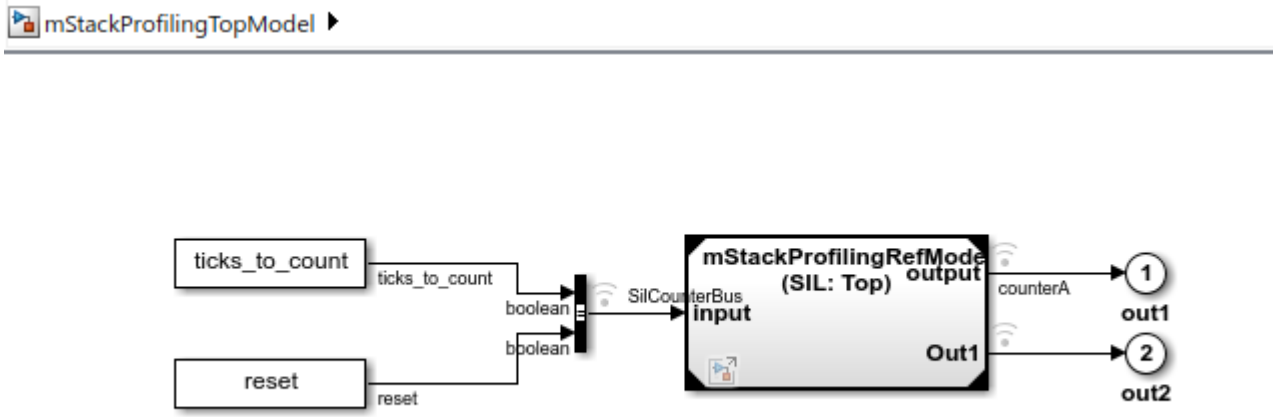
In the MATLAB base workspace, the simulation generates a variable with the default name, `stackProfile`, which stores the stack usage data. You can specify an alternative name through the **Stack workspace variable** configuration parameter.

View Stack Usage Metrics for Tasks and Functions

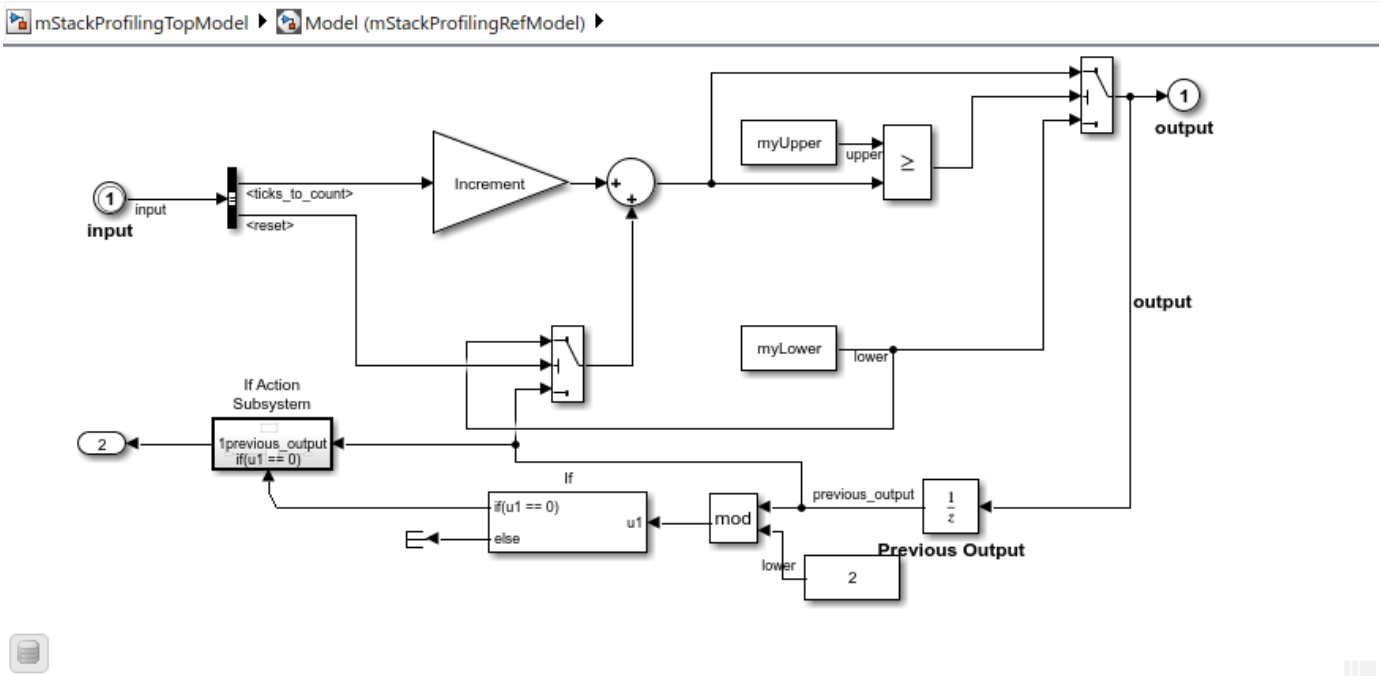
Suppose you run a SIL simulation of this model:

mStackProfilingTopModel

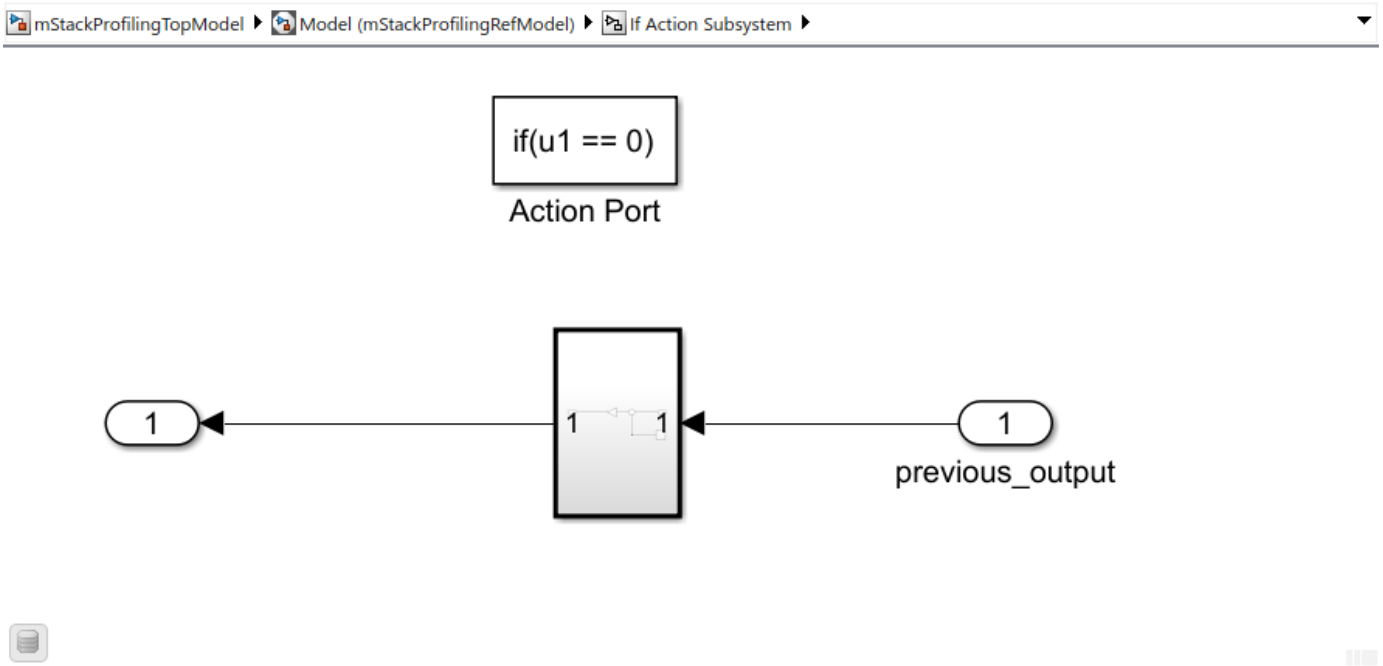
The top model, `mStackProfilingTopModel`, is a modification of the shipped model `SILModelBlock`.



The referenced model, `mStackProfilingRefModel`, is a modification of the shipped model `SILCounter`.



The If Action Subsystem block contains this subsystem.



When the SIL simulation is complete, the code stack profiling report opens. See “Code Stack Profiling Report” on page 81-60.

To view stack usage metrics in the Code Profile Analyzer:

1 Click the **SIL/PIL** tab.

2 In the **Results** gallery, under **Stack Profiling Results**, click **Code Profile Analyser**. The app displays information in the **Stack Memory** and **Summary** panels.

In the **Stack Memory** panel, the **Profiled Tasks** view shows stack usage metrics for these generated code sections:

- The model initialization function `mStackProfilingRefModel_initialize`.
- The task represented by the step function `mStackProfilingRefModel_step`.

Profiled Tasks

Section	Minimum Usage	Average Usage	Maximum Usage	Minimum Function-Call Depth	Maximum Function-Call Depth	Calls
mStackProfilingRefModel_initialize	48	48	48	1	1	1
mStackProfilingRefModel_step	64	89	112	1	2	101

For each code section, the table provides this information:

- **Minimum Usage** — Minimum stack memory usage in bytes.
- **Average Usage** — Average stack memory usage in bytes
- **Maximum Usage** — Maximum stack memory usage in bytes
- **Minimum Function-Call Depth** — Minimum number of nested function calls.
- **Maximum Function-Call Depth** — Maximum number of nested function calls.
- **Calls** — Number of times that the generated code section is called.

The **Profiled Functions** view shows the memory demand of each executed function.

Profiled Functions

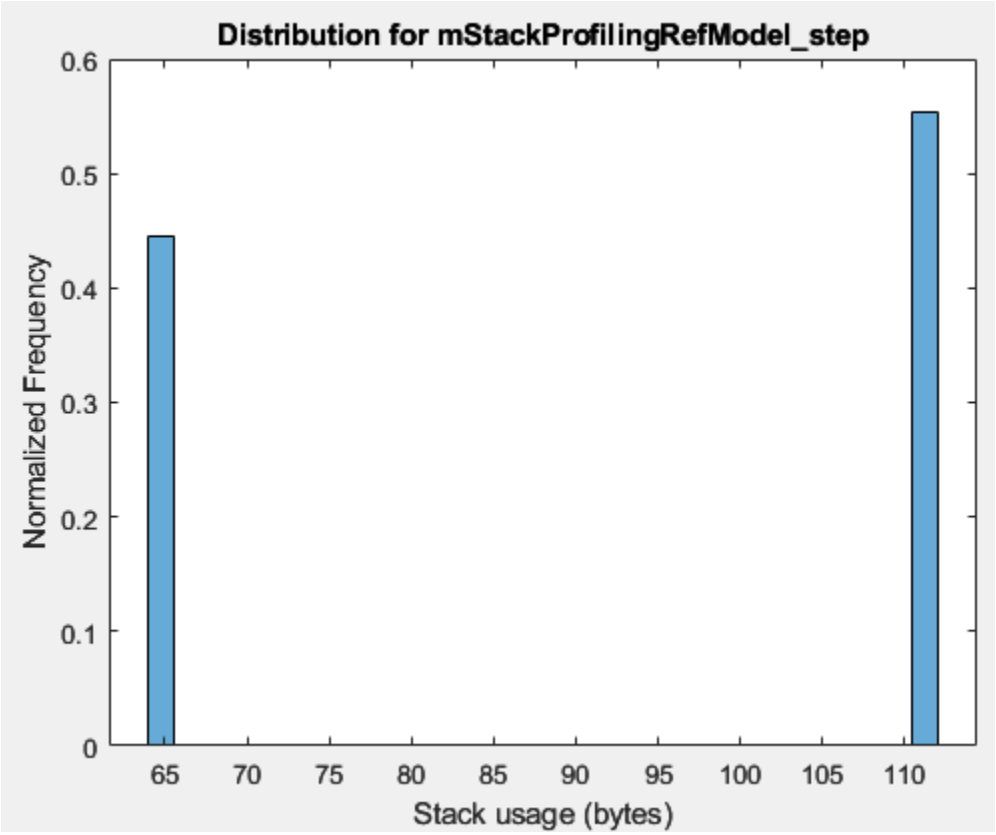
Function	Memory Usage	Calls
Subsystem	48	51
mStackProfilingRefModel_step	64	101
mStackProfilingRefModel_initialize	48	1

For each function, the table provides this information:

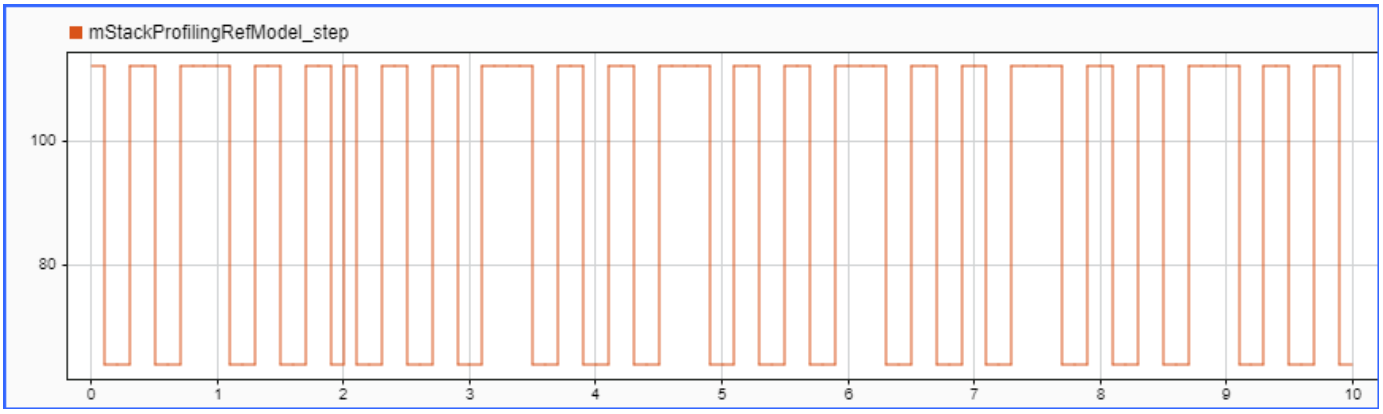
- **Function** — Function name in generated code
- **Memory Usage** — Required stack memory in bytes
- **Calls** — Number of times that function is called

To view the stack usage distribution for a task:

1 In the **Profiled Tasks** view, click a task row.
2 In the **Results** section, click **Generate Distribution**.



To view the variation of stack usage over the simulation, in the Results section, click Export to SDI.



Stack memory usage for `mStackProfilingRefModel_step` fluctuates between 112 and 64 bytes. The fluctuation occurs because the nested function generated from the If Action Subsystem block is called by `mStackProfilingRefModel_step` only when the signal `previous_output` has an even value.

```
…
if ((int32_T)(uint8_T)((int32_T)rtb_previous_output % 2) == 0) {
    /* Outputs for IfAction SubSystem: '<Root>/If Action Subsystem' incorporates:
     *  ActionPort: '<S1>/Action Port'
     */

    /* Outputs for Atomic SubSystem: '<S1>/Subsystem' */
```

```
    Subsystem(rtb_previous_output, &rtY.Out1);

    /* End of Outputs for SubSystem: '<S1>/Subsystem' */
    /* End of Outputs for SubSystem: '<Root>/If Action Subsystem' */
  }
…
```

To view the call to a function:

1. In the **Profiled Functions** view, click the row that contains the function, for example, `mStackProfilingRefModel_step`.
2. In the **Results** section, click **Highlight Code**. The code generation report shows the call in the generated code.

```
/* Model step function */
void mStackProfilingRefModel_step(void)
{
  uint8_T rtb_previous_output;

  /* UnitDelay: '<Root>/Previous Output' */
  rtb_previous_output = rtDWork.PreviousOutput_DSTATE;

  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/C1'
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/input'
   *  Sum: '<Root>/Add'
   */
  if (rtU.input.reset) {
    rtDWork.PreviousOutput_DSTATE = 40U;
  }
```

To trace the model component from which code is generated, in the **Results** section, click **Open Model**.

Compare Stack Usage Against Baseline

You can compare stack usage values from the current simulation against values from a baseline simulation.

1. On the toolstrip, in the **Analysis** section, click **Comparison**.
2. From the **Compare current data against** drop-down list, select the workspace variable that contains results from the baseline simulation. You can update the drop-down list by clicking the **Refresh list** button.
3. Click the **Compare** button.

The panel provides these views:

- **Task Comparison** — A percentage comparison of task stack usage values for the two simulations. You also see a comparison for the number of nested, function calls in each task.

- **Function Comparison** — A percentage comparison of function stack usage values for the two simulations. You also see a comparison for the number of calls to each function.

The table cells are colored:

- Green if the current execution-time metric is less than the baseline value.
- Yellow if the current execution-time metric is greater than the baseline value.

If the workspace variable for the baseline simulation does not contain data for a code section, the panel displays `NaN`.

Task Comparison (%)

Section	Minimum Usage	Average Usage	Maximum Usage	Minimum Function-Call Depth	Maximum Function-Call Depth	Calls
mStackProfilingRefModel_initialize	0	0	0	0	0	0
mStackProfilingRefModel_step	0	1.1364	0	0	0	0

Function Comparison (%)

Function	Memory Usage	Calls
Subsystem	0	2
mStackProfilingRefModel_step	0	0
mStackProfilingRefModel_initialize	0	0

Summary Panel (Stack Usage)

The **Summary** panel provides miscellaneous simulation information.

Summary

Profiled component:	mStackProfilingTopModel
Workflow:	SIL/PIL
Environment:	Simulink
Word size (bits):	32
Largest atomic integer:	Long
Pointer size (bits):	64
Max stack allowed:	Not set
Driver type:	Custom
Time created:	17-Oct-2022 13:17:25

- **Profiled component** — Simulink model that is the source of generated code
- **Workflow** — Type of code execution

- **Environment** — Simulation environment
- **Word size (bits)**, **Largest atomic integer**, and **Pointer size (bits)** — Values of configuration parameters `ProdWordSize` , `ProdLargestAtomicInteger`, and `ProdBitPerPointer` respectively. Hardware characteristics that indicate how the target compiler views memory and aligns variables.
- **Max stack allowed** — Value of `MaxStackUsageAllowed` property in workspace variable that contains stack usage measurements. See “Code Stack Profiling Report” on page 81-60.
- **Driver type** — Driver used to obtain stack usage data from the target hardware. Custom or Generic.
- **Time created** — Date and time when workspace variable with stack usage metrics was created.

Code Stack Profiling Report

You can also view stack usage metrics for the generated code through the HTML report that opens at the end of the simulation.

Code Stack Profiling Report for mStackProfilingTopModel/Model

The code stack profiling report provides metrics based on data collected from a SIL or PIL execution. Stack usage is calculated from data recorded by instrumentation probes added to the SIL or PIL test harness and inside the code generated for each component. See Code Stack Profiling for more information.

1. Summary

Largest atomic integer type	Long
Word size (bits)	32
Pointer size (bits)	64
Stack driver	Target specific
Profiling data created	20-Oct-2022 10:03:26

2. Profiled Tasks

Task	Minimum Stack Usage in bytes	Average Stack Usage in bytes	Maximum Stack Usage in bytes	Minimum Function-Call Depth	Maximum Function-Call Depth	Calls
mStackProfilingRefModel_initialize	48	48	48	1	1	1
mStackProfilingRefModel_step	64	89	112	1	2	101

3. Profiled Functions

Function	Memory in bytes	Calls
Subsystem	48	51
mStackProfilingRefModel_step	64	101
mStackProfilingRefModel_initialize	48	1

Note If you close the report, you can reopen the report. Use one of these actions:

- On the **SIL/PIL** tab, in the **Results** gallery, under **Stack Profiling Results**, click **Generate Report**.
- In the Code Profile Analyzer, in the **Results** section, click **Generate Report**.

In section 1, the report:

- Provides information that indicates how the target compiler views memory and aligns variables.
- States whether a target-specific or generic driver obtained stack usage data from the target hardware.
- Provides a timestamp for the report.

For each task, section 2 of the report provides this information:

- **Minimum Stack Usage in bytes** –– Minimum stack usage, in bytes.
- **Average Stack Usage in bytes** –– Average stack usage, in bytes.
- **Maximum Stack Usage in bytes** –– Maximum stack usage, in bytes.
- **Minimum Function-Call Depth** –– Minimum number of nested function calls.
- **Maximum Function-Call Depth** –– Maximum number of nested function calls.
- **Calls** –– Number of times that the generated code is called.

The report provides more information through clickable icons. The specified workspace variable that contains stack usage data, for example, `stackProfile`, must be present in the base workspace.

If you click , you can view the stack usage distribution for the simulation.

If you click , you can view the variation of stack usage over the simulation.

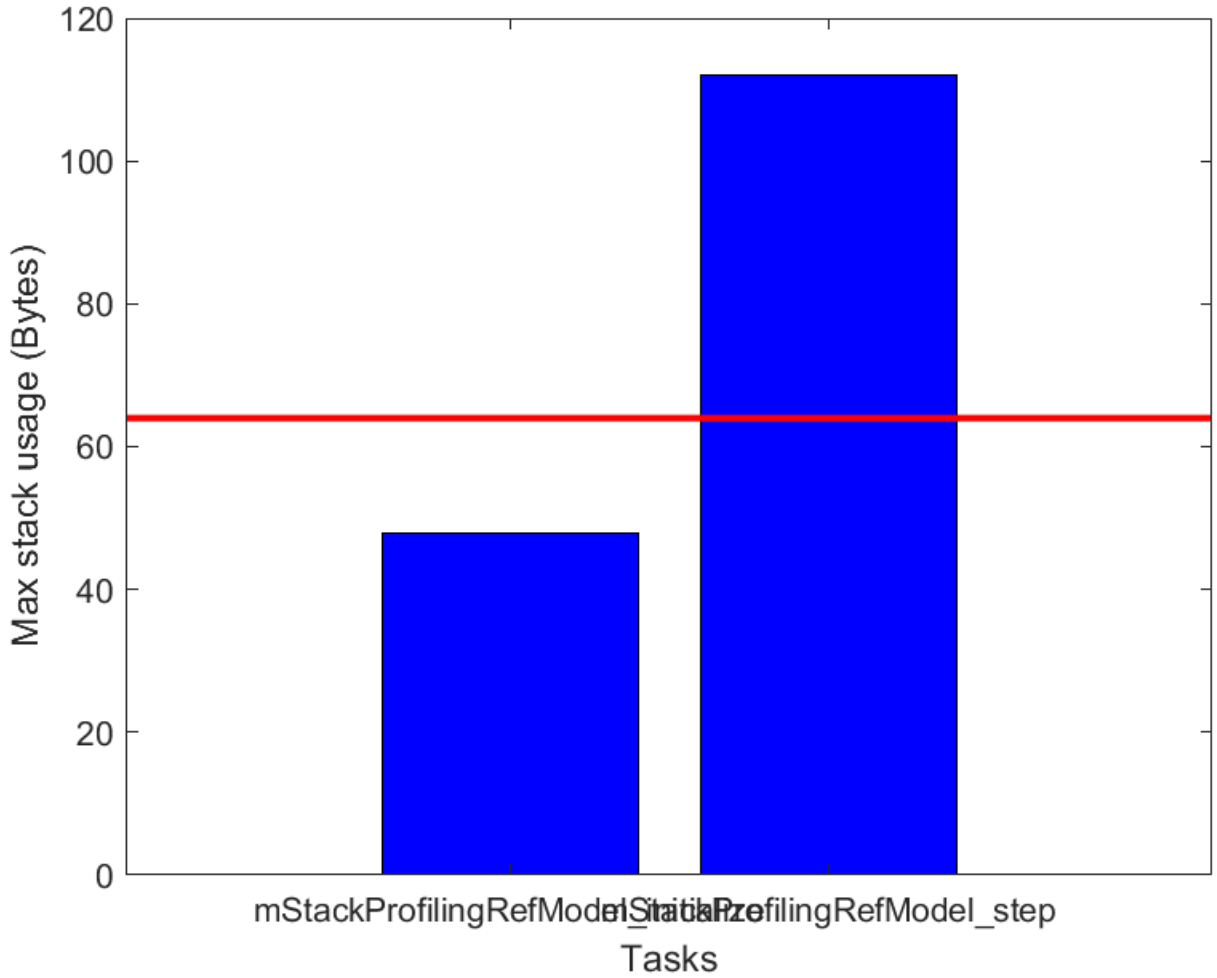
For each function, section 3 of the report provides this information:

- **Memory in bytes** –– Stack memory used, in bytes
- **Calls** –– Number of times function is called.

If you click , you can view the function call in the code generation report.

If you specify a value for **Maximum stack size (bytes)** (`MaxStackSize`), the report provides Section 4. This section displays the maximum stack use for each task with reference to the maximum stack size permitted, which the bar chart displays as a horizontal red line. If the `MaxStackSize` value is much greater than the stack usage of the tasks, the bar chart does not display the red line. To help your visual analysis of stack usage requirements, you can use the `MaxStackUsageAllowed` property of the workspace variable to respecify the maximum stack size value. For example:

```
stackProfile.MaxStackUsageAllowed = 64;
report(stackProfile);
```



Implement Driver to Obtain Stack Usage Data During PIL Simulation

A PIL simulation requires a driver implementation that obtains stack usage data from the target hardware. The driver must return the value of the stack register.

Note If you do not specify a driver for your target hardware, the PIL simulation tries to use a default generic driver.

When you set up PIL target connectivity, specify the driver through an `rtw.connectivity.Config` subclass. This code provides an example.

```
classdef overheadConnectivityConfig < rtw.connectivity.Config
  methods
    function this = customConnectivityConfig(componentArgs)

      % Create builder
      targetApplicationFramework = ...
        mypil.TargetApplicationFramework(componentArgs);
```

```
      builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
        targetApplicationFramework, '');

      % Create launcher
      launcher = mypil.Launcher(componentArgs, builder);

      % Set up communication
      hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                         componentArgs, ...
                         launcher, ...
                         rtiostreamLibTCPIP);

      % Call super class constructor to register components
        this@rtw.connectivity.Config(componentArgs,...
                                     builder,...
                                     launcher,...
                                     hostCommunicator);

      % Specify driver implementation that obtains stack usage
      % data from the target hardware
      stackUsageDriver = coder.profile.StackDriver();

      stackUsageDriver.PtrDataType = 'uint64';
      stackUsageDriver.HeaderFile = 'myHeaderFile.h';
      stackUsageDriver.SourceFile = 'mySourceFile.c';
      stackUsageDriver.IncludePaths = {'path/To/myFolder1', ...
                                       'path/To/myFolder2', ...
                                       'path/To/myFolder3'};
      stackUsageDriver.DriverFunction = 'myDriverFunction';
      this.setStackDriver(stackUsageDriver);
    end
  end
end
```

For more information about setting up PIL target connectivity, see:

- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Create PIL Target Connectivity Configuration for MATLAB” on page 90-12

See Also

`coder.profile.StackDriver`

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “Configure Processor-In-The-Loop (PIL) for a Custom Target” on page 88-113
- “Stack Usage Profiling for Code Generated From MATLAB Code” on page 82-13

Code Execution Profiling for MATLAB Coder

- “Execution Time Profiling for SIL and PIL” on page 82-2
- “Generate Execution Time Profile” on page 82-3
- “View Execution Times” on page 82-4
- “Analyze Execution Time Data” on page 82-9
- “Stack Usage Profiling for Code Generated From MATLAB Code” on page 82-13
- “Removal of Unused Class Properties in Generated C/C++ Code” on page 82-19

Execution Time Profiling for SIL and PIL

During a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can produce a profile of execution times for code generated from entry-point functions. The software calculates execution times from data that is obtained through instrumentation probes added to the SIL or PIL application.

Use the execution time profile to check whether your code runs within the required time on your target hardware:

- If code execution overruns, look for ways to reduce execution time.
- If your code easily meets time requirements, consider enhancing functionality to exploit the unused processing power.

At the end of the SIL or PIL execution, you can:

- View a report of code execution times.
- Use the Simulation Data Inspector to view and compare plots of function execution times.
- Access and analyze execution time profiling data.

See Also

Related Examples

- “Generate Execution Time Profile” on page 82-3
- “View Execution Times” on page 82-4
- “Analyze Execution Time Data” on page 82-9

Generate Execution Time Profile

Before running a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, enable execution time profiling:

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click and then click `Open existing project`. Select the project.
3. On the **Generate Code** page, click **Verify Code**.
4. Select the **Enable entry point execution profiling for SIL/PIL** check box.
5. If you also want to generate execution time profiles for functions that are called within entry-point functions, select the **Enable function execution profiling** check box.

Note Function profiling does not support the OpenMP library. Before you generate code, on the **Generate Code** page, click the arrow next to **Generate**. Then, click **More Settings**. Under **Advanced**, set **Enable OpenMP library if possible** to `No`.

On a Mac operating system, SIL and PIL execution does not support execution-time profiling for functions that are called within entry-point functions.

Or, from the Command Window, specify properties of your `coder.EmbeddedCodeConfig` object. For example:

```
% Enable entry point execution profiling
config.CodeExecutionProfiling = true;

% Enable function execution profiling
config.CodeProfilingInstrumentation = true;

% Disable OpenMP
config.EnableOpenMP = false;
```

See Also

Related Examples

- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23
- “View Execution Times” on page 82-4
- “Analyze Execution Time Data” on page 82-9

More About

- “Execution Time Profiling for SIL and PIL” on page 82-2

View Execution Times

In this section...
“Code Execution Profiling Report” on page 82-4 “Visualize Task Scheduling” on page 82-8

You can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution that produces execution-time metrics for tasks and functions in your generated code. During execution, you can use the Simulation Data Inspector to observe streamed execution times. When execution is complete, you can:

- View execution-time metrics for a profiled entry-point function.
- Open a report of execution-time metrics for all profiled functions. Through report icons, you can view, for example, the profiled code section and the execution-time distribution for each profiled function.
- Use the Simulation Data Inspector to plot and compare execution times from various execution runs.

Code Execution Profiling Report

When you run a SIL or PIL execution with execution time profiling enabled, the software generates a message in the **Test Output** tab. For example:

```
Current plot held
### Starting SIL execution for 'euclidean'
    To terminate execution: clear euclidean_sil
    Execution profiling data is available for viewing. Go to Simulation Data Inspector.
    Execution profiling report available after termination.
```

To observe streamed execution times while the execution runs, click the `Simulation Data Inspector` link.

To open the code execution profiling report:

1. Click the `clear euclidean_sil` link.

   The software terminates the execution process and displays a new link.

   ```
   ### Stopping SIL execution for 'euclidean'
       Execution profiling report: report(getCoderExecutionProfile('euclidean'))
   ```

2. Click the new link.

The report contains these sections:

1. **Summary**

   This section gives information about the creation of the report.

1. Summary

Total time	105583
Unit of time	ns
Command	report(, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	3.6e+09
Profiling data created	01-Dec-2021 19:33:56

2 Profiled Sections of Code

This section contains information about profiled code sections. In the **Section** column, expand and collapse profiled sections by clicking [+] and [–] respectively. This graphic shows fully expanded sections.

2. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls
euclidean_initialize	155	155	155	155	1
[-] euclidean	52841	52631	27382	27242	2
fabs	126	124	126	124	2
fabs	27	26	27	26	2
fabs	27	26	27	26	2
sqrt	428	322	428	322	2
fabs	183	30	183	30	430
fabs	42	27	42	27	430
fabs	32	27	32	27	430
sqrt	67	32	67	32	430
euclidean_terminate	166	166	166	166	1

The report contains time measurements for:

- The *`entry_point_fn`*`_initialize` function, for example, `euclidean_initialize`.
- The entry-point function, for example, `euclidean`.
- The *`entry_point_fn`*`_terminate` function, for example, `euclidean_terminate`.

By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the `report` command:

```
% Create workspace variable
executionProfile=getCoderExecutionProfile('kalman01');

report(executionProfile, ...
       'Units', 'Seconds', ...
```

```
        'ScaleFactor', '1e-06', ...
        'NumericFormat', '%0.3f')
```

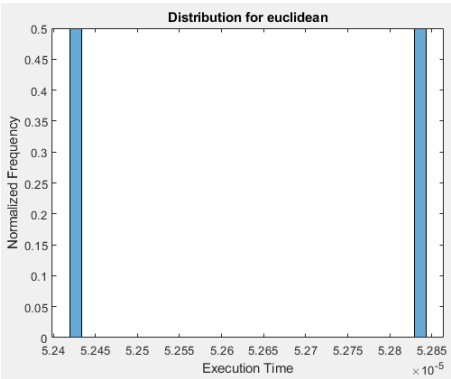
The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known. On a Windows machine, the software determines this value for a SIL simulation. On a Linux machine, you must manually calibrate the timer. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:

```
executionProfile.TimerTicksPerSecond = 1e9;
```

To view execution-time metrics for a code section in the Command Window, on the corresponding row, click the icon .

To display measured execution times, click the Simulation Data Inspector icon . You can use the Simulation Data Inspector to manage and compare plots from various executions.

To display the execution-time distribution, click the icon .



In this example, to create the histogram, the software uses these commands:

```
executionProfile=getCoderExecutionProfile('kalman01')
section=executionProfile.Sections(2);
data=section.ExecutionTimeInSeconds;
histogram(data, 30,'Normalization','probability');
```

The software creates the histogram only if the number of calls to the function is greater than one.

If the generated code contains nested functions, the execution generates pie charts that show the relative execution times of caller and called functions. To display the pie charts, click the icon for the caller function.

The following table lists the information provided in the code section profiles.

Column	Description
Section	Name of function from which code is generated.
Maximum Execution Time	Longest time between start and end of code section.
Average Execution Time	Average time between start and end of code section.

Column	Description
Maximum Self Time	Maximum execution time, excluding time in child sections.
Average Self Time	Average execution time, excluding time in child sections.
Calls	Number of calls to the code section.
	Icon that you click to display the profiled code section.
	Icon that you click to display measured execution times with Simulation Data Inspector.
	Icon that you click to display the execution-time distribution for the profiled code section.
	Icon that you click to display pie charts that show the relative execution times of caller and called functions in the generated code.

3 Execution Times in Percentages

This section provides function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code.

3. Execution Times in Percentages [hide]

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
euclidean_initialize	100%	100%	0.1468%	0.1468%
[-] euclidean	51.761%	51.761%	51.604%	99.696%
fabs	0.23539%	0.23539%	0.23468%	0.23468%
fabs	0.049612%	0.049612%	0.049461%	0.049461%
fabs	0.049084%	0.049084%	0.048935%	0.048935%
sqrt	0.61117%	0.61117%	0.60932%	0.60932%
fabs	12.094%	12.094%	12.057%	12.057%
fabs	11.086%	11.086%	11.052%	11.052%
fabs	11.109%	11.109%	11.075%	11.075%
sqrt	13.006%	13.006%	12.966%	12.966%
euclidean_terminate	100%	100%	0.1568%	0.1568%

This table describes the information that each column provides.

Column	Comparison Performed
Self Time / Caller Function	Function self time compared with the total execution time of the caller function.
Self Time / Task	Function self time compared with the total execution time of the task.
Self Time / Simulation	Function self time compared with the total execution time.
Function / Simulation	Function execution time, which includes self time and time spent in child functions, compared with the total execution time.

4 Definitions

This section provides descriptions of some metrics.

4. Definitions

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

Visualize Task Scheduling

You can use the Simulation Data Inspector to visualize task scheduling and the order of function calls. At the end of the execution, run the `schedule` function. For more information, see “Visualize Task Scheduling”.

See Also

Related Examples

- “Generate Execution Time Profile” on page 82-3
- “Analyze Execution Time Data” on page 82-9
- Simulation Data Inspector

Analyze Execution Time Data

After a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can analyze execution-time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

In the following example, you run a SIL execution and apply supplied methods to execution-time data.

Extract Execution Time Data for Kalman Estimator Code

1 Run SIL execution to generate execution time data

Open this example to obtain the files for this tutorial:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

```
openExample('ecoder/KalmanFilterExample')
```

Create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
config.GenerateReport = true; % HTML report
```

Configure the object for SIL and enable execution time profiling.

```
config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
```

If you want to generate execution time profiles for functions that are called within entry-point functions, you must also disable the OpenMP library.

```
config.CodeProfilingInstrumentation = true;
config.EnableOpenMP = false;
```

On a Mac operating system, SIL and PIL execution does not support execution-time profiling for functions that are called within entry-point functions.

Generate library code for the `kalman01` MATLAB function and the SIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

Run the MATLAB test file `test01_ui` with `kalman01_sil`. `kalman01_sil` is the SIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

You see the following message.

```
### Starting SIL execution for 'kalman01'
    To terminate execution: clear kalman01_sil
    Execution profiling data is available for viewing. Go to Simulation Data Inspector.
    Execution profiling report available after termination.
Current plot released
```

Terminate the SIL execution process. Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
   Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

2 Create workspace variable that holds execution time data

Use the `getCoderExecutionProfile` function to create a workspace variable that holds execution time profiling data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

3 Extract code sections

Use the `Sections` method.

```
allSections = executionProfile.Sections
```

The software displays the number of code sections and a list of properties.

```
allSections =

  1x3 ExecutionTimeTaskSection array with properties:

    Name
    Number
    ExecutionTimeInTicks
    SelfTimeInTicks
    TurnaroundTimeInTicks
    TotalExecutionTimeInTicks
    TotalSelfTimeInTicks
    TotalTurnaroundTimeInTicks
    MaximumExecutionTimeInTicks
    MaximumExecutionTimeCallNum
    MaximumSelfTimeInTicks
    MaximumSelfTimeCallNum
    MaximumTurnaroundTimeInTicks
    MaximumTurnaroundTimeCallNum
    NumCalls
    ExecutionTimeInSeconds
    Time
```

4 Extract execution time data from specific code section

Specify the code section that you want to examine.

```
secondSectionProfile = executionProfile.Sections(2)
```

The software displays profile data for the code section.

```
secondSectionProfile =

  ExecutionTimeTaskSection with properties:

                          Name: 'kalman01'
                        Number: 2
           ExecutionTimeInTicks: [1x300 uint64]
                SelfTimeInTicks: [1x300 uint64]
          TurnaroundTimeInTicks: [1x300 uint64]
      TotalExecutionTimeInTicks: 6641016
           TotalSelfTimeInTicks: 6641016
     TotalTurnaroundTimeInTicks: 6641016
    MaximumExecutionTimeInTicks: 48864
    MaximumExecutionTimeCallNum: 158
         MaximumSelfTimeInTicks: 48864
         MaximumSelfTimeCallNum: 158
   MaximumTurnaroundTimeInTicks: 48864
   MaximumTurnaroundTimeCallNum: 158
                       NumCalls: 300
```

```
ExecutionTimeInSeconds: [1x300 double]
                  Time: [300x1 double]
```

You can extract specific properties, for example, the name of a profiled function.

```
nameOfSection = secondSectionProfile.Name
```

The software displays the name.

```
nameOfSection =

'kalman01'
```

The following table lists the information that you can extract from each code section.

Property	Description
`Name`	Name of entry-point function
`Number`	Code section number
`ExecutionTimeInTicks`	Vector of execution times, measured in timer ticks. Each element contains the difference between the timer reading at the start and at the end of the code section. The data type is the same data type as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.
`SelfTimeInTicks`	Vector of timer tick numbers. Each element contains the number of ticks recorded for the code section, excluding the time spent in calls to child functions.
`TurnaroundTimeInTicks`	Vector of timer tick numbers. Each element contains the number of ticks recorded between the start and the finish of the code section. Unless the code is preempted, this number is the same number as the execution time.
`TotalExecutionTimeInTicks`	Total number of timer ticks recorded for the code section over the entire execution.
`TotalSelfTimeInTicks`	Total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.
`TotalTurnaroundTimeInTicks`	Total number of timer ticks recorded between the start and the finish of the profiled code section over the entire execution. Unless the code is preempted, this number is the same as the total execution time.
`MaximumExecutionTimeInTicks`	Maximum number of timer ticks recorded for a single invocation of the code section over the execution.
`MaximumExecutionTimeCallNum`	Number of call in which `MaximumExecutionTimeInTicks` occurs.
`MaximumSelfTimeInTicks`	Maximum number of timer ticks recorded for a single code section invocation, but excluding the time spent in calls to child functions.
`MaximumSelfTimeCallNum`	Number of call in which `MaximumSelfTimeInTicks` occurs.

Property	Description
`MaximumTurnaroundTimeInTicks`	Maximum number of timer ticks recorded between the start and the finish of a single invocation of the profiled code section over the execution. Unless the code is preempted, this time is the same as the maximum execution time.
`MaximumTurnaroundTimeCallNum`	Number of call in which `MaximumTurnaroundTimeInTicks` occurs.
`NumCalls`	Total number of calls to the code section over the entire execution.
`ExecutionTimeInSeconds`	Vector of execution times, measured in seconds. Each element contains the difference between the timer reading at the start and at the end of the code section. Produced only if `TimerTicksPerSecond` is set.
`Time`	Vector of execution time measurements for the code section.

See Also

Related Examples

- “Generate Execution Time Profile” on page 82-3
- “View Execution Times” on page 82-4
- Simulation Data Inspector

Stack Usage Profiling for Code Generated From MATLAB Code

To determine the size of stack memory that is required to run generated code, you can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution that generates a stack usage profile. The SIL or PIL execution instruments the generated code and uses the instrumentation to calculate stack usage. When the execution is complete, you can view a code stack profiling report that shows minimum, average, and maximum memory demand. Using the Simulation Data Inspector, you can observe streamed stack usage information during and after the execution. The profiles that you generate enable you to observe the effect of compiler optimization and data input on stack usage.

Benefits of Dynamic Stack Usage Profiling

Static stack usage analysis has some limitations. For example, it does not consider:

- Memory optimization performed by a compiler. For example, variables that are optimized away or stored in registers.
- Additional memory that the compiler uses in a function call. Compilers store the address to which the execution returns after the function call.
- Memory alignment that the compiler applies.

The dynamic stack usage profiling provided by a SIL or PIL execution does not have the listed limitations because the execution calculates the actual memory usage of the compiled code.

Generate Stack Usage Profile

Before running a software-in-the-loop on page 90-4 or processor-in-the-loop (PIL) on page 90-23 execution, enable stack usage profiling:

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** , under **Code Generation**, click the app icon.
2. To open your project, click and then click `Open existing project`. Select the project.
3. On the **Generate Code** page, click the arrow next to **Generate**.
4. In the Generate dialog box:
   a. Click **More Settings**.
   b. Click **Debugging**.
   c. Under **SIL/PIL**, select the **Enable entry point stack profiling** check box.
   d. Click **Close**.
   e. Click **Generate**.

Note Stack usage profiling does not support the OpenMP library. Before you generate code, on the **Generate Code** page, click the arrow next to **Generate**. Then, click **More Settings**. Under **Advanced**, set **Enable OpenMP library if possible** to `No`.

Or, from the Command Window, specify properties of your `coder.EmbeddedCodeConfig` object. For example:

```
% Enable entry point stack usage profiling
config.CodeStackProfiling = true;

% Disable OpenMP
config.EnableOpenMP = false;
```

View Stack Usage Profile From SIL Execution

For a MATLAB function, perform a software-in-the-loop execution that generates stack usage measurements and then use the data to determine stack memory requirements.

In your local working folder, create the function `foo.m`, which evaluates an integer factorial by using a recursive function.

```
function out = foo(in)
  coder.inline('never');
  out = factorial_(in);
end

function out = factorial_(in)
  coder.inline('never');
  if in == 1
      out = 1;
  else
      out = in * factorial_(in-1);
  end
end
```

Configure stack usage profiling for the SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;
config.VerificationMode = 'SIL';
config.CodeStackProfiling = true;
config.EnableOpenMP = false;
```

In the folder, create `test_example.m`, a MATLAB script that calls the compiled function multiple times.

```
for idx = 1: size(X,1)
    Y(idx,1) = foo(X(idx,1));
end
```

Create a test vector of integers.

```
X=[1 2 5 8]';
```

Generate library code for the `foo` MATLAB function and the SIL interface and start the SIL execution.

```
codegen -report -config config foo -args {1} -test test_example
```

```
Code generation successful: View report

Running test file: 'test_example' with MEX function 'foo_sil.mexw64'.
### Starting SIL execution for 'foo'
    To terminate execution: clear foo_sil
    Stack profiling data is available for viewing. Open Simulation Data Inspector.
    Stack profiling report available after termination.
```

To observe streamed stack usage measurements while the execution runs, click the `Simulation Data Inspector` link.

To open the code stack profiling report, terminate the SIL execution process by clicking the link `clear foo_sil`.

```
### Application stopped
### Stopping SIL execution for 'foo'
    Stack profiling report: report(getCoderStackProfile('foo'))
```

Then, click the link `report(getCoderStackProfile('foo'))`.

You can create a workspace variable that holds the execution time data.

```
stackUsageProfile= getCoderStackProfile('foo');
```

If you close the report, you can reopen the report by running:

```
report(stackUsageProfile)
```

Code Stack Profiling Report for foo

The code stack profiling report provides metrics based on data collected from a SIL or PIL execution. Stack usage is calculated from data recorded by instrumentation probes added to the SIL or PIL test harness and inside the code generated for each component. See Code Stack Profiling for more information.

1. Summary

Largest atomic integer type	Char
Word size (bits)	64
Pointer size (bits)	64
Maximum stack (bytes)	200000
Stack driver	Target specific
Profiling data created	28-Oct-2021 12:15:19

2. Profiled Tasks

Task	Minimum Stack Usage in bytes	Average Stack Usage in bytes	Maximum Stack Usage in bytes	Calls	
foo_initialize	48	48	48	1	
foo	112	304	560	4	
foo_terminate	48	48	48	1	

Section 1:

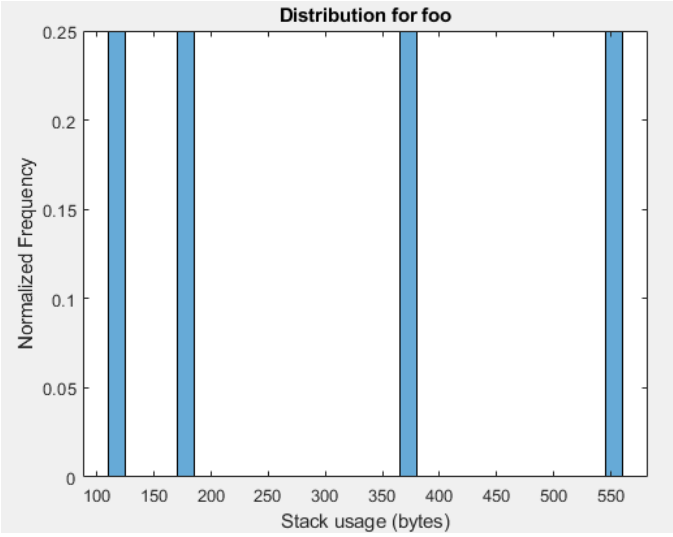
- Provides information that indicates how the target compiler views memory and aligns variables.
- States whether a target-specific or generic driver obtained stack usage data from the target hardware.
- Provides a timestamp for the report.

Section 2 provides, for each task, this information:

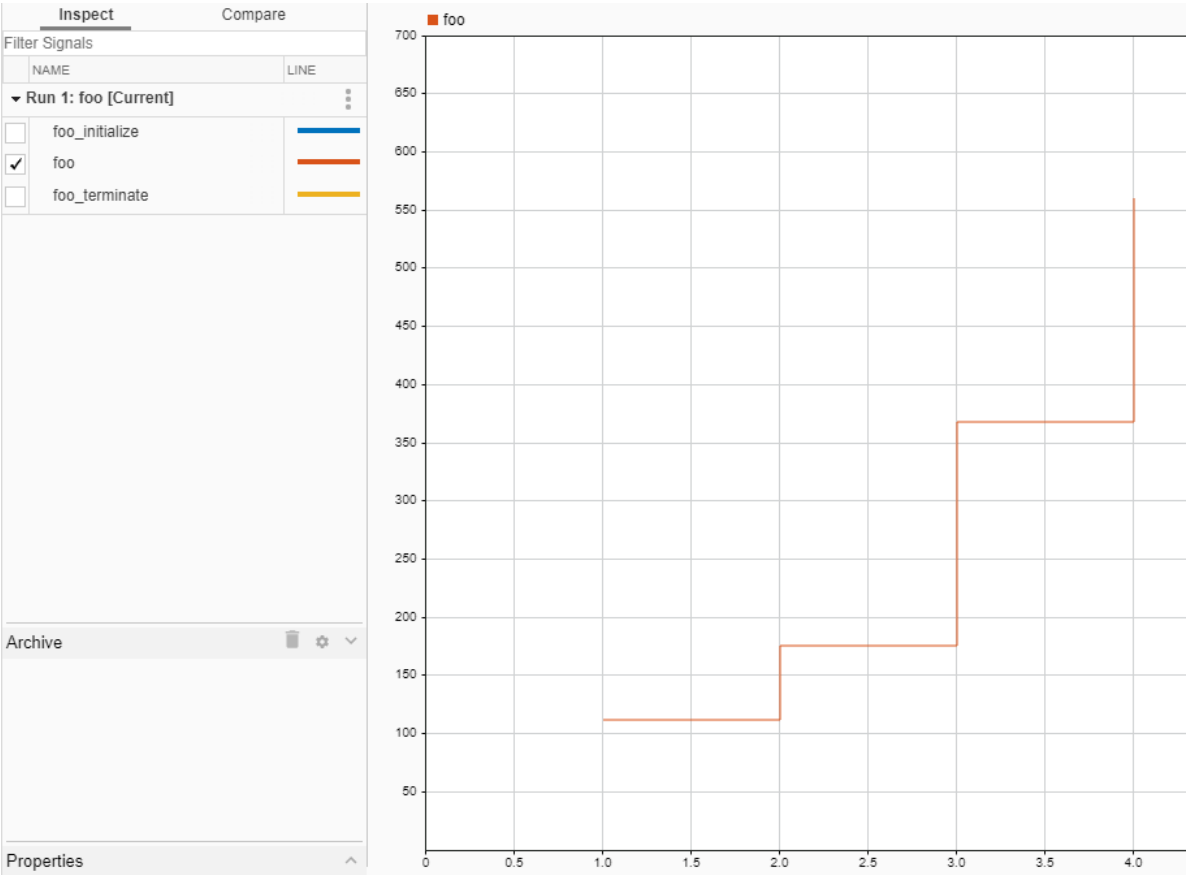
- Minimum, average, and maximum stack usage in bytes.
- Number of times the generated code is called.

The report provides more information through clickable icons. The workspace variable, for example, `stackUsageProfile`, must be present in the base workspace.

If you click , you can view the stack usage distribution for the execution.



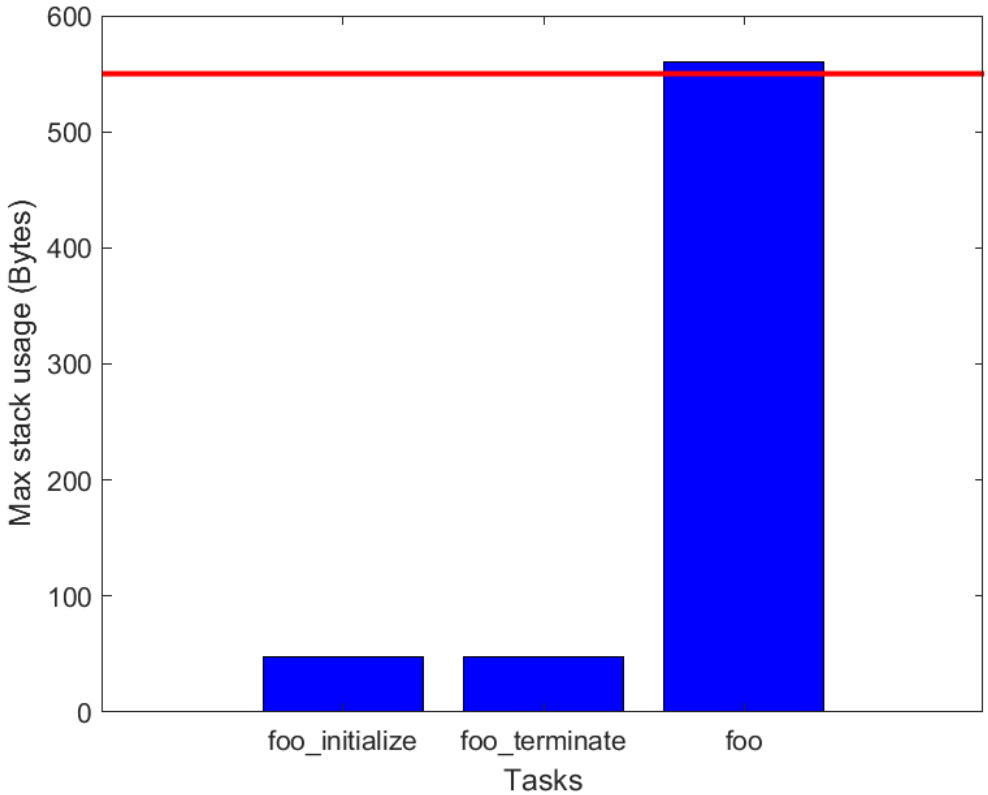
If you click , you can view the variation of stack usage over the execution.



The integer value of the input argument for `foo` increases with successive calls to the function. As the integer value increases, the number of recursive calls to the function `factorial_()` also increases, which results in the observed increases of stack memory usage.

Section 3 displays the maximum stack use for each task with reference to the maximum stack size permitted. In the SIL or PIL execution, the MATLAB Coder configuration setting `StackUsageMax` parameter specifies the maximum stack size, which the bar chart displays as a horizontal red line. If the `StackUsageMax` value is much greater than the stack usage of the tasks, the bar chart does not display a red line. To help your visual analysis of stack usage requirements, you can use the `MaxStackUsageAllowed` property of the workspace variable to respecify the maximum stack size value. For example:

```
stackUsageProfile= getCoderStackProfile('foo');
stackUsageProfile.MaxStackUsageAllowed = 550;
report(stackUsageProfile);
```



Implement Driver to Obtain Stack Usage Data During PIL Execution

A PIL execution requires a driver implementation that obtains stack usage data from the target hardware. The driver must return the value of the stack register.

Note If you do not specify a driver for your target hardware, the PIL execution tries to use a default generic driver.

When you set up PIL target connectivity, specify the driver through an `rtw.connectivity.Config` subclass. This code provides an example.

```
classdef overheadConnectivityConfig < rtw.connectivity.Config
  methods
```

```
    function this = customConnectivityConfig(componentArgs)

      % Create builder
      targetApplicationFramework = ...
        mypil.TargetApplicationFramework(componentArgs);
      builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
        targetApplicationFramework, '');

      % Create launcher
      launcher = mypil.Launcher(componentArgs, builder);

      % Set up communication
      hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                         componentArgs, ...
                         launcher, ...
                         rtiostreamLibTCPIP);

      % Call super class constructor to register components
        this@rtw.connectivity.Config(componentArgs,...
                                     builder,...
                                     launcher,...
                                     hostCommunicator);

      % Specify driver implementation that obtains stack usage
      % data from the target hardware
      stackUsageDriver = coder.profile.StackDriver();

      stackUsageDriver.PtrDataType = 'uint64';
      stackUsageDriver.HeaderFile = 'myHeaderFile.h';
      stackUsageDriver.SourceFile = 'mySourceFile.c';
      stackUsageDriver.IncludePaths = {'path/To/myFolder1', ...
                                       'path/To/myFolder2', ...
                                       'path/To/myFolder3'};
      stackUsageDriver.DriverFunction = 'myDriverFunction';
      this.setStackDriver(stackUsageDriver);
    end
  end
end
```

For more information about setting up PIL target connectivity, see:

- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Create PIL Target Connectivity Configuration for MATLAB” on page 90-12

See Also

`coder.profile.StackDriver`

Related Examples

- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23
- “Stack Usage Profiling for Code Generated from Simulink Models” on page 81-53

Removal of Unused Class Properties in Generated C/C++ Code

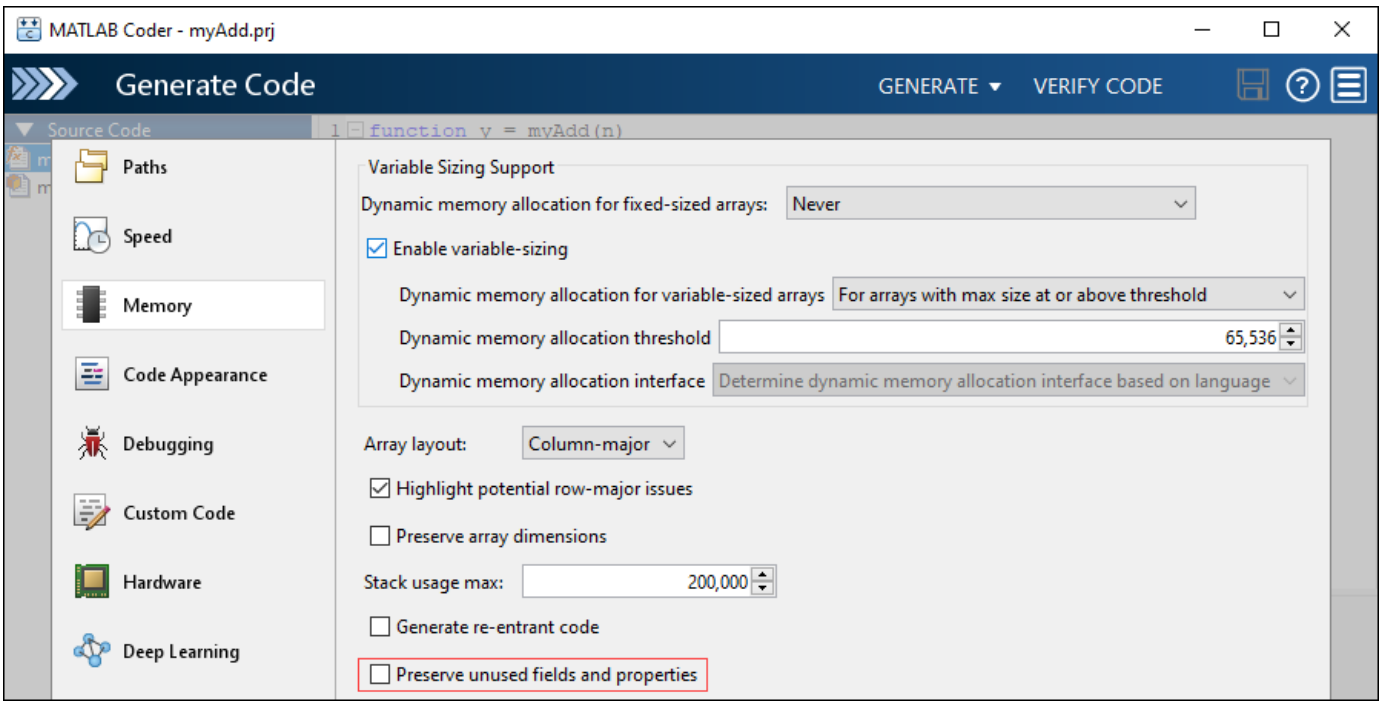
C/C++ code generated by Embedded Coder contains unused class properties or structure fields. Removing unused properties or fields helps improve the memory footprint and run time of the code. By default, the code generator removes properties or fields that are unused in the generated C/C++ code.

This feature applies to standalone code generation. You can preserve the unused properties or fields in generated C/C++ code by using the MATLAB Coder app or at the command line.

Preserve Unused Class Properties or Structure Fields by using MATLAB Coder App

To preserve unused properties or fields:

1. Open the MATLAB Coder app.
2. On the **Generate Code** page, click **More Settings**.
3. On the **Memory** tab, select the **Preserve unused fields and properties** check box.



Preserve Unused Class Properties or Structure Fields at the Command Line

To preserve the unused properties or fields, set the configuration parameter `PreserveUnusedStructFields` to `true` and use the configuration object `cfg` in the `codegen` command.

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

Examples

The following examples demonstrate the usage of the `PreserveUnusedStructFields` configuration option to generate C/C++ code.

Generate C++ Code for MATLAB Classes

This table compares the generated C++ code for `myClass`, with unused properties removed and with unused properties preserved.

MATLAB Code	Unused Class Properties Removed (default)	Unused Class Properties Preserved

`% Entry-point function`
`function y = myAdd(n)`
`  %#codegen`
`  o = myClass(n);`
`  y = o.a + o.b;`
`end`
 	
`classdef myClass`
`  properties`
`  a`
`  b`
`  c`
`  end`
 	
`  methods`
`  function obj = myClass(x)`
`  coder.inline('never')`
`  obj.a = x;`
`  obj.b = x + 1;`
`  obj.c = x + 2;`
`  end`
`  end`
`end`
	Code generation command
	
	`cfg = coder.config('lib');`
	`cfg.TargetLang = "C++";`
	`codegen myAdd -args {5} -config cfg -report`
	
	Generated C++ Code
	
	`class myClass {`
	`  public:`
	`  void init(double x);`
	`  double a;`
	`  double b;`
	`};`
	
	`void myClass::init(double x)`
	`{`
	`  a = x;`
	`  b = x + 1.0;`
	`}`
		Code generation command
		
		`cfg = coder.config('lib');`
		`cfg.PreserveUnusedStructFields = true;`
		`cfg.TargetLang = "C++";`
		`codegen myAdd -args {5} -config cfg -re`
		
		Generated C++ Code
		
		`class myClass {`
		`  public:`
		`  void init(double x);`
		`  double a;`
		`  double b;`
		`  double c;`
		`};`
		
		`void myClass::init(double x)`
		`{`
		`  a = x;`
		`  b = x + 1.0;`
		`  c = x + 2.0;`
		`}`

Generate C Code for MATLAB Structures

This table compares the generated C code for `myStruct`, with unused fields removed and with unused fields preserved.

MATLAB Code	Unused Structure Fields Removed (default)	Unused Structure Fields Preserved

`% Entry-point function`
`function out = myStruct(n)`
`  %# codegen`
`  s.a = [n n n];`
`  s.b = n+2;`
`  s.c = n;`
`  out = myAdd([s s]);`
`end`
 	
`function out = myAdd(s)`
`  coder.inline('never');`
`  out = s(1).b + s(2).b;`
`end`
	Code generation command
	
	`cfg = coder.config('lib');`
	`codegen myStruct -args {0} -config cfg -report`
	
	Generated C Code
	
	`typedef struct {`
	`  double b;`
	`} struct_T;`
	
	`/*`
	` * Arguments  : double n`
	` * Return Type  : double`
	` */`
	`double myStruct(double n)`
	`{`
	`  struct_T b_s[2];`
	`  struct_T s;`
	`  s.b = n + 2.0;`
	`  b_s[0] = s;`
	`  b_s[1] = s;`
	`  return myAdd(b_s);`
	`}`
		Code generation command
		
		`cfg = coder.config('lib');`
		`cfg.PreserveUnusedStructFields = true;`
		`codegen myStruct -args {0} -config cfg`
		
		Generated C Code
		
		`typedef struct {`
		`  double a[3];`
		`  double b;`
		`  double c;`
		`} struct_T;`
		
		`/*`
		` * Arguments  : double n`
		` * Return Type  : double`
		` */`
		`double myStruct(double n)`
		`{`
		`  struct_T b_s[2];`
		`  struct_T s;`
		`  s.a[0] = n;`
		`  s.a[1] = n;`
		`  s.a[2] = n;`
		`  s.b = n + 2.0;`
		`  s.c = n;`
		`  b_s[0] = s;`
		`  b_s[1] = s;`
		`  return myAdd(b_s);`
		`}`

Usage Notes

- When `PreserveUnusedStructFields` is set to `false`, the code generator preserves properties or fields of the entry-point function input and output arguments and values passed to `coder.ceval`.
- When `PreserveUnusedStructFields` is set to `true`, the code generator removes properties or fields whose data types are unknown.

See Also

`codegen` | `coder.inline` | `coder.EmbeddedCodeConfig`

High-throughput Applications in Simulink Coder

- “Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection” on page 83-2
- “Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models” on page 83-10
- “Generate Code for Deep Learning Networks Using MATLAB Function Block” on page 83-22
- “Generate Code for Blocks from Deep Neural Networks Library” on page 83-27
- “Workflow for Deep Learning C/C++ Code Generation for Simulink Models” on page 83-32

Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection

This example shows how to generate C++ code from a Simulink® model that performs lane and vehicle detection by using convolutional neural networks (CNNs). The example takes the frames of a traffic video as an input, outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle, and detects vehicles in the frame. This example uses a pretrained lane detection network and a pretrained vehicle detection network from the *Object Detection Using YOLO v2 Deep Learning* example of the Computer Vision Toolbox™. For more information, see “Object Detection Using YOLO v2 Deep Learning” (Computer Vision Toolbox).

This example illustrates the following concepts:

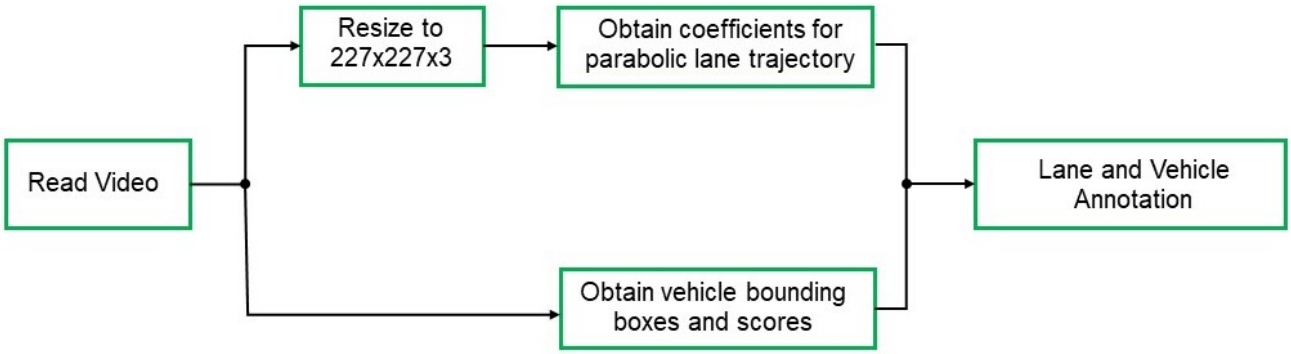
- Model the lane detection application in Simulink. First you process the traffic video by resizing to 227-by-227-by-3 and multiplying by a constant factor of 255. Subsequently, you process the traffic video by using the pretrained network loaded in the Predict block from Deep Learning Toolbox™. Finally, if the left and right lane boundaries are detected, the parabolic coefficients to model the trajectories of the lane boundaries are obtained.
- Model the vehicle detection application in Simulink by processing the traffic video using a pretrained YOLO v2 detector. This network detects vehicles in the video and outputs the coordinates of the bounding boxes for these vehicles and their confidence score.
- Configure the model for code generation.

Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).
- Refer to MKLDNN CPU Support for the list of processors that support MKL-DNN library.
- Deep Learning Toolbox™ for using the `DAGNetwork` object.
- Computer Vision Toolbox™ for video I/O operations.

Algorithmic Workflow

The block diagram for the algorithmic workflow of the Simulink model follows.



Get Pretrained Lane and Vehicle Detection Networks

This example uses the `trainedLaneNet` and `yolov2ResNet50VehicleExample` MAT files containing the pretrained networks. The files are approximately 143 MB and 98 MB in size, respectively. Download the files.

```
lanenetFile = matlab.internal.examples.downloadSupportFile('gpucoder/cnn_models/lane_detection',
vehiclenetFile = matlab.internal.examples.downloadSupportFile('vision/data','yolov2ResNet50Vehic
```

Download Test Traffic Video

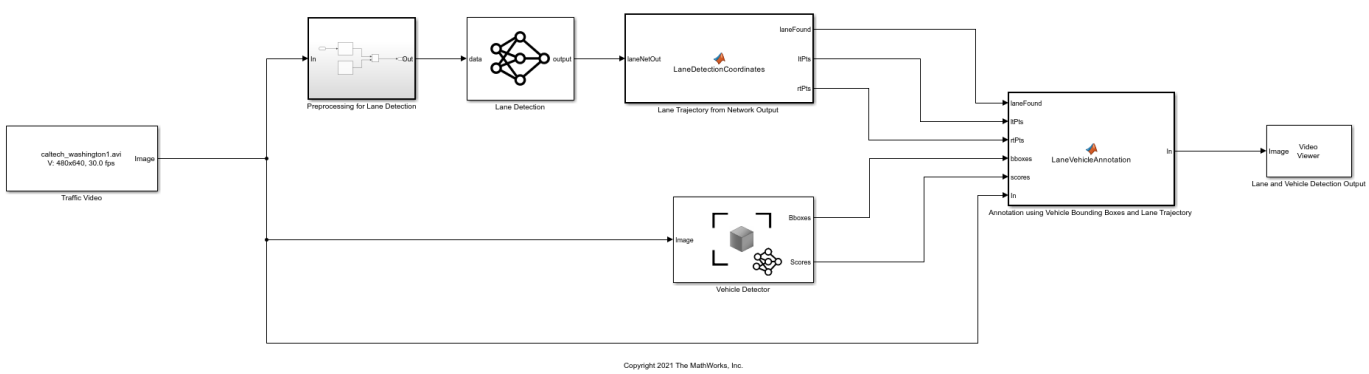
To test the model, the example uses the Caltech lanes data set. The file is approximately 16 MB in size. Download this file.

```
mediaFile = matlab.internal.examples.downloadSupportFile('gpucoder/media','caltech_washington1.av
```

Lane and Vehicle Detection Simulink Model

The following diagram shows the Simulink model for performing lane and vehicle detection on the traffic video. When the model runs, the Video Viewer block displays the traffic video with lane and vehicle annotations.

```
model='laneAndVehicleDetection';
open_system(model);
```



Set the file paths of the downloaded network model in the Predict and Detector blocks of the Simulink model.

```
set_param('laneAndVehicleDetection/Lane Detection','NetworkFilePath',lanenetFile)
set_param('laneAndVehicleDetection/Vehicle Detector','DetectorFilePath',vehiclenetFile)
```

Set the location of the test video that you load to the Simulink model.

```
set_param('laneAndVehicleDetection/Traffic Video','inputFileName',mediaFile)
```

Lane Detection

The Predict block loads the pretrained lane detection network from the `trainedLaneNet.mat` file. This network takes an image as an input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by the parabolic equation:

$$y = ax^2 + bx + c$$

Here, y is the lateral offset and x is the longitudinal distance from the vehicle. The network outputs the three parameters a, b, and c per lane. The `LaneDetectionCoordinates` MATLAB Function block defines a function `lane_detection_coordinates` that takes the output from the predict block and outputs three parameters: `laneFound`, `ltPts`, and `rtPts`. The block uses thresholding to determine if both left and right lane boundaries are both found. If are found, `laneFound` is set to be true and the trajectories of the boundaries are calculated and stored in `ltPts` and `rtPts`.

```
type lane_detection_coordinates
```

```
function [laneFound,ltPts,rtPts] = lane_detection_coordinates(laneNetOut)

% Copyright 2020 The MathWorks, Inc.

persistent laneCoeffMeans;
if isempty(laneCoeffMeans)
    laneCoeffMeans = [-0.0002    0.0002    1.4740   -0.0002    0.0045   -1.3787];
end

persistent laneCoeffStds;
if isempty(laneCoeffStds)
    laneCoeffStds = [0.0030    0.0766    0.6313    0.0026    0.0736    0.9846];
end

params = laneNetOut .* laneCoeffStds + laneCoeffMeans;

isRightLaneFound = abs(params(6)) > 0.5; %c should be more than 0.5 for it to be a right lane
isLeftLaneFound =  abs(params(3)) > 0.5;

persistent vehicleXPoints;
if isempty(vehicleXPoints)
    vehicleXPoints = 3:30; %meters, ahead of the sensor
end

ltPts = coder.nullcopy(zeros(28,2,'single'));
rtPts = coder.nullcopy(zeros(28,2,'single'));

if isRightLaneFound && isLeftLaneFound
    rtBoundary = params(4:6);
    rt_y = computeBoundaryModel(rtBoundary, vehicleXPoints);
    ltBoundary = params(1:3);
    lt_y = computeBoundaryModel(ltBoundary, vehicleXPoints);

    % Visualize lane boundaries of the ego vehicle
    tform = get_tformToImage;
    % map vehicle to image coordinates
    ltPts =  tform.transformPointsInverse([vehicleXPoints', lt_y']);
    rtPts =  tform.transformPointsInverse([vehicleXPoints', rt_y']);
    laneFound = true;
else
    laneFound = false;
end

end
```

Vehicle Detection

A YOLO v2 object detection network is composed of two subnetworks: a feature extraction network followed by a detection network. This pretrained network uses a `ResNet-50` for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to YOLO v2. The Simulink model performs vehicle detection using the Object Detector block. This block takes an image as input and outputs the bounding box coordinates along with the confidence scores for vehicles in the image.

Annotation of Vehicle Bounding Boxes and Lane Trajectory in Traffic Video

The `LaneVehicleAnnotation` MATLAB Function block defines a function `lane_vehicle_annotation`, which annotates the vehicle bounding boxes with the confidence scores. Also, if `laneFound` is true, then the left and right lane boundaries stored in `ltPts` and `rtPts` are annotated in the traffic video.

```
type lane_vehicle_annotation
```

```
function In = lane_vehicle_annotation(laneFound, ltPts, rtPts, bboxes, scores, In)

% Copyright 2020 The MathWorks, Inc.

if ~isempty(bboxes)
    In = insertObjectAnnotation(In, 'rectangle', bboxes, scores);
end

pts = coder.nullcopy(zeros(28, 4, 'single'));
if laneFound
    prevpt =  [ltPts(1,1) ltPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt ltPts(k,1) ltPts(k,2)];
        prevpt = [ltPts(k,1) ltPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    prevpt =  [rtPts(1,1) rtPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt rtPts(k,1) rtPts(k,2)];
        prevpt = [rtPts(k,1) rtPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    In = insertMarker(In, ltPts);
    In = insertMarker(In, rtPts);
end

end
```

Get Pretrained Lane and Vehicle Detection Networks

This function downloads the `yolov2ResNet50VehicleExample.mat` file.

```
getVehicleDetectionAndLaneDetectionNetworks()
```

Run Simulation

Open Configuration Parameters dialog box. On the **Simulation Target** pane, in the **Deep Learning** group, select the Target library as `MKL-DNN`.

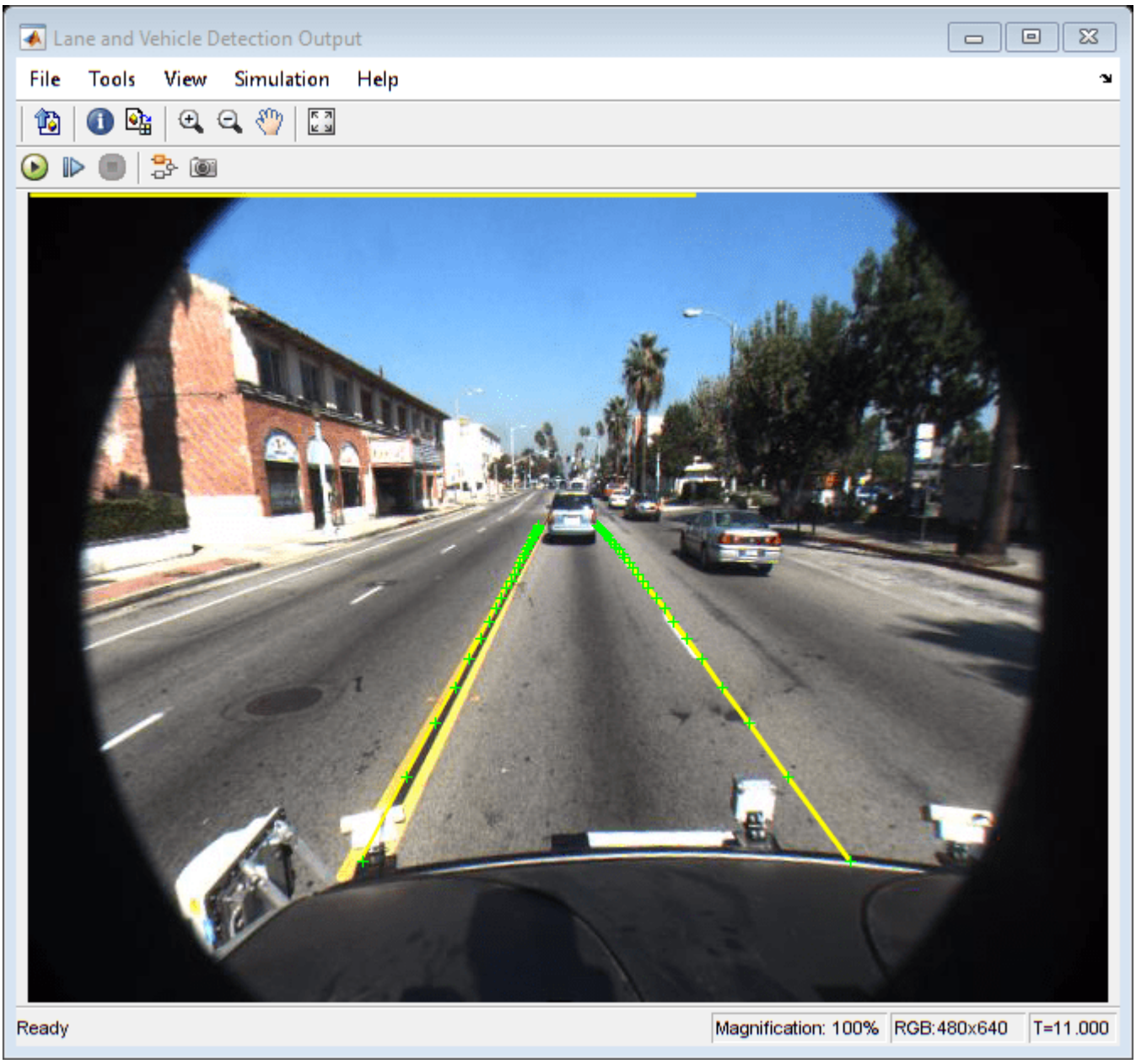
```
set_param(model,'SimDLTargetLibrary','MKL-DNN');
```

On the **Interface** pane, in the **Deep Learning** group, select the **Target library** as `MKL-DNN`.

```
set_param(model, 'DLTargetLibrary','MKL-DNN');
```

To verify the lane and vehicle detection algorithms and display the lane trajectories, vehicle bounding boxes, and scores for the traffic video loaded in the Simulink model, run the simulation.

```
set_param('laneAndVehicleDetection', 'SimulationMode', 'Normal');
sim('laneAndVehicleDetection');
```



Generate and Build Simulink Model

In the **Code Generation** pane, select the **Language** as **C++**.

```
set_param(model,'TargetLang','C++');
```

Generate and build the Simulink model using the `slbuild` command. The code generator places the files in the `laneAndVehicleDetection_grt_rtw` build subfolder under your current working folder.

```
currentDir = pwd;
status = evalc("slbuild('laneAndVehicleDetection')");
```

Generated C++ Code

The subfolder named `laneAndVehicleDetection_grt_rtw` contains the generated C++ code corresponding to the different blocks in the Simulink model and the specific operations being performed in those blocks. For example, the file `trainedLaneNet0_0.h` contains the C++ class, which contains attributes and member functions representing the pretrained lane detection network.

```
hfile = fullfile(currentDir, 'laneAndVehicleDetection_grt_rtw',...
    'trainedLaneNet0_0.h');
rtwdemodbtype(hfile,'#ifndef RTW_HEADER_trainedLaneNet0_0_h_',...
    '#endif')
```

```
#ifndef RTW_HEADER_trainedLaneNet0_0_h_
#define RTW_HEADER_trainedLaneNet0_0_h_
#include "MWOnednnTargetNetworkImpl.hpp"
#include "rtwtypes.h"
#include "MWTensorBase.hpp"
#include "MWTensor.hpp"
#include "MWCNNLayer.hpp"
#include "MWInputLayer.hpp"
#include "MWElementwiseAffineLayer.hpp"
#include "MWFusedConvActivationLayer.hpp"
#include "MWNormLayer.hpp"
#include "MWMaxPoolingLayer.hpp"
#include "MWFCLayer.hpp"
#include "MWReLULayer.hpp"
#include "MWOutputLayer.hpp"
#include "MWConvLayer.hpp"
#include "MWYoloExtractionLayer.hpp"
#include "MWSigmoidLayer.hpp"
#include "MWExponentialLayer.hpp"
#include "MWYoloSoftmaxLayer.hpp"
#include "MWConcatenationLayer.hpp"
#include "MWActivationFunctionType.hpp"
#include "MWRNNParameterTypes.hpp"
#include "MWTargetTypes.hpp"
#include "shared_layers_export_macros.hpp"
#include "MWOnednnUtils.hpp"
#include "MWOnednnCustomLayerBase.hpp"
#include "MWOnednnCommonHeaders.hpp"

class trainedLaneNet0_0
{
 public:
```

```
  boolean_T isInitialized;
  boolean_T matlabCodegenIsDeleted;
  trainedLaneNet0_0();
  void setSize();
  void resetState();
  void setup();
  void predict();
  void cleanup();
  real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
  int32_T getLayerOutputSize(int32_T layerIndex, int32_T portIndex);
  real32_T *getInputDataPointer(int32_T index);
  real32_T *getInputDataPointer();
  real32_T *getOutputDataPointer(int32_T index);
  real32_T *getOutputDataPointer();
  int32_T getBatchSize();
  ~trainedLaneNet0_0();
 private:
  int32_T numLayers;
  MWTensorBase *inputTensors;
  MWTensorBase *outputTensors;
  MWCNNLayer *layers[18];
  MWOnednnTarget::MWTargetNetworkImpl *targetImpl;
  void allocate();
  void postsetup();
  void deallocate();
};
```

Similarly, the file `yolov2ResNet50VehicleExample0_0.h` contains the C++ class, which represents the pretrained YOLO v2 detection network.

```
hfile = fullfile(currentDir, 'laneAndVehicleDetection_grt_rtw',...
    'yolov2ResNet50VehicleExample0_0.h');
rtwdemodbtype(hfile,'#ifndef RTW_HEADER_yolov2ResNet50VehicleExample0_0_h_',...
    '#endif')
```

```
#ifndef RTW_HEADER_yolov2ResNet50VehicleExample0_0_h_
#define RTW_HEADER_yolov2ResNet50VehicleExample0_0_h_
#include "MWOnednnTargetNetworkImpl.hpp"
#include "rtwtypes.h"
#include "MWTensorBase.hpp"
#include "MWTensor.hpp"
#include "MWCNNLayer.hpp"
#include "MWInputLayer.hpp"
#include "MWElementwiseAffineLayer.hpp"
#include "MWFusedConvActivationLayer.hpp"
#include "MWNormLayer.hpp"
#include "MWMaxPoolingLayer.hpp"
#include "MWFCLayer.hpp"
#include "MWReLULayer.hpp"
#include "MWOutputLayer.hpp"
#include "MWConvLayer.hpp"
#include "MWYoloExtractionLayer.hpp"
#include "MWSigmoidLayer.hpp"
#include "MWExponentialLayer.hpp"
#include "MWYoloSoftmaxLayer.hpp"
#include "MWConcatenationLayer.hpp"
#include "MWActivationFunctionType.hpp"
```

```
#include "MWRNNParameterTypes.hpp"
#include "MWTargetTypes.hpp"
#include "shared_layers_export_macros.hpp"
#include "MWOnednnUtils.hpp"
#include "MWOnednnCustomLayerBase.hpp"
#include "MWOnednnCommonHeaders.hpp"

class yolov2ResNet50VehicleExample0_0
{
 public:
  boolean_T isInitialized;
  boolean_T matlabCodegenIsDeleted;
  yolov2ResNet50VehicleExample0_0();
  void setSize();
  void resetState();
  void setup();
  void predict();
  void activations(int32_T layerIdx);
  void cleanup();
  real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
  int32_T getLayerOutputSize(int32_T layerIndex, int32_T portIndex);
  real32_T *getInputDataPointer(int32_T index);
  real32_T *getInputDataPointer();
  real32_T *getOutputDataPointer(int32_T index);
  real32_T *getOutputDataPointer();
  int32_T getBatchSize();
  ~yolov2ResNet50VehicleExample0_0();
 private:
  int32_T numLayers;
  MWTensorBase *inputTensors;
  MWTensorBase *outputTensors;
  MWCNNLayer *layers[57];
  MWOnednnTarget::MWTargetNetworkImpl *targetImpl;
  void allocate();
  void postsetup();
  void deallocate();
};
```

Note: If the **System target file** parameter is set to `grt.tlc`, you must select the **Generate code only** model configuration parameter. If you set the **System target file** to `ert.tlc`, you can clear the **Generate code only** parameter, but to create an executable file, you must generate an example main program.

Cleanup

Close the Simulink model.

```
save_system('laneAndVehicleDetection');
close_system('laneAndVehicleDetection');
```

Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models

This example shows how to generate a generic C/C++ executable that does not depend on any third-party deep learning libraries from a deep learning Simulink® model. The example predicts the remaining useful life (RUL) of a turbofan engine, measured in cycles at each step of an input time series that represents data from various sensors in an engine.

For a video demonstration of this example, see Generate Generic C/C++ Code for Deep Learning Networks in Simulink.

This example is based on the *Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning* example from MATLAB® Coder™. For more information, see “Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning”.

Prerequisites

For PIL verification, you will need:

- STMicroelectronics STM32F407G-Discovery, STM32F746G-Discovery, or STM32F769I-Discovery board
- USB type A to Mini-B cable
- USB TTL-232 cable - TTL-232R 3.3V (serial communication for STM32F4-Discovery board)
- Embedded Coder® Support Package for STMicroelectronics Discovery Boards. To install this support package, use the **Add-Ons** menu in the **Environment** section of the MATLAB **Home** tab.

Pretrained RUL Network

This example uses a pretrained LSTM network to predict the remaining useful life of an engine, measured in cycles. The LSTM network consists of an LSTM layer with 200 hidden units, followed by a fully connected layer of size 50 and a dropout layer with dropout probability 0.5.The network was trained using the *Turbofan Engine Degradation Simulation Data Set* as described in [1]. The training data contains simulated time series data for 100 engines. Each sequence varies in length and corresponds to a full run to failure (RTF) instance. The test data contains 100 partial sequences and corresponding values of the remaining useful life at the end of each sequence. For more information on training the network, see the example “Sequence-to-Sequence Regression Using Deep Learning” (Deep Learning Toolbox).

```
net = coder.loadDeepLearningNetwork('rulNetwork.mat')
```

```
net = 
  SeriesNetwork with properties:

         Layers: [6×1 nnet.cnn.layer.Layer]
     InputNames: {'sequenceinput'}
    OutputNames: {'regressionoutput'}
```

For information on the networks, layers, and classes supported for code generation, see “Networks and Layers Supported for Code Generation”.

Download and Prepare Test Data

This section summarizes the steps to download and prepare the test data that this example uses. For more information on the Turbofan Engine Degradation Simulation data set and the preprocessing steps, see the example “Sequence-to-Sequence Regression Using Deep Learning” (Deep Learning Toolbox).

Download Data Set

Create a directory to store the Turbofan Engine Degradation Simulation data set.

```
dataFolder = fullfile(tempdir,"turbofan");
if ~exist(dataFolder,'dir')
    mkdir(dataFolder);
end
```

Download and extract the Turbofan Engine Degradation Simulation data set.

```
filename = matlab.internal.examples.downloadSupportFile("nnet","data/TurbofanEngineDegradationSi
unzip(filename,dataFolder)
```

Calculate Mean and Standard Deviation of Training Data

In the following step, you normalize the test predictors using the mean and standard deviation of the training data. So, you must first use the training data to calculate these normalization parameters.

Load the training data, each column is one observation, each row is one feature. Remove the features that have constant values.

```
filenamePredictors = fullfile(dataFolder,"train_FD001.txt");
[XTrain] = processTurboFanDataTrain(filenamePredictors);

m = min([XTrain{:}],[],2);
M = max([XTrain{:}],[],2);
idxConstant = M == m;

for i = 1:numel(XTrain)
    XTrain{i}(idxConstant,:) = [];
end
```

Calculate the mean and standard deviation over all observations.

```
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);
```

Prepare Test Data

Prepare the test data using the function `processTurboFanDataTest` attached to this example. The function `processTurboFanDataTest` extracts the data from `filenamePredictors` and `filenameResponses` and returns the cell arrays `XValidate` and `YValidate`, which contain the test predictor and response sequences, respectively.

```
filenamePredictors = fullfile(dataFolder,"test_FD001.txt");
filenameResponses = fullfile(dataFolder,"RUL_FD001.txt");
[XValidate,YValidate] = processTurboFanDataTest(filenamePredictors,filenameResponses);
```

Remove features with constant values using `idxConstant` calculated from the training data. Normalize the test predictors using the parameters `mu` and `sig` calculated from the training data.

Clip the test responses at the threshold 150. This same clipping threshold was used on the training data while training the network.

```
thr = 150;
for i = 1:numel(XValidate)
    XValidate{i}(idxConstant,:) = [];
    XValidate{i} = (XValidate{i} -  mu) ./ sig;
    YValidate{i}(YValidate{i} > thr) = thr;
end
```

To make the input validation data compatible with Simulink code generation, the sequence lengths for each of the independent 100 observations are zero-padded to create uniformly-sized, 17-by-303 input arrays.

```
sequenceLengths = cellfun(@length,XValidate,'UniformOutput',true);
maxSequenceLen = max(sequenceLengths);
padFcn = @(x) [x,zeros(size(x,1),maxSequenceLen-size(x,2))];
XValidatePad = cellfun(padFcn,XValidate,'UniformOutput',false);
```

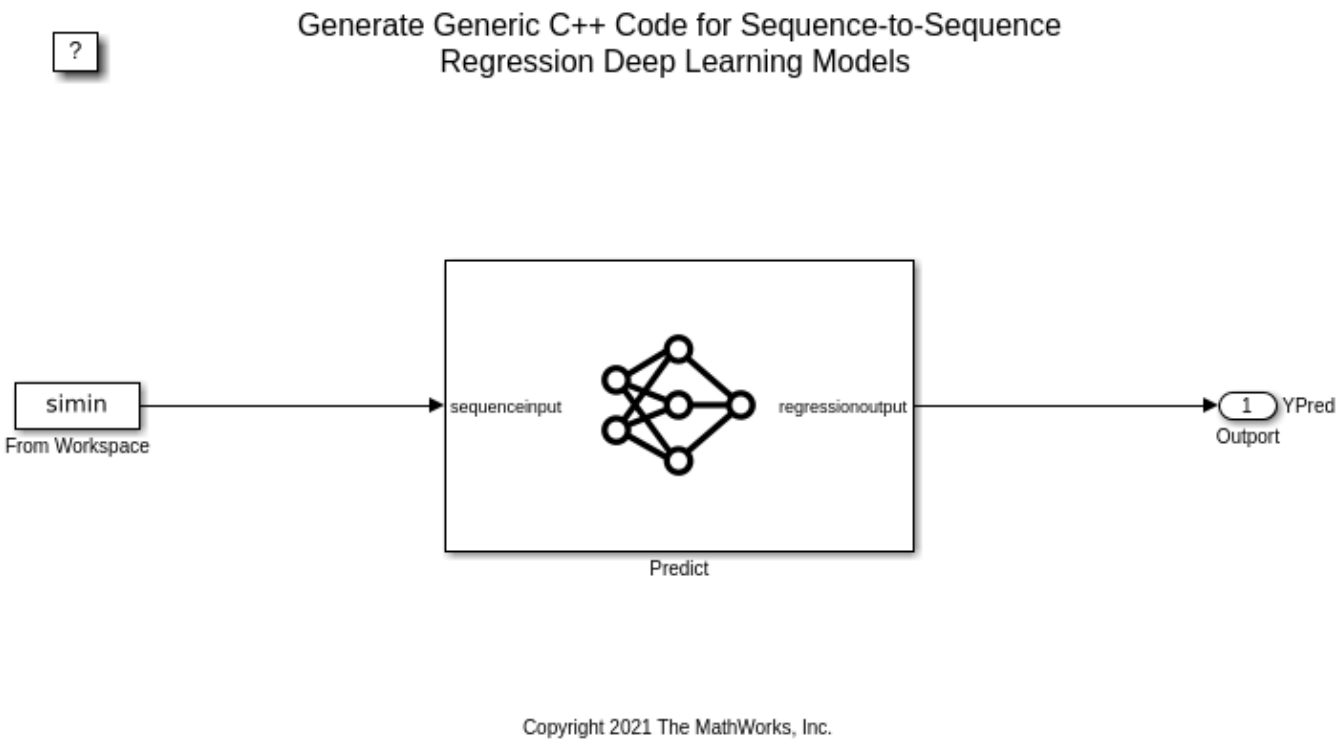
The padded values are then converted to a 17-by-303-100 numeric array. To import this data into the Simulink model, specify a structure variable containing the data values and an empty time vector. During simulation, the input for the first time step is read from the first 17-by-303 element of the array. The value for the second time step is read from the second element, and so on, for a total of 100 steps.

```
simin.time = [];
simin.signals.values = cell2mat(reshape(XValidatePad,1,1,[]));
simin.signals.dimensions = size(XValidatePad{1});
```

Simulink Model for Prediction

The Simulink model for predicts the remaining useful life of a turbofan engine is shown. The model uses the `Predict` block from the `Deep Neural Networks` library that imports the trained network from a `rulNetwork` MAT-file. Additionally, the `Mini-batch size` parameter of the block is set to 1.

```
model = 'rulPredict';
open_system(model)
```



Run the Simulation

To validate the Simulink model, run the simulation.

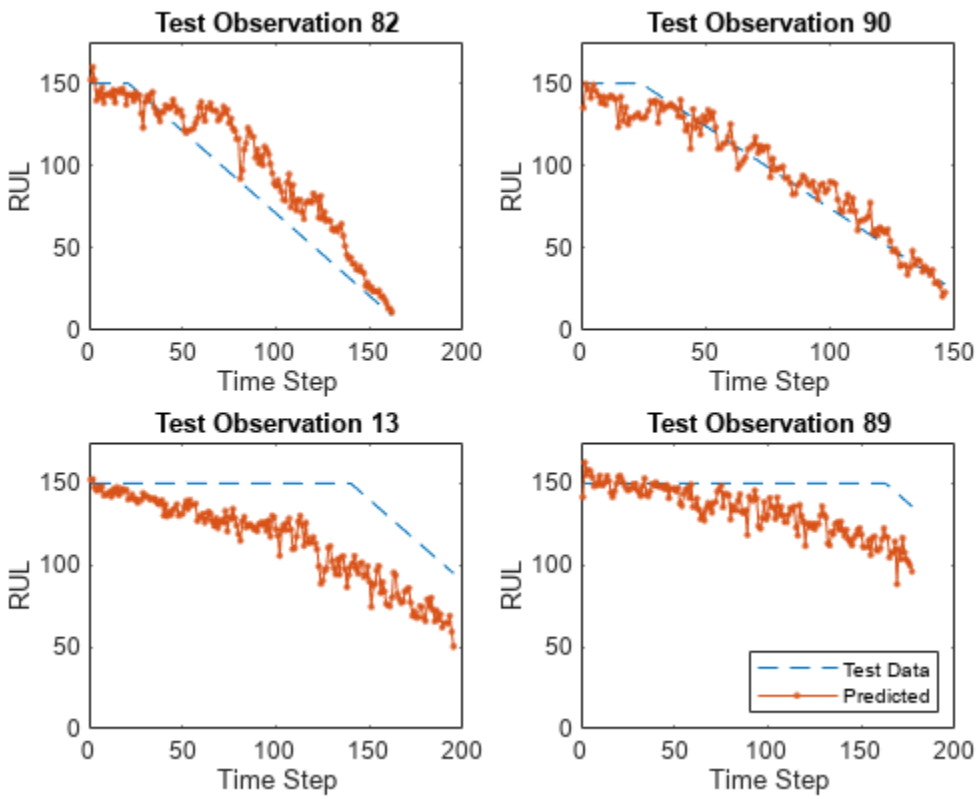
```
set_param(model,'SimulationMode','Normal');
sim(model);
```

The output `YPred` of the Simulink model contains the predicted remaining useful life values from the network. This output is first trimmed to remove the results from the zero-padded values and then converted to a cell array.

```
YPred_cell = squeeze(mat2cell(YPred,1,maxSequenceLen,ones(1,100)));

for t = 1:length(sequenceLengths)
    YPred_cell{t}(:,sequenceLengths(t) + 1:end) = [];
end
```

Plot the predicted remaining useful life (RUL) values for four randomly-selected observations and compare them to the validation data.

```
observationIdx = randperm(100,4);
rulExamplePlots(observationIdx,YValidate,YPred_cell);
```



Configure Model for Code Generation

To generate C/C++ code that does not depend on any third-party deep learning libraries and build an executable, select the generic real-time (ERT) target from Embedded Coder™.

```
cs = getActiveConfigSet(model);
switchTarget(cs,'ert.tlc',[]);
```

Configure code generation specific parameters. Setting MAT-file logging enables the generated code to redirect simulation data to a MAT-file.

```
set_param(cs,'TargetLang','C');
set_param(cs,'Toolchain','Automatically locate an installed toolchain');
set_param(cs,'ObjectivePriorities','Execution efficiency');
set_param(cs,'DLTargetLibrary','None');
set_param(cs,'GenerateReport','on');
set_param(cs,'CodeInterfacePackaging','Nonreusable function');
set_param(cs,'MatFileLogging', 'on');
```

Generate and Build the Simulink Model

Generate and build the Simulink model by using the `slbuild` command. The code generator places the files in a *build folder,* a subfolder named rulPredict_`ert_rtw` under your current working folder.

```
evalc("slbuild('rulPredict')");
```

Run the Generated Executable

Run the generated executable. After successful execution, the `rulPredict` executable creates a `rulPredict` MAT-file in the current working folder.

```
if ispc
    status = system('rulPredict.exe');
else
    status = system('./rulPredict');
end
```

```
** created rulPredict.mat **
```

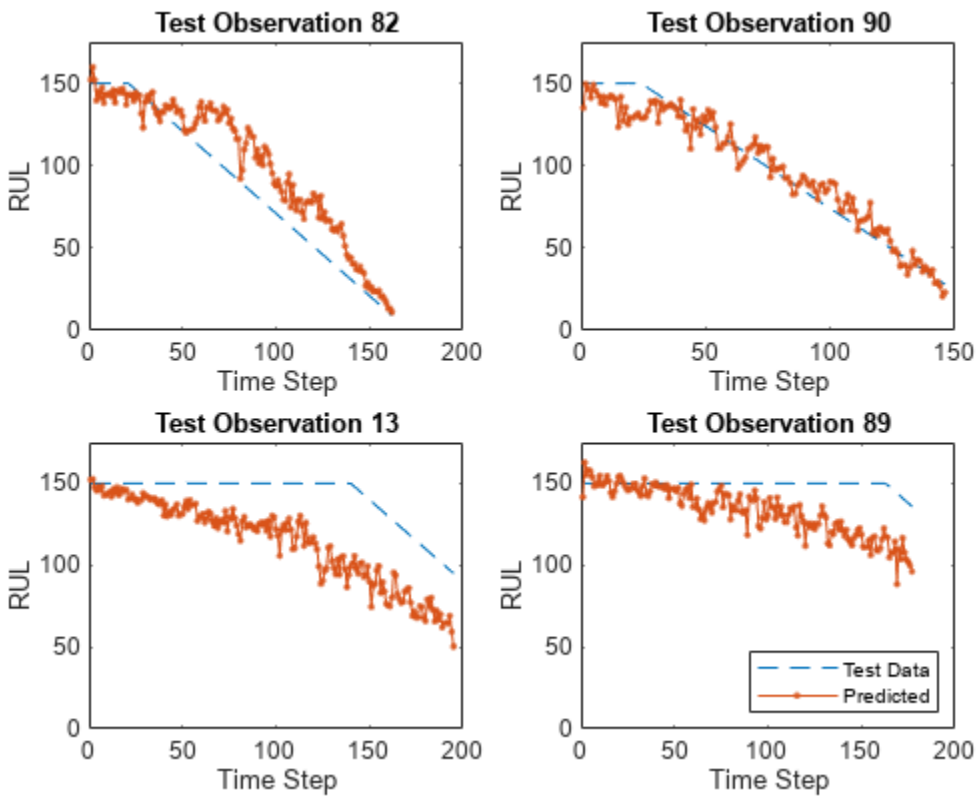
```
load('rulPredict.mat')
```

The output `rt_YPred` contains the predicted remaining useful life values from the network. This output is first trimmed to remove the results from the zero-padded values and then converted to a cell array.

```
rt_YPred_cell = squeeze(mat2cell(rt_YPred,1,maxSequenceLen,ones(1,100)));

for t = 1:length(sequenceLengths)
    rt_YPred_cell{t}(:,sequenceLengths(t) + 1:end) = [];
end
```

Plot the predicted remaining useful life (RUL) values for four randomly-selected observations and compare them to the validation data.

```
rulExamplePlots(observationIdx,YValidate,rt_YPred_cell);
```



Code Verification with Processor-in-the-Loop (PIL) Simulation

To ensure that the behavior of the deployment code matches the design, you can configure the Simulink model to run a PIL simulation on embedded boards. In a PIL simulation, the generated code runs on embedded boards such as the STMicroelectronics® Discovery. The results of the PIL simulation are transferred to Simulink to verify numerical equivalence of the simulation and the code generation results.

This example shows PIL verification on the STMicroelectronics Discovery board by using the *Embedded Coder® Support Package for STMicroelectronics Discovery Boards*. This example is pre-configured to run on the STM32F746G-Discovery board. You can configure this model to use other supported embedded boards by selecting them as the "Hardware board" on the Hardware Implementation pane of the Model Configuration Parameters.

Prepare Test Data

For PIL simulation, this example sorts the observations by sequence length and picks the first 10. These observations are zero-padded to create uniformly-sized, 17-by-54 input arrays. The padded values are then converted to a 17-by-54-10 numeric array. To import this data into the Simulink model, specify a structure variable containing the data values and an empty time vector. During simulation, the input for the first time step is read from the first 17-by-54 element of the array. The value for the second time step is read from the second element, and so on, for a total of 10 steps.

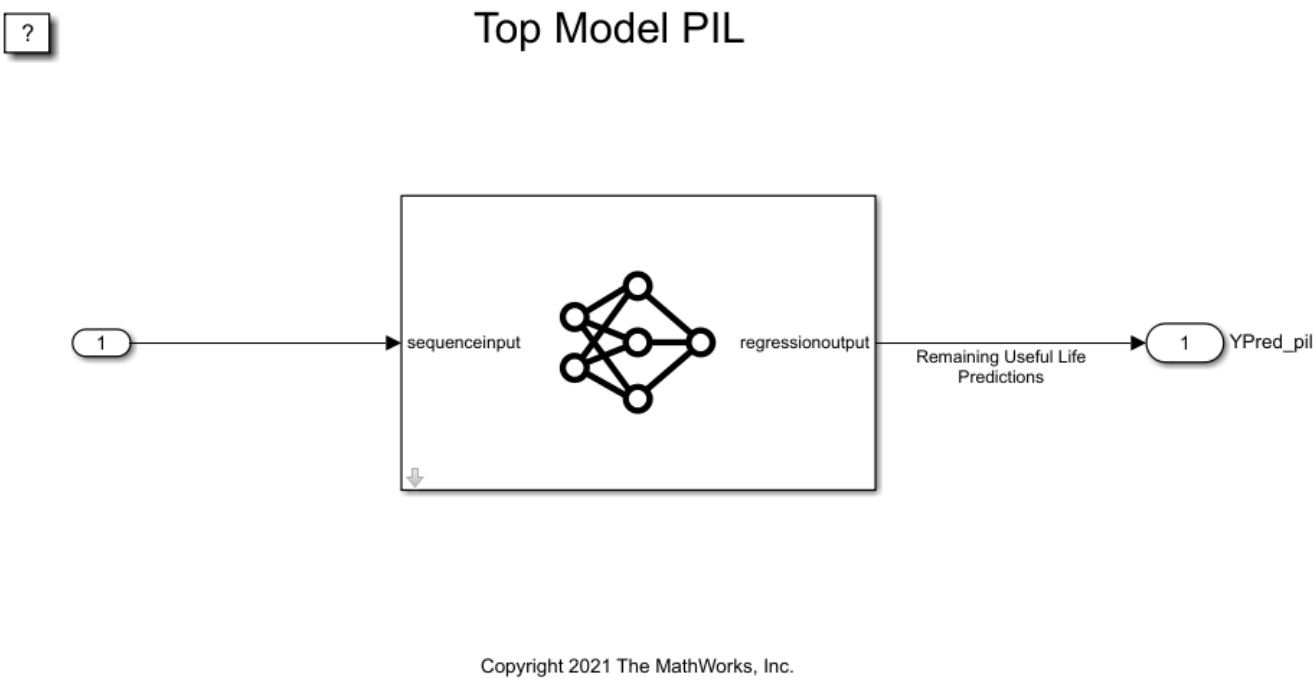
```
[~,idx] = sort(cellfun(@length,XValidate));
XValidatePIL = XValidate(idx(1:10));
YValidatePIL = YValidate(idx(1:10));
```

```
sequenceLengths = cellfun(@length,XValidatePIL,'UniformOutput',true);
maxSequenceLen = max(sequenceLengths);
padFcn = @(x) [x,zeros(size(x,1),maxSequenceLen-size(x,2))];
XValidatePILPad = cellfun(padFcn,XValidatePIL,'UniformOutput',false);
simin.time = [];
simin.signals.values = cell2mat(reshape(XValidatePILPad,1,1,[]));
simin.signals.dimensions = size(XValidatePILPad{1});
```

Top Model PIL

The `rulPredict_pil` is a modified version of the `rulPredict` model for PIL verification. To load the test vectors, the `rulPredict_pil` model replaces the `From Workspace` block with an `Inport` block. The `Inport` block is configured to accept a 17-by-54 array of double data type.

```
pilModel = 'rulPredict_pil';
open_system(pilModel);
```



Configure the `rulPredict_pil` model for the **STM32F467G-Discovery** target.

```
cs = getActiveConfigSet(pilModel);
set_param(cs,'HardwareBoard','STM32F746G-Discovery');
set_param(cs,'Toolchain','GNU Tools for ARM Embedded Processors');
```

The STM32F4-Discovery board supports two different communication interfaces for PIL: ST-LINK and serial. The ST-LINK communication interface does not require any additional cables or hardware besides a USB type A to Mini-B cable used to connect the STM32F4-Discovery board to the host computer. The serial communication interface requires a USB TTL-232 cable. Running a PIL simulation using the serial communication interface is much faster than the running a PIL simulation using ST-LINK. This example is configured to use the serial interface.

Additionally, you must set the COM port parameter in Configuration Parameters > Hardware Implementation > Target Hardware Resources > PIL to match the port number of the serial interface on your Windows computer.

Run PIL Simulation

To compare the PIL results with simulation, run the `rulPredict_pil` model in normal mode and then in PIL mode.

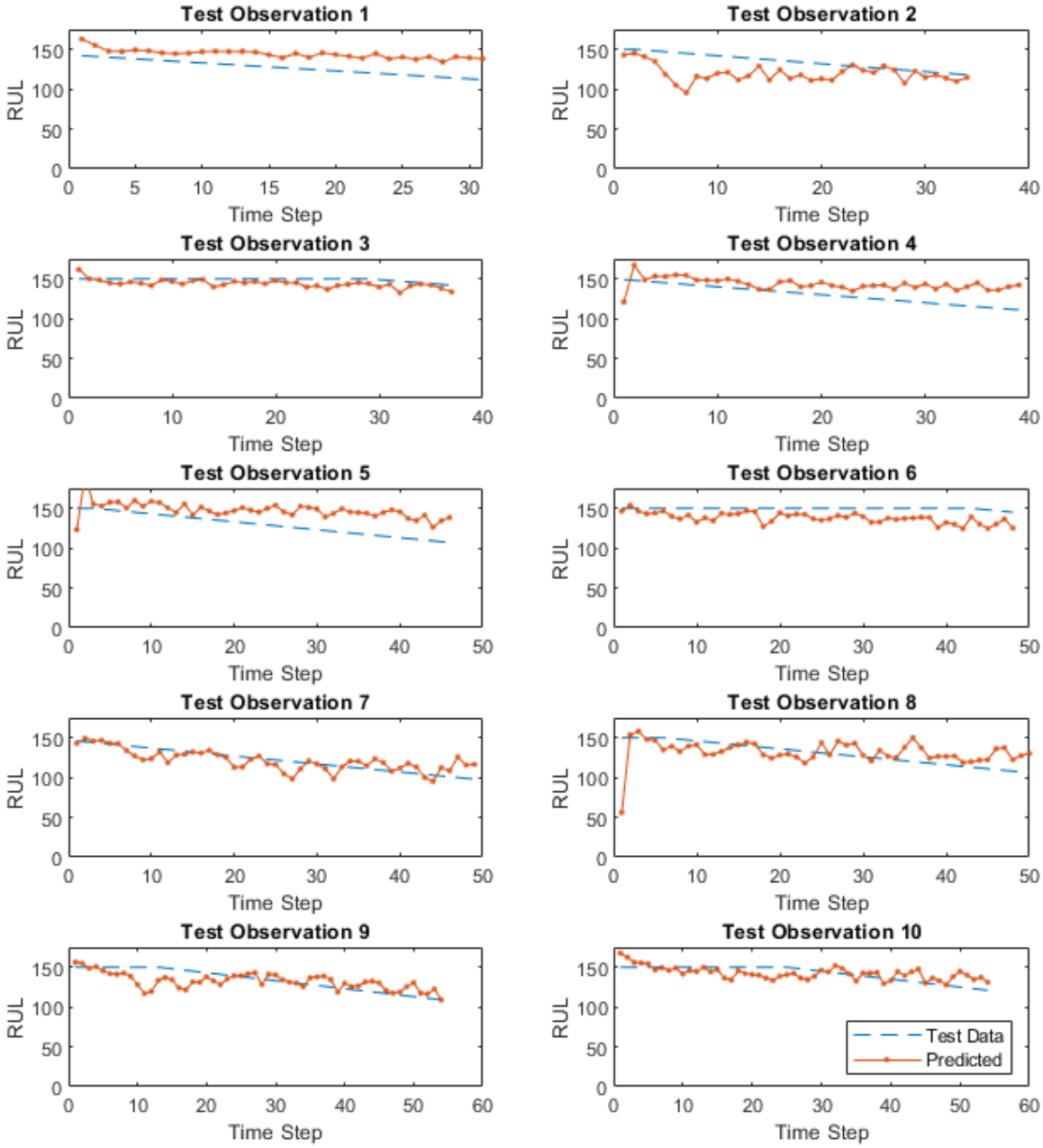
```
set_param(cs,'LoadExternalInput','on');
set_param(cs, 'ExternalInput','simin');
set_param(pilModel,'SimulationMode','Normal');

sim(pilModel);
YPred_ref = YPred_pil;

set_param(pilModel,'SimulationMode','Processor-in-the-Loop (PIL)')
sim(pilModel);
```

Plot Results

The output `YPred_pil` and `YPred_ref` contain the predicted remaining useful life values from the network. This output is first trimmed to remove the results from the zero-padded values and then converted to a cell array.

Plot the predicted remaining useful life (RUL) values for PIL and compare them to the validation data.

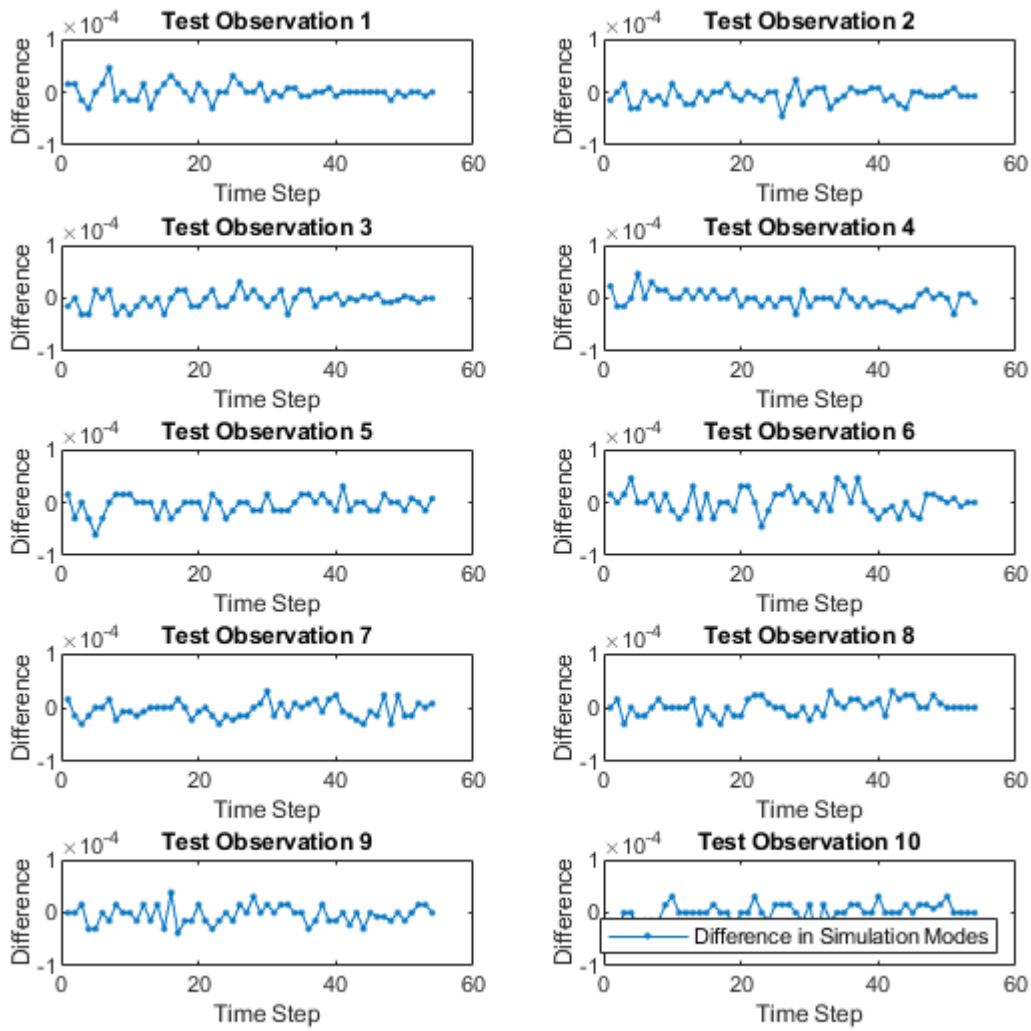
```
YPred_ref_cell = squeeze(mat2cell(YPred_ref,1,maxSequenceLen,ones(1,10)));
YPred_pil_cell = squeeze(mat2cell(YPred_pil,1,maxSequenceLen,ones(1,10)));

for t = 1:length(sequenceLengths)
    YPred_ref_cell{t}(:,sequenceLengths(t) + 1:end) = [];
    YPred_pil_cell{t}(:,sequenceLengths(t) + 1:end) = [];
end

rulExamplePlots([1:10],YValidatePIL,YPred_pil_cell);
```



Compute the error from the PIL simulation and plot the results.

```
YPred_error = YPred_ref-YPred_pil;

figure('Name', 'PIL Error')
```

```
for i = 1:10
    subplot(5,2,i)

    plot(YPred_error(:,:,i),'.-')
    maxerror = string(max(YPred_error(:,:,i)));
    ylim([-1e-4 1e-4])
    title("Test Observation " + i)
    xlabel("Time Step")
    ylabel("Difference")
end
legend(["Difference in Simulation Modes"],'Location','southeast')
```



Supporting Functions

```
function rulExamplePlots(observationIdx,YTest,YPred)
N = numel(observationIdx);
```

```
figure
for i = 1:N
    subplot(N/2,2,i)

    plot(YTest{observationIdx(i)},'--')
    hold on
    plot(YPred{observationIdx(i)},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + observationIdx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southeast')

end
```

References

1. Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pp. 1-9. IEEE, 2008.

Generate Code for Deep Learning Networks Using MATLAB Function Block

You can generate optimized code for Simulink models containing a variety of trained deep learning networks. You can implement the deep learning functionality in Simulink by using MATLAB Function blocks or by using blocks from the **Deep Neural Networks** library. When implementing deep learning functionality with MATLAB Function blocks, use the `coder.loadDeepLearningNetwork` function to load a trained deep learning network and use the object functions of the network object to obtain the desired responses. You can configure the code generator to take advantage of the Intel Math Kernel library for Deep Neural Networks (MKL-DNN). You can also generate generic C or C++ code that does not depend on third-party libraries. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in a network object.

Classify Images by Using GoogLeNet

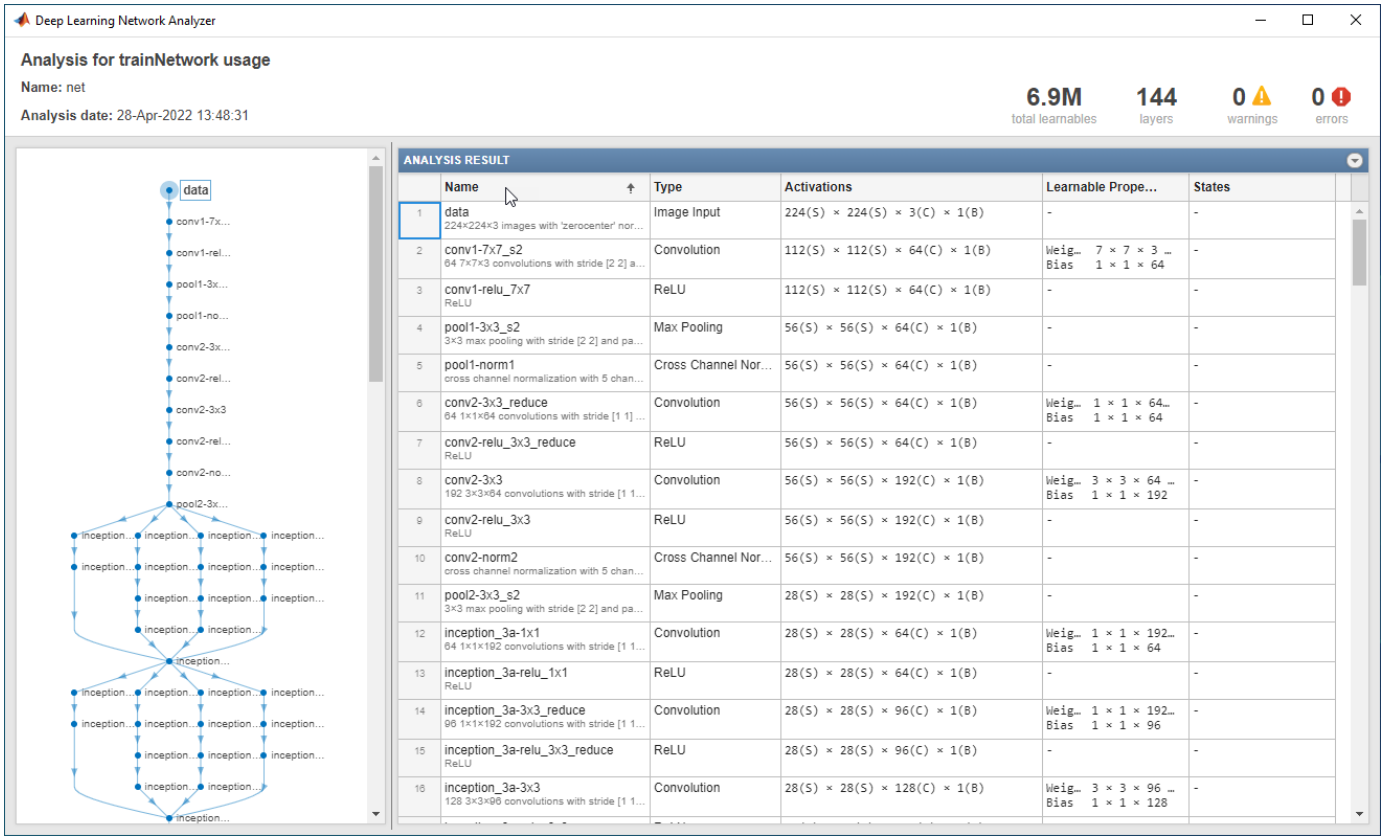
GoogLeNet is a convolutional neural network that is 22 layers deep. The network has been trained on over a million images and can classify an image into 1000 object categories (such as a keyboard, coffee mug, pencil, and animals).The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories. This example shows you how to perform simulation and generate C++ code for the pretrained GoogLeNet deep convolutional neural network and classify an image.

1. Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. This function requires the Deep Learning Toolbox Model for GoogLeNet Network support package. If this support package is not installed, then the function provides a download link.

   ```
   net = googlenet;
   ```

2. The object `net` contains the `DAGNetwork` object. Use the function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

   ```
   analyzeNetwork(net);
   ```



3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

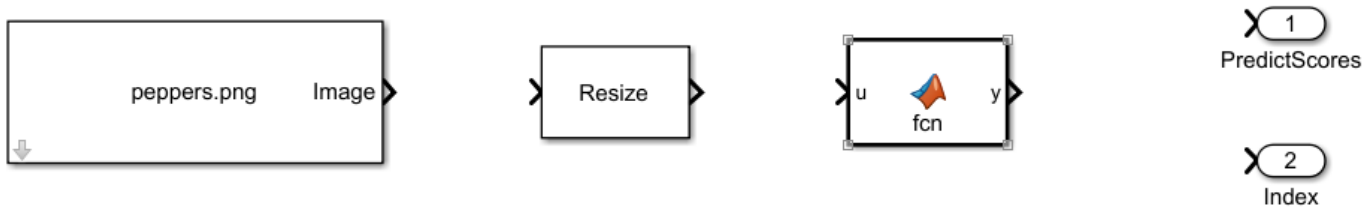
```
speedboat
window screen
isopod
wooden spoon
lipstick
drake
hyena
dumbbell
strawberry
custard apple
```

Create GoogLeNet Model

1 Create a Simulink model and insert a MATLAB Function block from the **User-Defined Functions** library.

2 Add an Image From File block from the **Computer Vision Toolbox** library and set the `File name` parameter to `peppers.png`.

3 Add a Resize block from the **Computer Vision Toolbox** library to the model. Set the **Specify** parameter of the Resize block to `Number of output rows and columns` and enter `[224 224]` as the value for **Number of output rows and columns**. This block resizes the input image to that of the input layer of the network.



4 Double-click the MATLAB Function block. A default function signature appears in the MATLAB Function Block Editor.

5 Define a function called `googlenet_predict`, which implements the prediction entry-point function. The function header declares `in` as an argument to the `googlenet_predict` function, with `scores` and `indxTop` as the return value.

```
function [scores,indxTop] = googlenet_predict(in) %#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('googlenet');
end

% pass in input
predict_scores = predict(mynet,in);
[scores,indx] = sort(predict_scores, 'descend');
indxTop = indx(1:5);
```

A persistent object `mynet` loads the `DAGNetwork` object. At the first call to the entry-point function, the persistent object is constructed and set up. On subsequent calls to the function, the same object is reused to call `predict` on inputs, avoiding reconstructing and reloading the network object.

You can use the `activations` method to network activations for a specific layer. For example, the following line of code returns the network activations for the layer specified in `layerIdx`.

```
out = activations(mynet,in,layerIdx,'OutputAs','Channels');
```

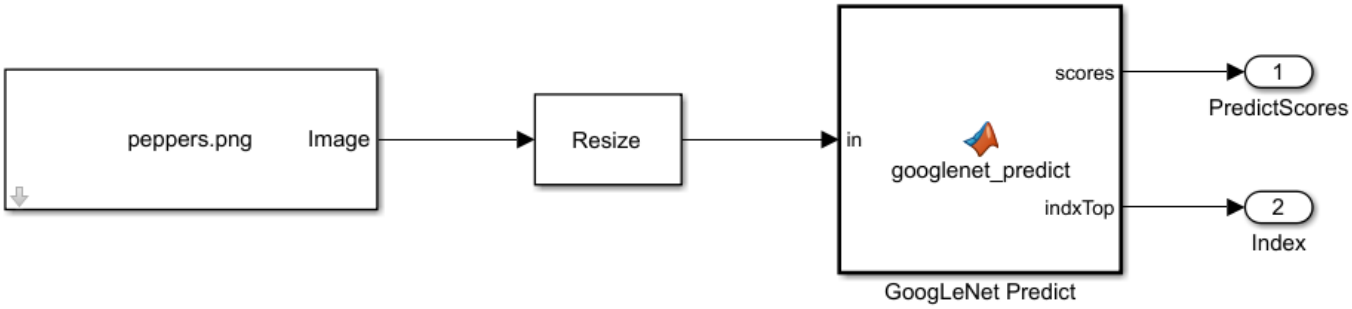
You can use the `classify` method to predict class labels for the image data in `in` using the trained network `mynet`.

```
[out,scores] = classify(mynet,in);
```

For LSTM networks, you can use the `predictAndUpdateState` and `resetState` methods.

6 Open the block parameters of the MATLAB Function block. On the **Code Generation** tab, select `Reusable function` for **Function packaging**.

7 Connect these blocks as shown in the diagram. Save the model as `googlenetModel`.



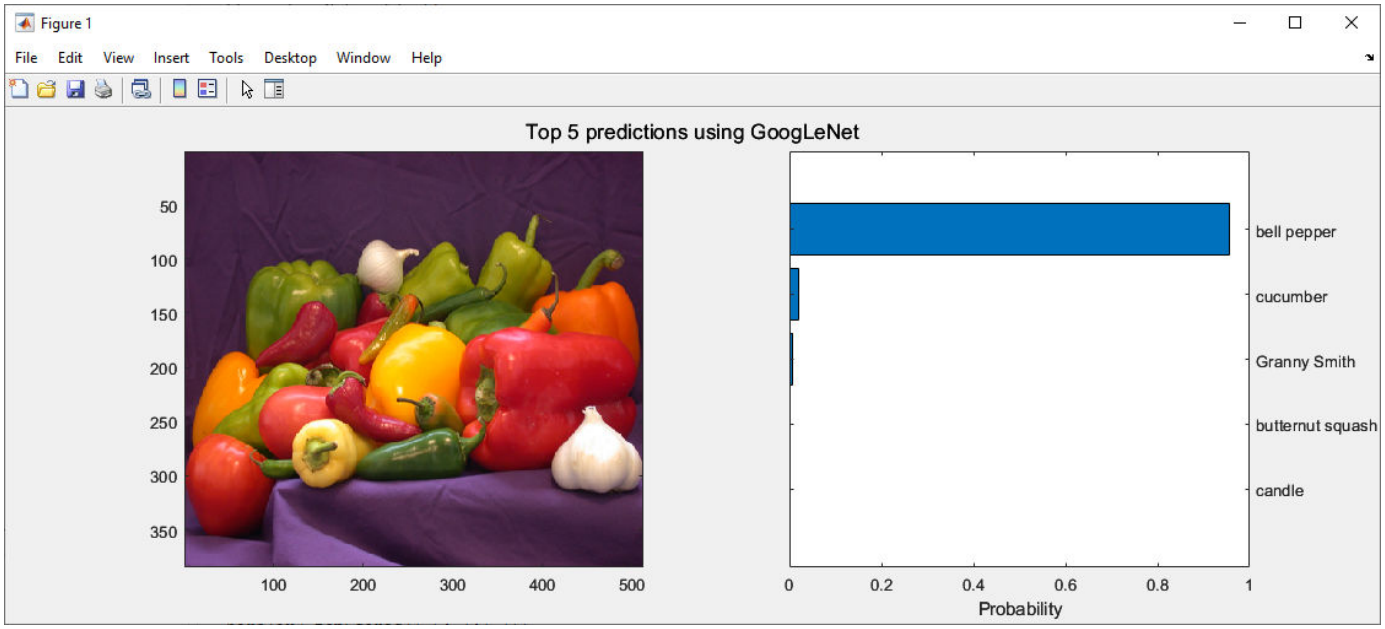
Simulate Model

1. Open the Configuration Parameters dialog box. Select the **Solver** pane. Set the **Type** parameter to `Fixed-step`. This setting maintains a constant (fixed) step size, which is required for code generation.
2. Select the **Simulation Target** pane. Set the **Language** to `C++`.
3. On the **Simulation Target** pane, set the **Target Library** parameter in the **Deep learning** group to `MKL-DNN`.
4. Click **OK** to save and close the Configuration Parameters dialog box.
5. To build and simulate the model, use this command:

   ```
   out = sim('googlenetModel');
   ```

6. Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

   ```
   im = imread('peppers.png');
   classNamesTop = classNames(out.yout{2}.Values.Data(:,:,1))

   h = figure;
   h.Position(3) = 2*h.Position(3);
   ax1 = subplot(1,2,1);
   ax2 = subplot(1,2,2);

   image(ax1,im);
   barh(ax2,out.yout{1}.Values.Data(1,5:-1:1,1))
   xlabel(ax2,'Probability')
   yticklabels(ax2,classNamesTop(5:-1:1))
   ax2.YAxisLocation = 'right';
   sgtitle('Top 5 predictions using GoogLeNet')
   ```



Generate Code

1. Open the **Simulink Coder** or **Embedded Coder** app. The **C Code** tab opens.
2. In the Configuration Parameters dialog box, if you have a Simulink Coder license, set the **System target file** to `grt.tlc`. If you have Embedded Coder, set the **System target file** parameter to `ert.tlc`.
3. Set the **Language** to `C++`.
4. Select **Generate code only**.
5. On the **Code Generation** > **Interface** pane, set the **Target Library** in the **Deep learning** group to `MKL-DNN`.
6. Clear the **MAT-file logging** parameter.
7. Click **OK** to save and close the Configuration Parameters dialog box.
8. On the **C++ Code** app, click **Generate Code**.

See Also

`googlenet` (Deep Learning Toolbox) | Image From File (Computer Vision Toolbox) | MATLAB Function

Related Examples

- "Generate Code for Blocks from Deep Neural Networks Library" on page 83-27
- "Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models"
- "Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection"

Generate Code for Blocks from Deep Neural Networks Library

You can generate optimized code for Simulink models containing a variety of trained deep learning networks. You can implement the deep learning functionality in Simulink by using MATLAB Function blocks or by using blocks from the **Deep Neural Networks** library. For information on the blocks in the **Deep Neural Networks** library, see Blocks.

You can configure the code generator to take advantage of the Intel Math Kernel library for Deep Neural Networks (MKL-DNN). You can also generate generic C or C++ code that does not depend on third-party libraries. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in network object.

Classify Images by Using GoogLeNet

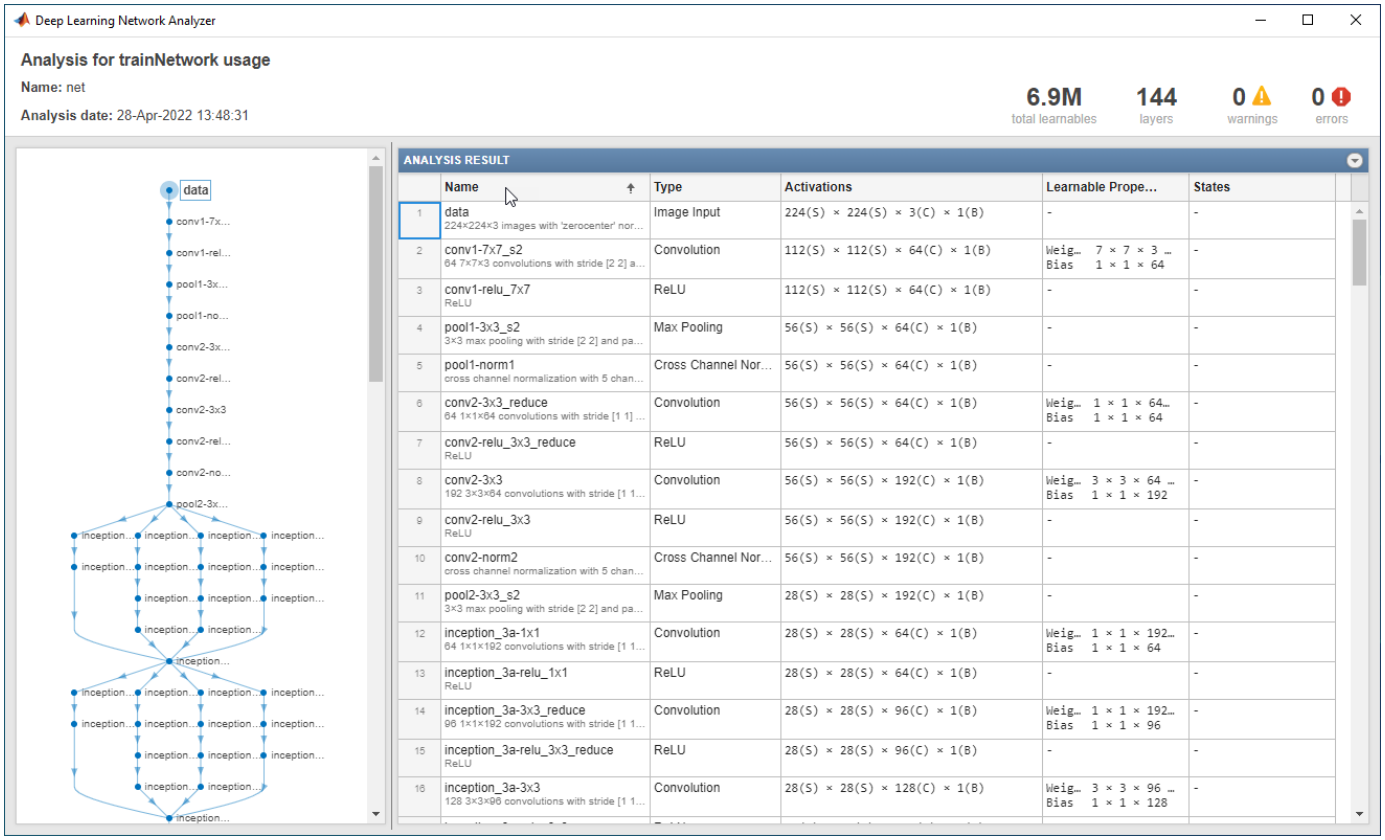
GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as a keyboard, coffee mug, pencil, and animals). The network takes an image as input, and then outputs a label for the object in the image with the probabilities for each of the object categories. This example shows how to perform simulation and generate C++ code for the pretrained `googlenet` deep convolutional neural network and classify an image. The pretrained networks are available as support packages from the Deep Learning Toolbox.

1. Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. This function requires the Deep Learning Toolbox Model for GoogLeNet Network support package. If this support package is not installed, then the function provides a download link.

   ```
   net = googlenet;
   ```

2. The object `net` contains the `DAGNetwork` object. Use the function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

   ```
   analyzeNetwork(net);
   ```



3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
speedboat
window screen
isopod
wooden spoon
lipstick
drake
hyena
dumbbell
strawberry
custard apple
```

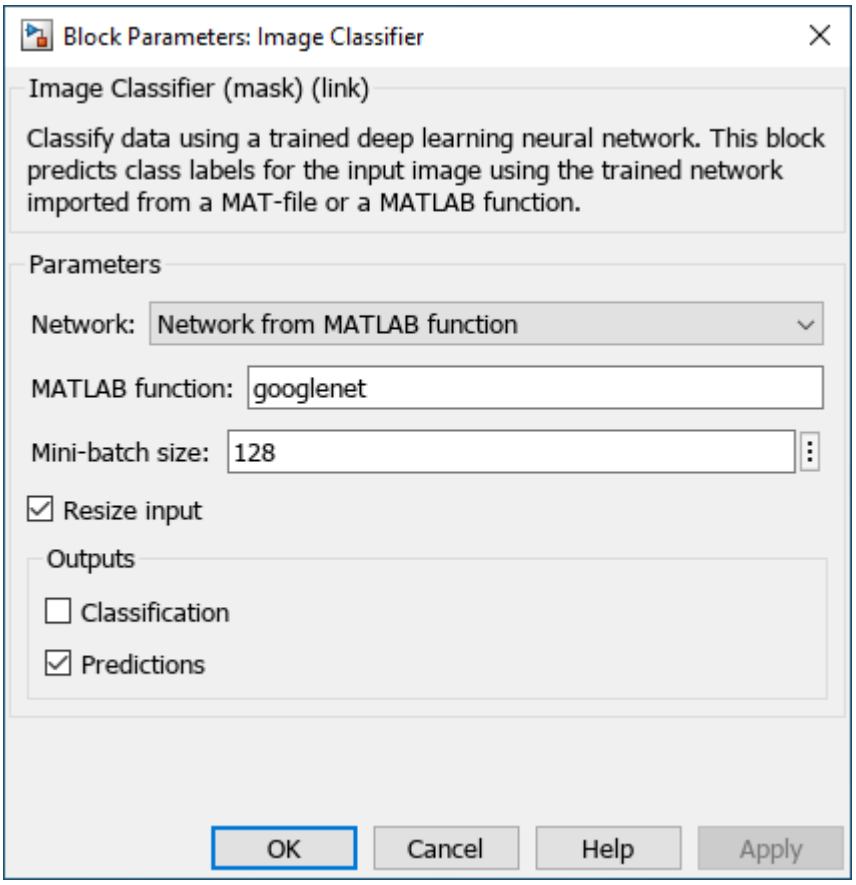
Create GoogLeNet Model

1 Create a Simulink model.

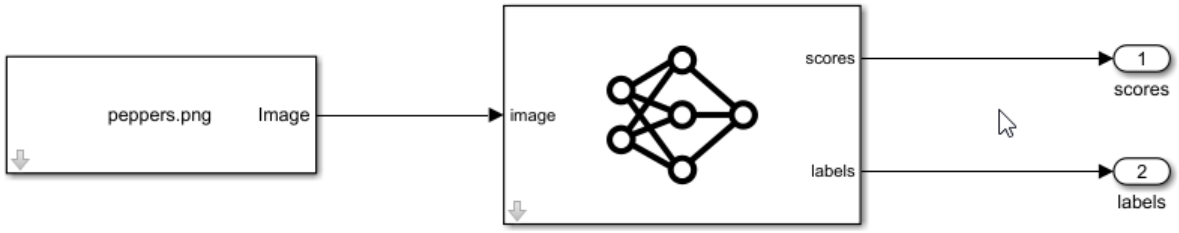
2 Add an Image From File block from the **Computer Vision Toolbox** library and verify that the `File name` parameter to `peppers.png`.

3 Insert an Image Classifier block from the **Deep Learning Toolbox** > **Deep Neural Networks** library. Set these block configuration parameters:

- Set the **Network** parameter to `Network from MATLAB function`.
- Set the **MATLAB function** parameter to `googlenet`.
- Select the **Resize input** parameter.
- Select the **Predictions** parameter.
- Clear the **Classifications** parameter.



4 Add two Outport blocks and connect them to the Image Classifier block. Connect the Image Classifier block to the Image From File block.



5 Save the model as `googlenetModel.slx`.

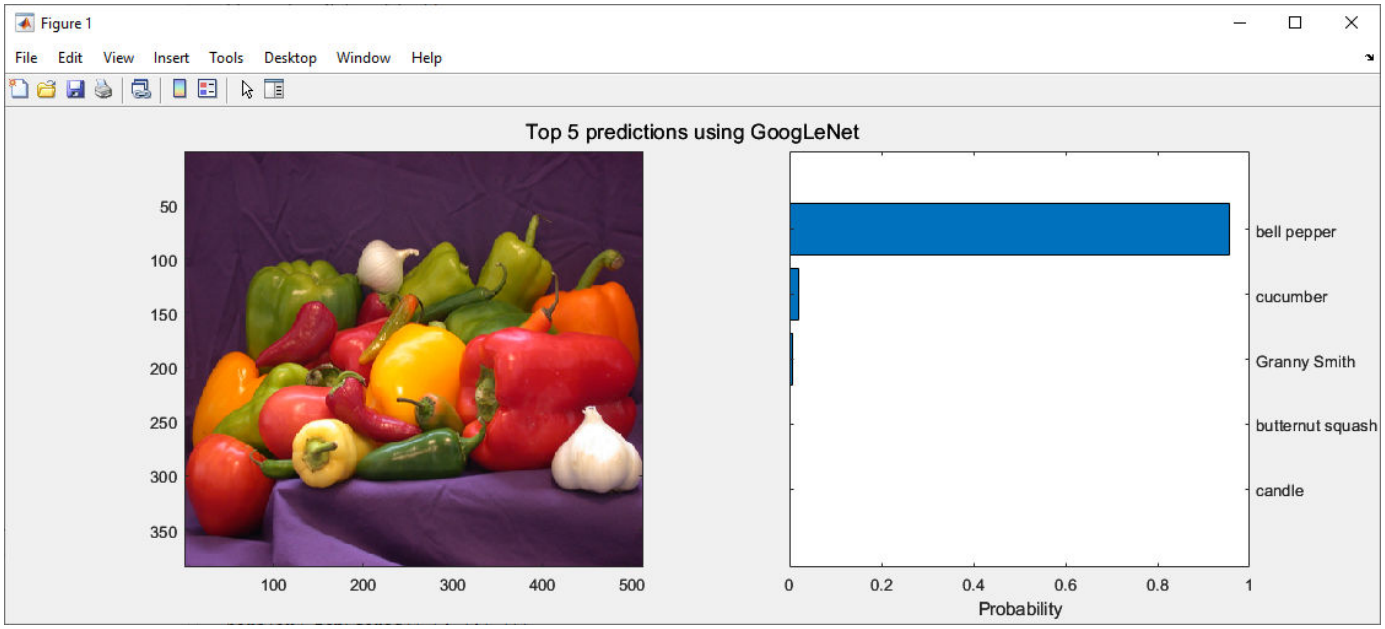
Simulate Model

1. Open the Configuration Parameters dialog box. Select the **Solver** pane. Set the **Type** parameter to `Fixed-step`. This setting maintains a constant (fixed) step size, which is required for code generation.
2. Select the **Simulation Target** pane. Set the **Language** to `C++`.
3. On the **Simulation Target** pane, verify that the **Target Library** parameter in the **Deep learning** group is set to `MKL-DNN`.
4. Click **OK** to save and close the Configuration Parameters dialog box.
5. To build and simulate the model, use this command:

   ```
   out = sim('googlenetModel');
   ```

6. Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

   ```
   im = imread('peppers.png');
   predict_scores = out.yout{1}.Values.Data(:,:,1);
   [scores,indx] = sort(predict_scores,'descend');
   topScores = scores(1:5);
   classNamesTop = classNames(indx(1:5))

   h = figure;
   h.Position(3) = 2*h.Position(3);
   ax1 = subplot(1,2,1);
   ax2 = subplot(1,2,2);

   image(ax1,im);
   barh(ax2,topScores(1,5:-1:1,1))
   xlabel(ax2,'Probability')
   yticklabels(ax2,classNamesTop(5:-1:1))
   ax2.YAxisLocation = 'right';
   sgtitle('Top 5 predictions using GoogLeNet')
   ```



Generate Code

1. Open the **Simulink Coder** or **Embedded Coder** app. The **C Code** tab opens.
2. In the Configuration Parameters dialog box, if you have a Simulink Coder license, set the **System target file** to `grt.tlc`. If you have Embedded Coder, set the **System target file** parameter to `ert.tlc`.
3. Set the **Language** to `C++`.
4. Select **Generate code only**.
5. On the **Code Generation** > **Interface** pane, set the **Target Library** in the **Deep learning** group to `MKL-DNN`.
6. On the **Interface** pane, clear the **MAT-file logging** parameter.
7. If the **System target file** parameter is set to `ert.tlc`, on the **Interface** pane, select the **variable-size signals** parameter.
8. Click **OK** to save and close the Configuration Parameters dialog box.
9. On the **C++ Code** app, click **Build**.

See Also

`googlenet` (Deep Learning Toolbox) | Image From File (Computer Vision Toolbox) | MATLAB Function

Related Examples

- “Generate Code for Deep Learning Networks Using MATLAB Function Block” on page 83-22
- “Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models” on page 83-10
- “Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection” on page 83-2

Workflow for Deep Learning C/C++ Code Generation for Simulink Models

With Simulink Coder, you can generate code from deep learning neural networks you design and implement using the Deep Learning Toolbox. To use Simulink Coder to generate code for deep learning networks, you must also install the MATLAB Coder Interface for Deep Learning. Deep learning uses neural networks to learn useful representations of features directly from data. You can obtain a pretrained neural network or train one yourself using the Deep Learning Toolbox. For more information, see “Train Deep Learning Network to Classify New Images” (Deep Learning Toolbox) and “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

Implement the trained neural network in Simulink by using blocks from the Deep Neural Networks library or by using a MATLAB Function block. When implementing the trained neural network with a MATLAB Function block, use the `coder.loadDeepLearningNetwork` to load a trained deep learning network and use the object functions of the network object to obtain the desired responses. The network must be supported for code generation. See “Networks and Layers Supported for Code Generation”.

You can generate C++ code that targets an embedded platform that uses an Intel processor or an ARM processor. The generated code calls the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) or the ARM Compute Library to apply high performance. The hardware and software requirements depend on the target platform. To apply these libraries, in the Model Configuration Parameters dialog box, set these parameter settings.

Pane	Parameter	Setting
Simulation Target	**Language**	`C++`
Simulation Target	**Target library**	`MKL-DNN`
Code Generation	**Language**	`C++`
Interface	**Target library**	• `MKL-DNN` • `ARM Compute`

For an example that uses the MKL-DNN library, see “Code Generation for Deep Learning Simulink Model That Performs Lane and Vehicle Detection” on page 83-2.

You can also generate generic C or C++ code that does not depend on third-party libraries. To generate generic C or C++ code, set these parameter settings.

Pane	Parameter	Setting
Simulation Target	**Language**	`C` or `C++`
Code Generation	**Language**	`C` or `C++`
Interface	**Target library**	`None`

For an example, see “Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models” on page 83-10.

Deep learning models typically work on large sets of labeled data. Performing inference on these models is computationally intensive, consuming a significant amount of memory. You can use pruning in combination with network quantization to reduce the inference time and memory footprint of the deep learning network, making it easier to deploy to low-power microcontrollers and FPGAs. For more information, see:

- “Quantization of Deep Neural Networks” (Deep Learning Toolbox)
- “Generate int8 Code for Deep Learning Networks”
- “Prune Image Classification Network Using Taylor Scores” (Deep Learning Toolbox)

To perform quantization, you must install the Deep Learning Toolbox Model Quantization Library support package.

See Also

Related Examples

- “Generate Code for Deep Learning Networks Using MATLAB Function Block” on page 83-22
- “Generate Code for Blocks from Deep Neural Networks Library” on page 83-27
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Workflow for Deep Learning Code Generation with MATLAB Coder”

Verification

Simulation and Code Comparison in Simulink Coder

Simulation and Code Comparison

In this section...

"Configure Signal Data for Logging" on page 84-2
"Log Simulation Data" on page 84-3
"Run Executable and Load Data" on page 84-4
"Visualize and Compare Results" on page 84-5
"Compare States for Simulation and Code Generation" on page 84-7

This example shows how to verify the answers computed by code generated from the `slexAircraftExample` model. It shows how to capture and compare two sets of output data. Simulating the model produces one set of output data. Executing the generated code produces a second set of output data.

Note To obtain a valid comparison between model output and the generated code, use the same **Solver selection** and **Step size** for the simulation run and the build process.

Configure Signal Data for Logging

Configure the model for logging and recording signal data.

1. Make sure that `slexAircraftExample` is closed. Clear the base workspace to eliminate the results of previous simulation runs. In the Command Window, type:

   ```
   clear
   ```

   The clear operation clears variables created during previous simulations and all workspace variables, some of which are standard variables that the `slexAircraftExample` model requires.

2. To open the model, in the Command Window, enter:

   ```
   openExample('slexAircraftExample')
   ```

   .

3. In the model window, choose **File > Save As**, navigate to the working folder, and save a copy of the `slexAircraftExample` model as `myAircraftExample`.

4. Set up your model to log signal data for signals: `Stick`, `alpha,rad`, and `q, rad/sec`. For each signal:

   a. Right-click the signal. From the context menu, select **Properties**.

   b. In the Signal Properties dialog box, select **Log signal data**.

   c. In the **Logging name** section, from the drop-down list, select `Custom`.

   d. In the text field, enter the logging name for the corresponding signal.

Signal Name	Logging Name
`Stick`	`Stick_input`
`alpha,rad`	`Alpha`

Signal Name	Logging Name
`q, rad/sec`	`Pitch_rate`

e Click **Apply** and **OK**.

For more information, see “Save Signal Data Using Signal Logging”.

5 In the Configuration Parameters dialog box:

a Set **Type** to `Fixed-step`.

b Set **Format** to `Structure with time`.

c Clear the **States** check box.

d Select the **Signal logging** check box.

e Select the **Record logged workspace data in Simulation Data Inspector** check box.

6 Save the model.

Proceed to “Log Simulation Data” on page 84-3.

Log Simulation Data

Run the simulation, log the signal data, and view the data in the Simulation Data Inspector.

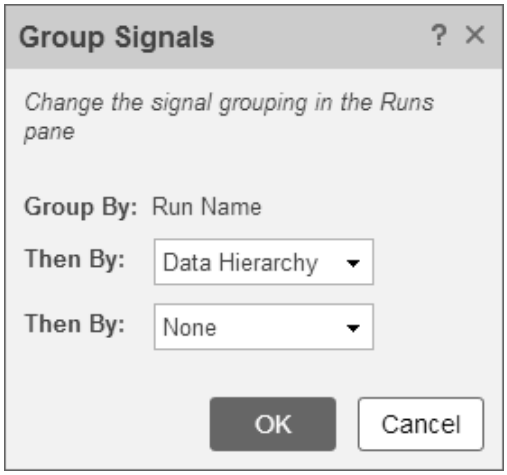
1 Run the model. When the simulation is done, on the Simulink Editor toolbar, the **Simulation Data Inspector** button is highlighted to indicate that new simulation output is available in the Simulation Data Inspector.

2 Click the **Simulation Data Inspector** button to open the Simulation Data Inspector.

3 Group the signals:

a On the **Visualize** tab, click **Group Signals**.

b In the Group Signals dialog box, select `Data Hierarchy` from the **Then By** list.



c Click **OK**.

4 Click the `logsout` expander to view the logged signals.

5 Click the **Format** tab.

6 Click the **Subplots** button and select `3x1` to show three subplots.

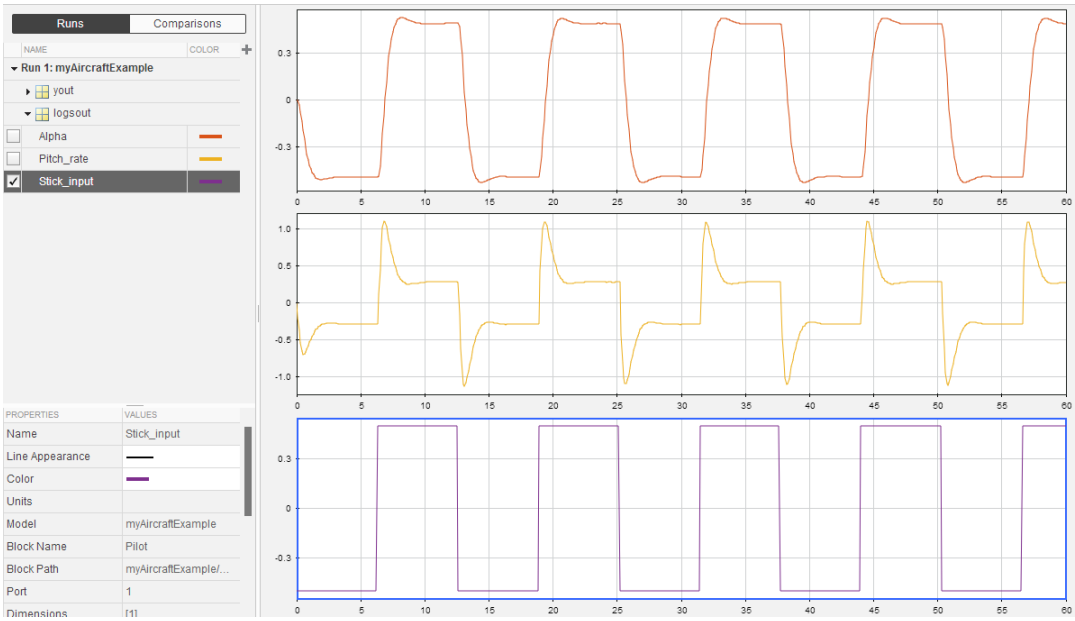
7 For each signal:

   a Click the top subplot. A blue border indicates the plot selection.

   b Select the check box next to the `Alpha` signal name. The signal data appears in the subplot.

   c Plot the `Pitch_rate` signal in the middle subplot.

   d Plot the `Stick_input` signal in the bottom subplot.



Proceed to “Run Executable and Load Data” on page 84-4.

Run Executable and Load Data

You must rebuild and run the `myAircraftExample` executable to obtain a valid data file because you have modified the model.

1 In the Configuration Parameters dialog box, set the **MAT-file variable name modifier** parameter to `rt_`. `rt_` is prefixed to each variable that you selected for logging in the first part of this example.

2 Click **Apply** and **OK**.

3 Save the model.

4 To generate code, on the **C Code** tab, click **Build** button.

5 When the build is finished, run the standalone program from the Command Window.

```
!myAircraftExample
```

The executing program writes the following messages to the Command Window.

```
** starting the model **
** created myAircraftExample.mat **
```

6 Load the data file `myAircraftExample.mat`.

```
load myAircraftExample
```

Tip For UNIX platforms, run the executable program in the Command Window with the syntax `!./executable_name`. If preferred, run the executable program from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs".

Proceed to "Visualize and Compare Results" on page 84-5.

Visualize and Compare Results

When you follow the example sequence that began in "Configure Signal Data for Logging" on page 84-2, you obtain data from a Simulink run of the model and from a run of the program generated from the model.

1 To view the execution output for `alpha,rad`, import the data into the Simulation Data Inspector.

   a On the Simulation Data Inspector **Visualize** tab, click the **Import** button to open the Import dialog.

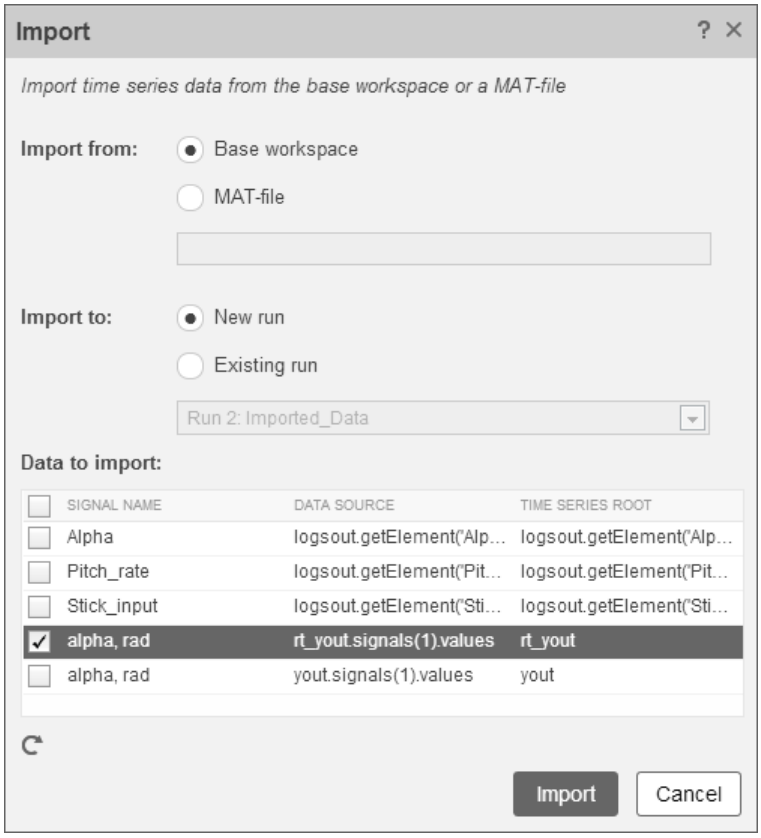
   b Specify **Import from** as **Base workspace**.

   c Specify **Import to** as **New run**.

   d To the left of **Signal Name**, click the check mark to clear the check boxes.

   e Select the check box for the `alpha,rad` data where the **Time Series Root** is `rt_yout`.

   f Click **Import**.



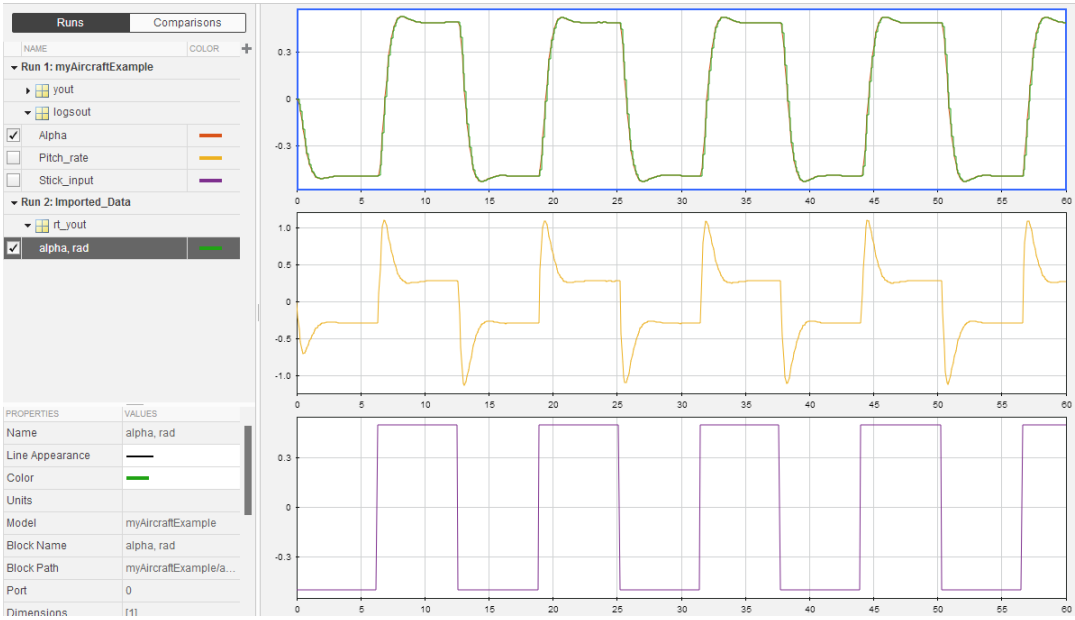
The selected data is now under **Run 2: Imported_Data**.

2 View a plot of the executed data.

   a Click the `rt_yout` expander.

   b Click the top subplot and select the check box next to the `alpha, rad` signal name. The signal data appears in the top subplot.

   The `alpha, rad` signal from Run 1 and Run 2 overlap in the subplot because the signals are equivalent.



It is possible to see a very small difference between simulation and code generation results. A slight difference can be caused by many factors, including:

- Different compiler optimizations
- Statement ordering
- Run-time libraries

For example, a function call such as `sin(2.0)` can return a slightly different value depending on which C library you use. Such variations can also cause differences between your results and these results.

Compare States for Simulation and Code Generation

The order in which Simulink logs states during simulation is different than the order in which Simulink Coder logs states during code generation. If you want to compare states between simulation and code generation, sort the states by block name.

For example, by default, Simulink exports state data to the MATLAB variable, `xout`. Simulink Coder exports state data to the variable `rt_xout`. To sort the state data for these variables, enter the following commands in the MATLAB Command Window:

```
[~,idx1]=sort({xout.signals.blockName});
xout_sorted=[xout.signals(idx1).values];
[~,idx2]=sort({rt_xout.signals.blockName});
rt_xout_sorted=[rt_xout.signals(idx2).values];
```

You can confirm that the logging order is the same between code generation and simulation by entering the following command in the MATLAB Command Window:

```
isequal(xout_sorted, rt_xout_sorted)
```

See Also

Related Examples

- “Log Program Execution Results”

Code Tracing in Embedded Coder

- “Verify Generated Code by Using Code Tracing” on page 85-2
- “Trace Simulink Model Elements in Generated Code” on page 85-8
- “Trace Stateflow Elements in Generated Code” on page 85-12
- “Link Generated Code to Requirements” on page 85-24
- “Reload Existing Traceability Information” on page 85-27
- “Customize Traceability Reports” on page 85-29
- “Trace Generated Code to Blocks” on page 85-32
- “Use Traceability in MATLAB Function Blocks” on page 85-34

Verify Generated Code by Using Code Tracing

Code tracing (traceability) uses hyperlinks to navigate between a line of generated code and its corresponding elements in a model. To find the lines of code and their corresponding elements, you can also right-click an element or elements in a model. This two-way navigation is *bidirectional* traceability.

Using code tracing, you can:

- Verify that generated code is as you expect. You can identify which model elements correspond to a line of code. You can track code from different elements that you have or have not reviewed.
- Verify that generated code meets the design requirements. You can link requirements to the model element and use code tracing to verify that the generated code for a model element meets the assigned requirements.

When you generate code from a Simulink model, traceability information is embedded in the generated code, unless explicitly unspecified. The traceability information includes links for tracing between the generated source code and the model. You can view the generated code by either the:

- “Code View” on page 85-3: View the generated code in the Code view of the Code perspective.
- “Code Generation Report” on page 85-4: View the generated code as an HTML report in the MATLAB web browser.

The generated code includes resources that support code tracing:

- Code element hyperlinks (indicated by underlining when you place your cursor over the code) to trace variables or types in the generated code to their declarations or definitions in the header files.
- Tags in code comments that identify elements in a model from which lines of code are generated.
- Line number hyperlinks that link to the model component from which the line of code was generated.

Traceable Elements

Bidirectional traceability is supported for Simulink blocks and these Stateflow elements:

- States
- Transitions
- State transition tables
- MATLAB functions. Traceability is not supported for external code that you call from a MATLAB function.
- Truth table blocks
- Graphical functions
- Simulink functions

Traceability in one direction is supported for these Stateflow elements:

- Events (code-to-model)

Code-to-model traceability works for explicit events, but not implicit events. Clicking a hyperlink for an explicit event in the generated code highlights that item in the **Contents** pane of the Model Explorer.

- Junctions (model-to-code)

  Model-to-code traceability works for junctions with at least one outgoing transition. Right-clicking such a junction in the Stateflow Editor highlights the line of code that corresponds to the first outgoing transition for that junction.

For more information, see “Trace Stateflow Elements in Generated Code” on page 85-12.

MATLAB Function blocks that you insert directly into a Simulink model are also traceable. For more information, see “Use Traceability in MATLAB Function Blocks” on page 85-34.

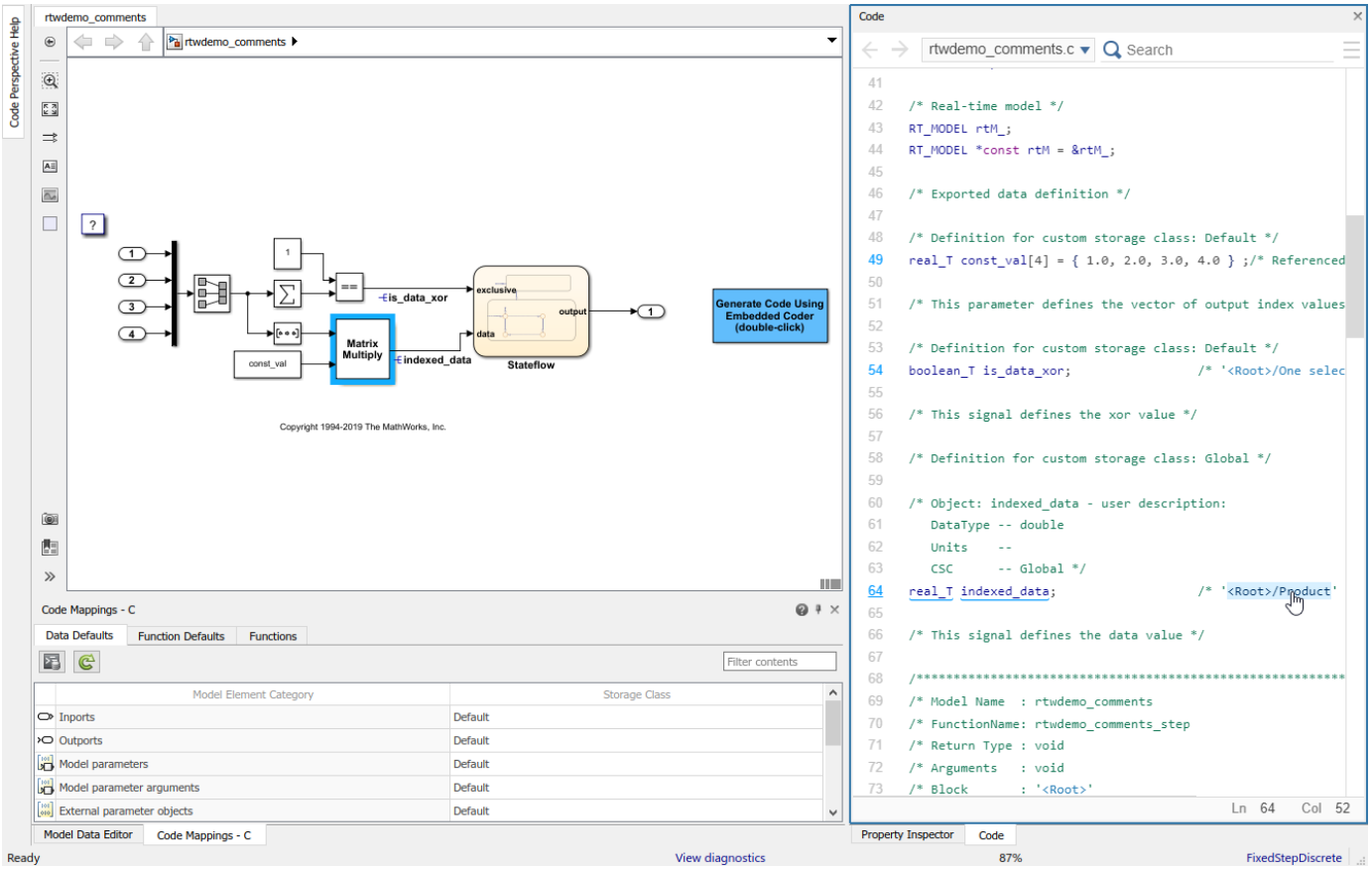
Traceability in Generated Code

- “Code View” on page 85-3
- “Code Generation Report” on page 85-4

Code View

1. Open the model `rtwdemo_comments`.
2. Open the Embedded Coder app. Build the model. On the **C Code** tab, click **Build**.

   The HTML code generation report opens by default. To use the report, see “Code Generation Report” on page 85-4.
3. View the generated code in the Code view in the Code perspective.
4. Switch between the generated source files by using the drop-down list at the top of the Code view.
5. The source code contains traceability information such as hyperlinked comments, line numbers, variables, and operators. Place your cursor over or click a comment or line number hyperlink. The Simulink Editor highlights the corresponding block or blocks in the model.



6 To highlight the generated code for a block in the model, select the block. The generated code for the block is highlighted in the Code view.

7 To open the generated code for a referenced model, open the referenced model and view the generated code in the Code view.

Code Generation Report

1 Open the model `rtwdemo_comments`.

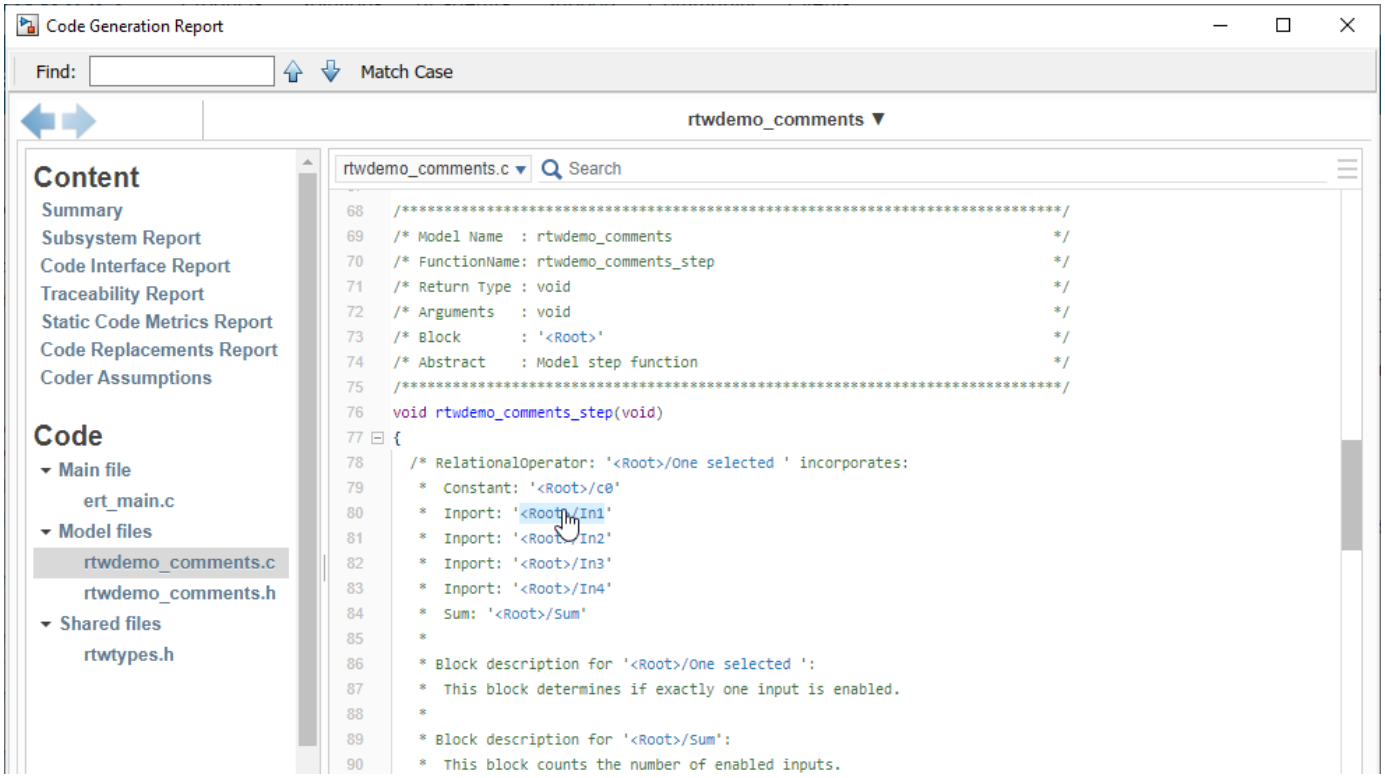
2 Open the Embedded Coder app. On the **C Code** tab, click **Settings** to open the Configuration Parameters dialog box. Select **Create code generation report** if it is not already selected. By default, **Open report automatically**, **Code-to-model**, and **Model-to-code** are selected.

3 If your model contains referenced models and you want to enable traceability for the referenced model's code generation report, repeat the previous step for each referenced model.

4 Build the model. On the **C Code** tab, click **Build**. The build process opens the code generation report in a MATLAB web browser.

5 In the left navigation pane, select a source code file. The source code and line numbers in the right pane contain hyperlinks to blocks in the model.

6 Click a comment or line number hyperlink. The Simulink Editor displays and highlights the corresponding block or blocks in the model.



7. To highlight the generated code for a block in the model, right-click the block and select **C/C++ Code** > **Navigate to C/C++ Code**. The generated code for the block is then highlighted in the HTML code generation report. To highlight the generated code for multiple blocks that you select, hold the **SHIFT** key, select multiple blocks, and then right-click any one block to select **C/C++ Code** > **Navigate to C/C++ Code**. The generated code for the blocks is then highlighted in the HTML code generation report.
8. If you have a referenced model in your model, in the left navigation pane, under **Reference Models**, click the link to a referenced model. The code generation report for the referenced model is now displayed in the MATLAB web browser.
9. In the left navigation pane, you can click the **Back** button to go back to the previous code generation report.

Traceability Tags

A traceability tag appears in a comment above the corresponding line of generated code. The format of the tag is `<system>/block_name`.

- *system* is one of the following:
  - The text `Root`
  - A unique system number assigned by the Simulink engine
- *block_name* is the name of the source block.

The code generator documents the tags for a model in the comments section of the generated header file *model*`.h`. For example, this comment appears in the header file for a model, `foo`, that has a subsystem `Outer` and a nested subsystem `Inner`:

```
/* Here is the system hierarchy for this model.
 *
 * <Root> : foo
 * <S1>   : foo/Outer
 * <S2>   : foo/Outer/Inner
 */
```

This code shows a tag comment above the generated line of code. A Gain block at the root level of a source model generates this code:

```
/* Gain: '<Root>/UnDeadGain1' */
rtb_UnDeadGain1_h = dead_gain_U.In1 *  dead_gain_P.UnDeadGain1_Gain;
```

This code shows a tag comment above the generated line of code. A Gain block within a subsystem one level below the root level of the source model generates this code:

```
/* Gain: '<S1>/Gain' */
dead_gain_B.temp0 *= (dead_gain_P.s1_Gain_Gain);
```

Operator Traceability

The generated code provides traceability between operators in the generated code and Simulink blocks, Stateflow elements, or MATLAB Function blocks.

To verify the generated code by using operator traceability, in the generated code, click an operator hyperlink to highlight the source block in the model.

These operators are supported.

Operator Type	Operators
Arithmetic	`+`, `-`, `*`, `/`, `%`
	`+=`, `-=`, `*=`, `/=`, `%=`
	`++`, `--` (prefix and postfix)
Logical	`!`, `&&`, `||`
Relational	`==`, `!=`, `<`, `>`, `<=`, `>=`
Bit	`~`, `|`, `^`, `&`, `>>`, `<<`
	`&=`, `^=`, `|=`, `<<=`, `>>=`
Conditional	`?:`

These operators are not supported.

Operator Type	Operator Examples
Assignment operator	`=`
Member of and pointer operators	Array subscript: `a[b]`
	Address of and pointer dereference: `&a`, `*a`
	Member of: `a.b`, `a->b`
Other operators	Parenthesis in function call: `foo(a,b)`
	Comma: `a`, `b`

Operator Type	Operator Examples

	Scope resolution: `a::b`
	Cast: `type(a)`
	`new`, `new[]`
	`delete`, `delete[]`

Traceability Limitations

These limitations apply to reports generated by Embedded Coder software:

- Under the following conditions, model-to-code traceability is disabled for a block if the block name contains:
  - A single quote (`'`).
  - An asterisk (`*`) that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (`/`).
  - The character ÿ (`char(255)`).
- If a block name contains a newline character (`\n`), the generated code comment for the block path hyperlink replaces the newline character with a space for readability.
- You cannot trace blocks representing these types of subsystems to generated code:
  - Virtual subsystems
  - Masked subsystems
  - Nonvirtual subsystems for which code has been removed due to optimization

  If you cannot trace a subsystem at subsystem level, you can trace individual blocks within the subsystem.
- If you open a model on a platform that is different from the platform used to generate code, you cannot use model-to-code and code-to-model traceability.
- Inline traceability is not available for files that are generated in `shared_utils` folder.

See Also

Related Examples

- “Trace Simulink Model Elements in Generated Code” on page 85-8
- “Trace Stateflow Elements in Generated Code” on page 85-12
- “Use Traceability in MATLAB Function Blocks” on page 85-34

Trace Simulink Model Elements in Generated Code

To verify the generated code, Embedded Coder provides bidirectional traceability between the Simulink model and the generated code. You can use either method for traceability:

- Code-to-model: The generated code displays these hyperlinks in comment lines in the generated code:
  - Block/subsystems names
  - Line numbers
  - Operators

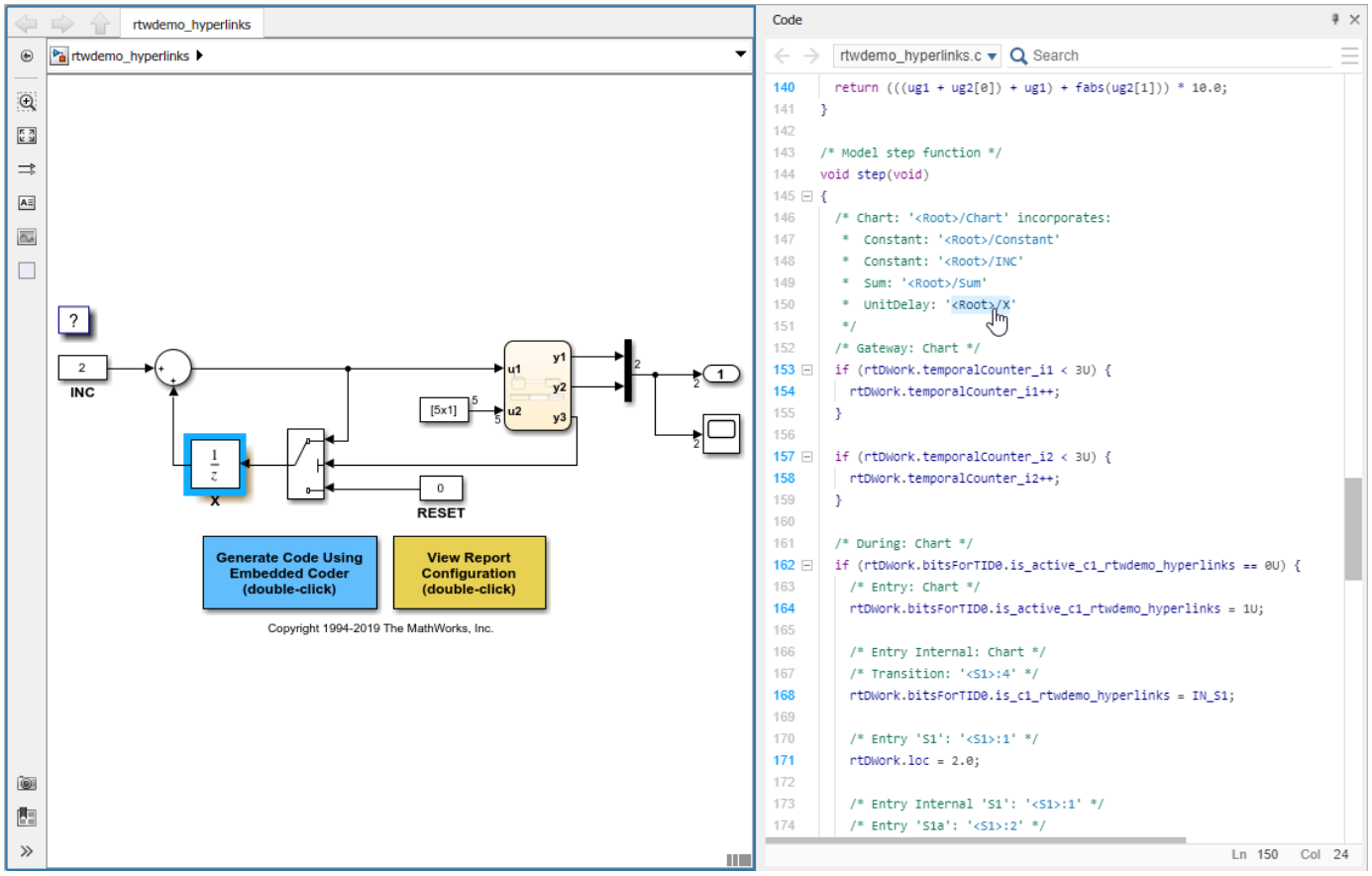
  To highlight the corresponding block or subsystem in the Simulink Editor, click the hyperlinks.
- Model-to-code: You can select single or multiple blocks of a model in the Simulink Editor and navigate to the corresponding generated code.

Note The Code view in code perspective provides bidirectional traceability by default.

Code-to-Model Traceability

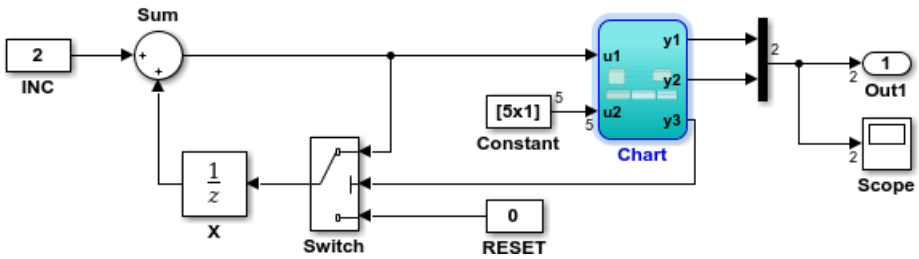
To use hyperlinks for tracing code-to-model elements:

1. Open the model and configure it as an ERT target.
2. Build or generate code for the model. The Code view opens by default to display the generated code next to the model.
3. In the generated code, place your cursor over the block names in comments to highlight source blocks. For example, generate an HTML report for the model `rtwdemo_hyperlinks`. In the generated code for the model step function in `rtwdemo_hyperlinks.c`, place your cursor over the `UnitDelay: '<Root>/X'` block. In the model window, the corresponding `UnitDelay` block is highlighted.



To use line numbers for tracing code-to-model elements:

1. In the previous model, `rtwdemo_hyperlinks`, click the hyperlink at line number 153. The line numbers can differ from the numbers that appear in your generated code.
2. In the model window, the Chart subsystem is highlighted and contains the functionality on line 153.



Traceability to Hidden Blocks

The code generator sometimes inserts hidden blocks during the code generation process for various reasons, for example, to maintain data integrity. Comments for the hidden blocks are included in the generated code. These comments, and the code generated from the hidden block, trace back to the original block in the model that triggered the insertion of the hidden block. Comments to hidden blocks follow this template:

```
/* <Hidden_Block_Type> generated from: <Hyperlinked_Original_Block_Name> */
```

For example,

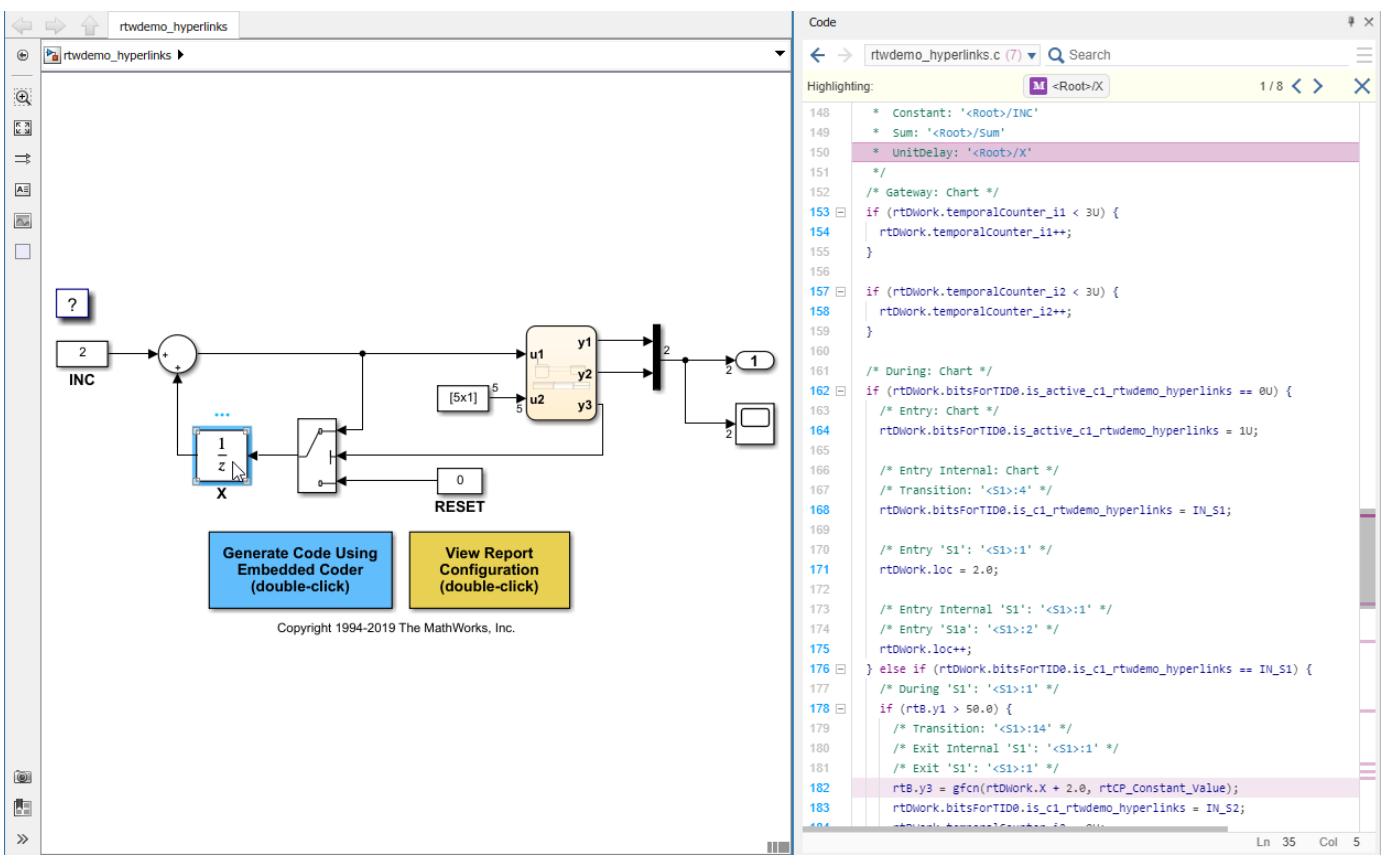
```
/* SignalConversion generated from: '<Root>/Subsystem' */
  Varianttoground_B.VM_Conditional_Signal_Subsystem_0 = 0.0;
```

The hidden block `SignalConversion` is not present in the model. You can trace to the `'<Root>/Subsystem'` block in the model that triggered the insertion of the hidden block by clicking on the hyperlink on `'<Root>/Subsystem'`. For more information on this example, see “Code Generation for Variant Blocks with One Variant Choice” on page 27-17.

Model-to-Code Traceability

To trace model elements to their corresponding generated code:

1. Complete steps 1 and 2 as in “Code-to-Model Traceability” on page 85-8
2. Build or generate code for the model. The generated code is displayed in the Code view window next to the model.
3. In the model window, click on a model element. To select multiple blocks, hold the **SHIFT** key and select additional blocks.
4. In the generated code in Code view window, you see the first instance of highlighted code that is generated for the model element. At the top of the Code view, numbers that appear to the right of generated file names indicate the total number of highlighted lines in each file. This figure shows the result of tracing the Unit Delay block in model `rtwdemo_hyperlinks`.



At the top of the code window, use the navigation bar to move forward and backward through multiple instances of highlighted lines. Use the navigation sidebar to go directly to a line of code.

If you close and reopen a model, the Code view does not open until you open the **Embedded Coder** app. The Code view might not be available because Embedded Coder cannot find a build folder for your model in the current working folder. Reset the current working folder to the parent folder of the existing build folder.

If you are using the code generation report and want to reload existing traceability information, see “Reload Existing Traceability Information” on page 85-27.

See Also

Related Examples

- “Verify Generated Code by Using Code Tracing” on page 85-2
- “Trace Stateflow Elements in Generated Code” on page 85-12

Trace Stateflow Elements in Generated Code

To verify the generated code for your Stateflow elements, you can trace Stateflow elements in your model to the generated code by using these types of navigation:

- Code-to-model: Trace generated code back to the model by clicking hyperlinks in the comments or the hyperlinked line numbers, which highlights the corresponding model element in the Simulink Editor.
- Model-to-code: Trace the model elements in the Simulink Editor to corresponding lines in generated code by right-clicking the model element and navigating to the generated code. This traceability is not supported for some Stateflow elements in Code view.

These examples illustrate how to trace different Stateflow elements.

Inline Traceability for Stateflow Elements

Inline traceability refers to the line-level traceability available in the generated code. You can click the hyper-linked line numbers to trace single or multiple Stateflow elements at the same time.

1. Open the model.

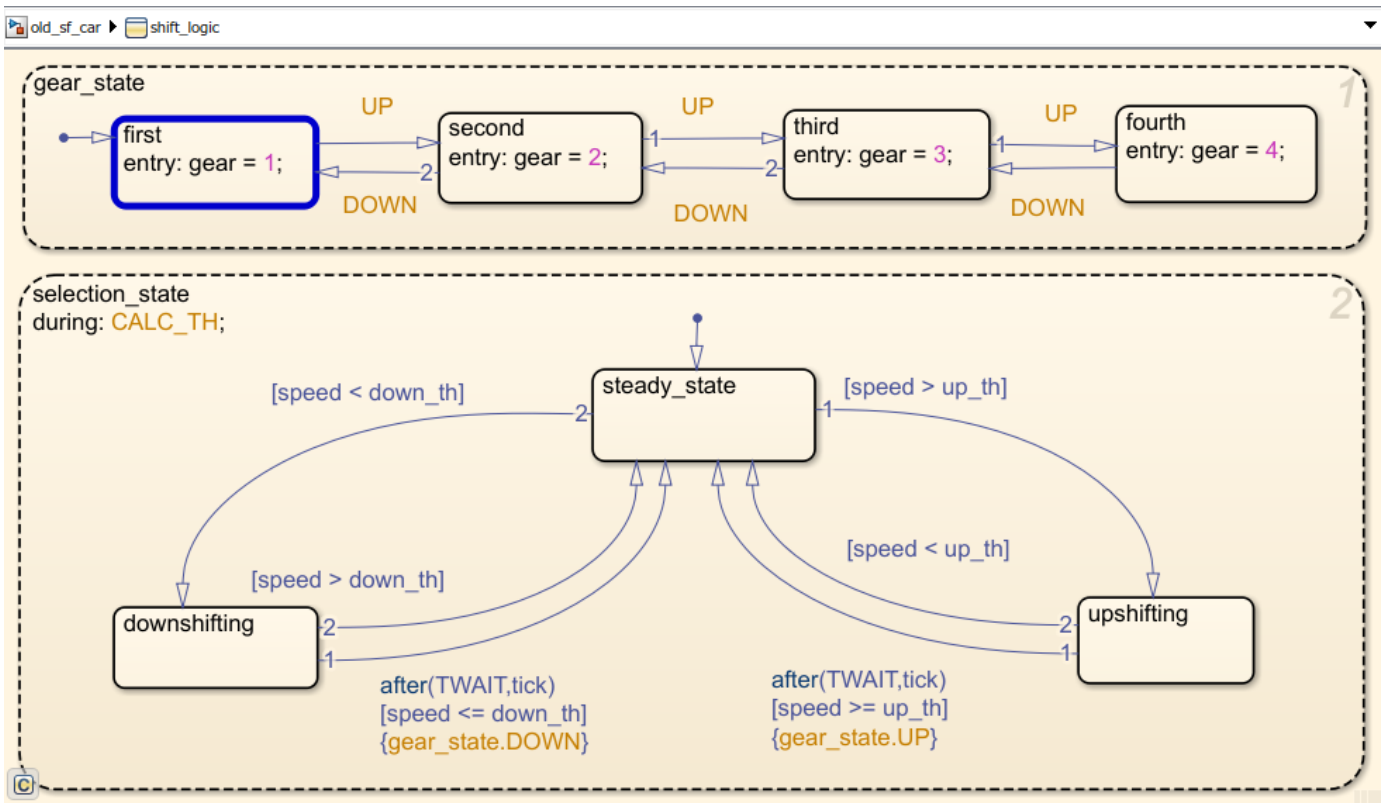
   ```
   openExample('old_sf_car')
   ```

2. Configure the model as an ERT target.
3. On the **Code Generation** > **Report** pane, select these model configuration parameters:
   - **Create code generation report**
   - **Open report automatically**
   - **Code-to-model**
   - **Model-to-code**
4. Select **Code Generation** > **Comments** > **Stateflow object comments** configuration parameter. This parameter enables the traceability comments for Stateflow elements.
5. Go to the **Code Generation** > **Interface** pane. In the **Software environment** section, select **continuous time**. Click **Apply**. Before generating code, you must perform this step because this model contains a block with a continuous sample time.
6. On the **C Code** tab, click **Build**.

   After the code generation process is complete, the code generation report appears.
7. In the report, on the left navigation pane, click the `old_sf_car.c` hyperlink.
8. To the view the inline traceability hyperlinks, scroll through the code. These line numbers can differ from the numbers that appear in your code generation report.

```
185 /* Function for Chart: '<Root>/shift_logic' */
186 static void old_sf_car_gear_state(const int32_T *sfEvent)
187 {
188   switch (old_sf_car_DW.is_gear_state) {
189    case old_sf_car_IN_first:
190     old_sf_car_B.gear = 1.0;
191     if (*sfEvent == old_sf_car_event_UP) {
192       old_sf_car_DW.is_gear_state = old_sf_car_IN_second;
193       old_sf_car_B.gear = 2.0;
194     }
195     break;
```

9 Click the hyperlink on line number `190`. The corresponding element is highlighted.



Bidirectional Traceability for States and Transitions

1 Open the model.

```
openExample('old_sf_car')
```

2 Complete steps 2 through 5 in “Inline Traceability for Stateflow Elements” on page 85-12.

3 Press **Ctrl+B** to generate source code and header files for the `old_sf_car` model that contains the `shift_logic` chart. After the code generation process is complete, the code generation report appears.

4 In the report, click the `old_sf_car.c` hyperlink.

View Results

1 To see the traceability comments, scroll through the code. These line numbers can differ from the numbers that appear in your code generation report.

```
188 /* Function for Chart: '<Root>/shift_logic' */
189 static void old_sf_car_gear_state(const int32_T *sfEvent)
190 {
191   /* During 'gear_state': '<S5>:2' */
192   switch (old_sf_car_DW.is_gear_state) {
193    case old_sf_car_IN_first:
194     old_sf_car_B.gear = 1.0;
195
196     /* During 'first': '<S5>:6' */
197     if (*sfEvent == old_sf_car_event_UP) {
198       /* Transition: '<S5>:12' */
199       old_sf_car_DW.is_gear_state = old_sf_car_IN_second;
200
201       /* Entry 'second': '<S5>:4' */
202       old_sf_car_B.gear = 2.0;
```

2 Click the `<S5>:2` hyperlink in this traceability comment:

```
/* During 'gear_state': '<S5>:2' */
```

The corresponding state appears highlighted in the chart.

3 Click the `<S5>:12` hyperlink in this traceability comment:

```
/* Transition: '<S5>:12' */
```

The corresponding transition appears highlighted in the chart. To remove highlighting from an element in the chart, press the **Esc** key.

4 You can also trace elements in the model to lines of generated code. In the chart, right-click the element `gear_state` and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for that state appears highlighted in `old_sf_car.c`.

```
188 /* Function for Chart: '<Root>/shift_logic' */
189 static void old_sf_car_gear_state(const int32_T *sfEvent)
190 {
191   /* During 'gear_state': '<S5>:2' */
192   switch (old_sf_car_DW.is_gear_state) {
193    case old_sf_car_IN_first:
194     old_sf_car_B.gear = 1.0;
```

5 In the chart, right-click the transition with the condition `[speed > up_th]` and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for that transition appears highlighted in `old_sf_car.c`.

```
492         case old_sf_car_IN_steady_state:
493          /* During 'steady_state': '<S5>:9' */
494          if (old_sf_car_B.mph > interp_up) {
495            /* Transition: '<S5>:18' */
496            old_sf_car_DW.is_selection_state = old_sf_car_IN_upshifting;
497            old_sf_car_DW.temporalCounter_i1 = 0U;
498          } else {
499            if (old_sf_car_B.mph < interp_down) {
500              /* Transition: '<S5>:19' */
```

Note For a list of the Stateflow elements in your model that are traceable, click the **Traceability Report** hyperlink in the code generation report.

Bidirectional Traceability for State Transition Tables

This example shows how to navigate bidirectionally between elements in a state transition table and the generated C/C++ and HDL code for traceability.

1. Open the model. This model is already configured for traceability.

   ```
   openExample('sf_cdplayer_STT')
   ```

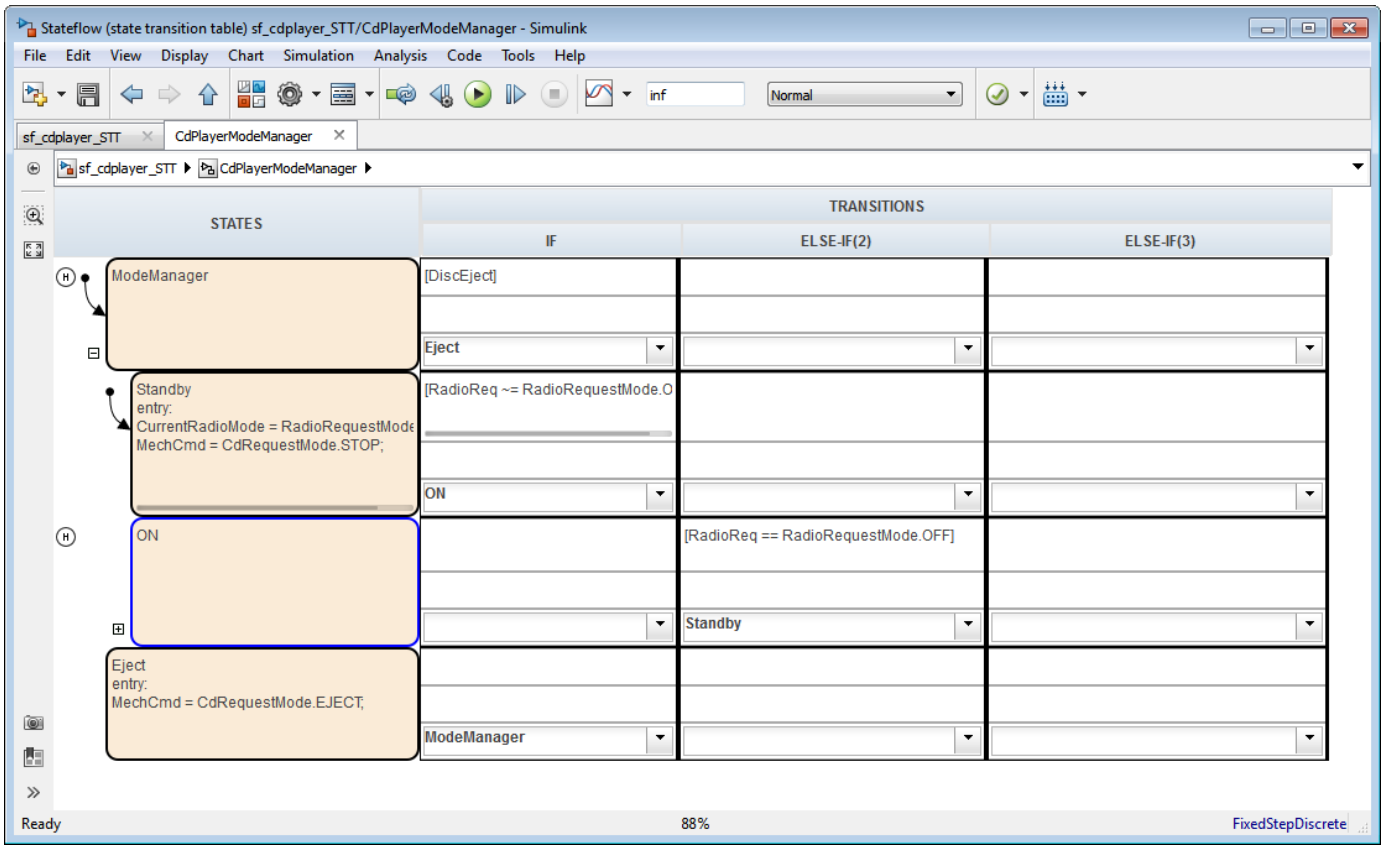
2. Press **Ctrl+B** to generate source code and header files for the `sf_cdplayer_STT` model. After the code generation process is complete, the code generation report appears.
3. Click the `sf_cdplayer_STT.c` hyperlink in the report.
4. To see the traceability comments, scroll through the code. The line numbers shown can differ from the numbers that appear in your code generation report.

```
60  /* Function for State Transition Table: '<Root>/CdPlayerModeManager' */
61  static void sf_cdplayer_S_enter_internal_ON(void)
62  {
63    /* Entry Internal 'ON': '<S2>:58' */
64    switch (sf_cdplayer_STT_DWork.bitsForTID1.was_ON) {
65     case sf_cdplayer_STT_IN_AMMode:
66      sf_cdplayer_STT_DWork.bitsForTID1.is_ON = sf_cdplayer_STT_IN_AMMode;
67      sf_cdplayer_STT_DWork.bitsForTID1.was_ON = sf_cdplayer_STT_IN_AMMode;
68
69      /* Entry 'AMMode': '<S2>:61' */
70      sf_cdplayer_STT_B.CurrentRadioMode = AM;
71      sf_cdplayer_STT_B.MechCmd = STOP;
72      break;
73
```

5. Click the `<S2>:58` hyperlink in this traceability comment:

   ```
   /* Entry Internal 'ON': '<S2>:58' */
   ```

   The corresponding state `'ON'` appears highlighted in the state transition table.



6 You can trace a state or transition from the state transition table to the generated code. Right-click the state `Standby` and select **C/C++ Code** > **Navigate to C/C++ Code**.

The entry code for the state `Standby` is highlighted in the generated code.

```
206     /* During 'ON': '<S2>:58' */
207     if (sf_cdplayer_STT_P.RR_Value == OFF) {
208       /* Transition: '<S2>:71' */
209       sf_cdplayer_ST_exit_internal_ON();
210       sf_cdplayer_STT_DWork.bitsForTID1.is_ModeManager =
211         sf_cdplayer_STT_IN_Standby;
212       sf_cdplayer_STT_DWork.bitsForTID1.was_ModeManager =
213         sf_cdplayer_STT_IN_Standby;
214
215       /* Entry 'Standby': '<S2>:57' */
216       sf_cdplayer_STT_B.MechCmd = STOP;
217     } else if (*RadioReq_prev != sf_cdplayer_STT_DWork.RadioReq_start) {
218       /* Transition: '<S2>:75' */
219       if (sf_cdplayer_STT_P.RR_Value == CD) {
```

Bidirectional Traceability for Truth Table Blocks

1. Open the model.

   ```
   openExample('sf_climate_control')
   ```

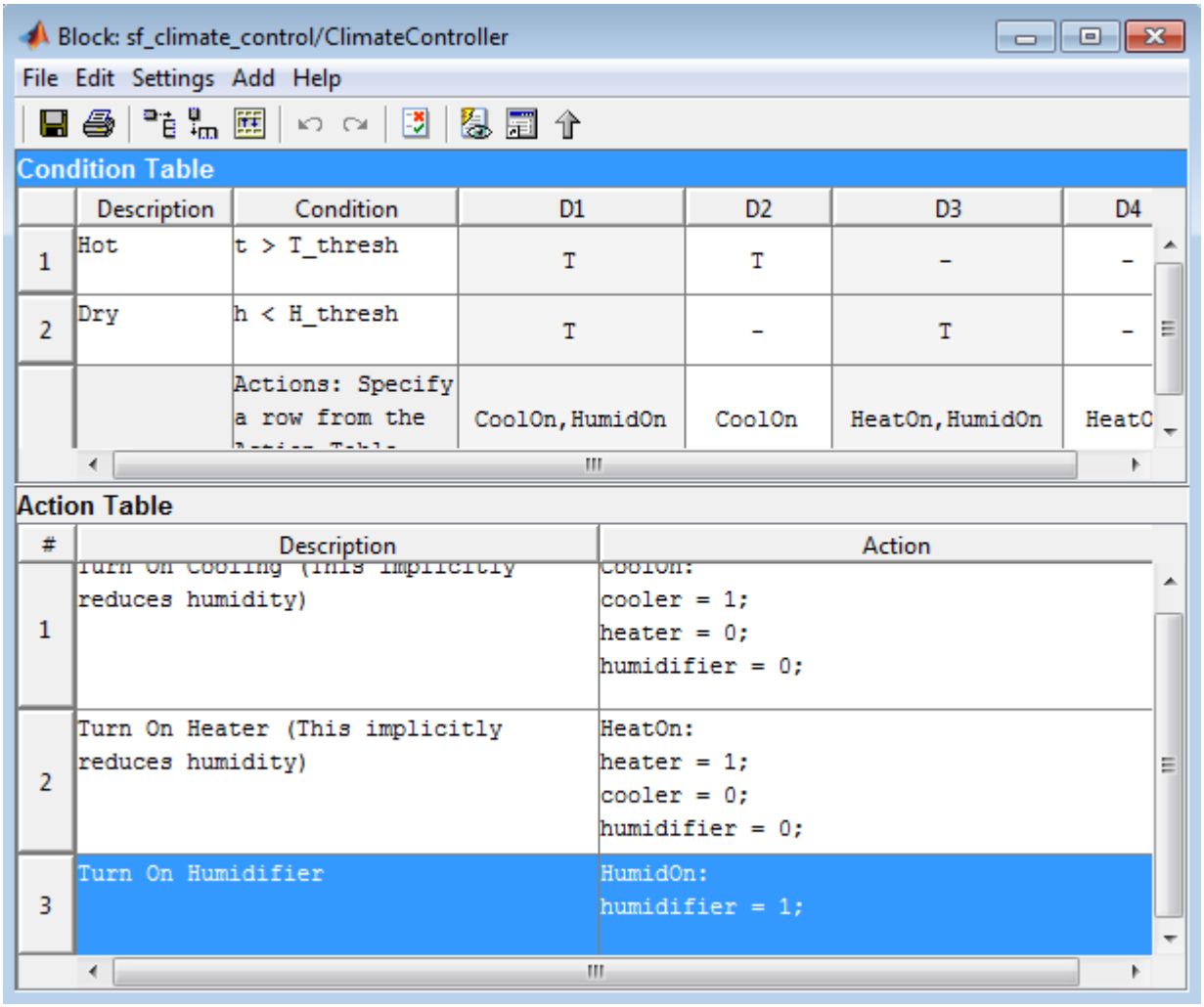
2. Complete steps 2 through 5 in “Inline Traceability for Stateflow Elements” on page 85-12.
3. To build the model, press **Ctrl+B**.
4. In the code generation report, click the `sf_climate_control.c` hyperlink.
5. To see the traceability comments, scroll through the code. These line numbers can differ from the numbers that appear in your code.

   ```
   68      /*  Turn On Humidifier */
   69      /* Action '3': '<S1>:1:47' */
   70      rtb_humidifier = 1;
   71    } else if (aVarTruthTableCondition_1) {
   72      /* Decision 'D2': '<S1>:1:18' */
   73      /*  Turn On Cooling (This implicitly reduces humidity) */
   74      /* Action '1': '<S1>:1:31' */
   75      rtb_cooler = 1;
   ```

6. Click the `<S1>:1:47` hyperlink in this traceability comment:

   ```
   /* Action '3': '<S1>:1:47' */
   ```

   In the Truth Table Editor, row 3 of the Action Table appears highlighted.



7 You can also trace a condition, decision, or action in the table to a line of generated code. For example, right-click a cell in the column D2 and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for that decision appears highlighted in `sf_climate_control.c`.

```
68      /*  Turn On Humidifier */
69      /* Action '3': '<S1>:1:47' */
70      rtb_humidifier = 1;
71    } else if (aVarTruthTableCondition_1) {
72      /* Decision 'D2': '<S1>:1:18' */
73      /*  Turn On Cooling (This implicitly reduces humidity) */
74      /* Action '1': '<S1>:1:31' */
75      rtb_cooler = 1;
```

Tip To select **C/C++ Code** > **Navigate to C/C++ Code** for a condition, decision, or action, right-click a cell in the row or column that corresponds to that truth table element.

Bidirectional Traceability for Graphical Functions

1. Open the model.

   ```
   openExample("stateflow/ClutchWithEnabledSubsystemsExample")
   ```

2. Complete steps 2 through 5 in “Inline Traceability for Stateflow Elements” on page 85-12.
3. In the Configuration Parameters dialog box, go to the **Solver** pane. In the **Solver selection** section, select `Fixed-step` in the **Type** field. Click **Apply**. Before generating code, you must perform this step because the model does not work with variable-step solvers.
4. To build the model, press **Ctrl+B**.
5. In the code generation report, click the `sf_clutch_enabled_subsystems.c` hyperlink.
6. To see the traceability comments, scroll through the code. These line numbers can differ from the numbers that appear in your code generation report.

   ```
   376        /* During 'Slipping': '<S1>:1' */
   377        /* Graphical Function 'detectLockup': '<S1>:10' */
   378        /* Transition: '<S1>:28' */
   379        /* Graphical Function 'getSlipTorque': '<S1>:3' */
   380        /* Transition: '<S1>:12' */
   ```

7. Click the `<S1>:3` hyperlink in this traceability comment:

   ```
   /* Graphical Function 'getSlipTorque': '<S1>:3' */
   ```

   In the chart, the graphical function `getSlipTorque` appears highlighted.
8. You can trace a graphical function in the chart to a line of generated code. For example, right-click the graphical function `detectSlip` and select **C/C++ Code** > **Navigate to C/C++ Code**.

   The code for that graphical function appears highlighted in `sf_clutch_enabled_subsystems.c`.

   ```
   309        /* During 'Locked': '<S1>:2' */
   310        /* Graphical Function 'detectSlip': '<S1>:6' */
   311        /* Transition: '<S1>:15' */
   312        /* Graphical Function 'getLockedTorque': '<S1>:5' */
   313        /* Transition: '<S1>:14' */
   314        if (fabs((5.0 * sf_clutch_enabled_subsystems_B.Tin - (5.0 * 2.0 - 1.0 *
   315               1.0) *
   316                  sf_clutch_enabled_subsystems_X.EngineVehicleIntegrator_CSTATE) /
   317                 (5.0 + 1.0)) > Ratioofstatictokinetic) {
   318          /* Transition: '<S1>:17' */
   ```

Code-to-Model Traceability for Events

1. Open the model.

   ```
   openExample('sf_security')
   ```

2. Complete steps 2 through 5 in “Inline Traceability for Stateflow Elements” on page 85-12.

3 To build the model, press **Ctrl+B**.

4 In the code generation report, click the `sf_security.c` hyperlink.

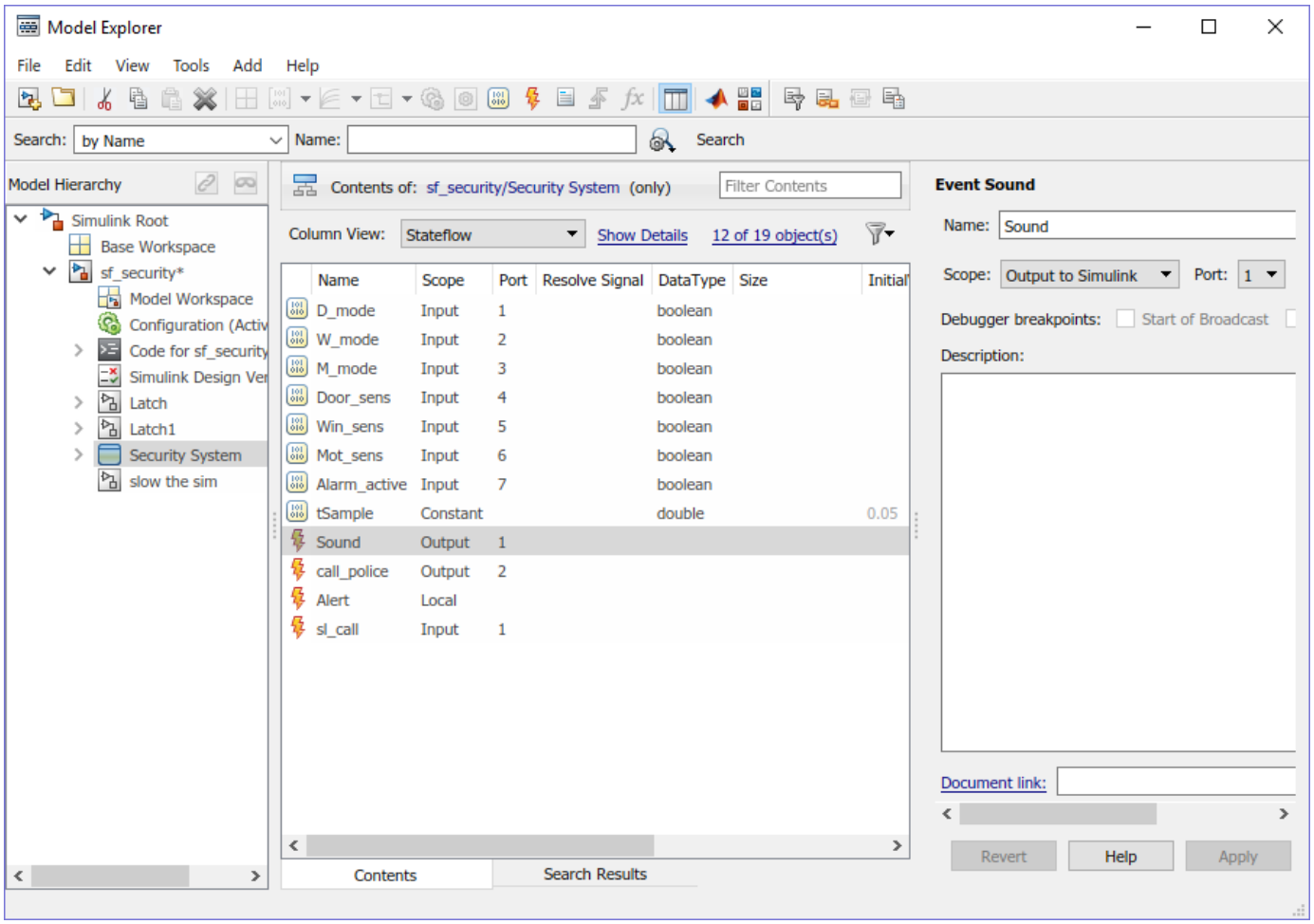
5 To see this traceability comment, scroll through the code. These numbers can differ from the numbers that appear in your code generation report.

```
236         /* Entry 'Pending': '<S3>:5' */
237         /* Event: '<S3>:56' */
238         sf_security_DW.SoundEventCounter++;
239       }
240       break;
```

6 Click the `<S3>:56` hyperlink in this traceability comment:

```
/* Event: '<S3>:56' */
```

In the **Contents** pane of the Model Explorer, the event `Sound` appears highlighted.



Model-to-Code Traceability for Junctions

1 Open the model.

```
openExample('ex_if_else_SF')
```

2. Complete steps 2 through 5 in “Inline Traceability for Stateflow Elements” on page 85-12.
3. To build the model, press **Ctrl+B**.
4. In the code generation report, open the Stateflow chart named `Chart`.
5. Right-click the top-left junction and select **C/C++ Code** > **Navigate to C/C++ Code**.

   The code for the first outgoing transition of that junction appears highlighted in `ex_if_else_SF.c`.

```
39  if (U.u1 > U.u2) {
40    /* Outport: '<Root>/y1' */
41    /* Transition: '<S1>:13' */
42    /* Transition: '<S1>:12' */
43    Y.y1 = U.u1;
```

Format of Traceability Comments for Stateflow Elements

The format of a traceability comment depends on the Stateflow element type.

State

Syntax

```
/* <ActionType> '<StateName>': '<elementHyperlink>' */
```

Example

```
/* During 'gear_state': '<S5>:2' */
```

This comment refers to the `during` action of the state `gear_state`, which has the hyperlink `<S5>:2`.

Transition

Syntax

```
/* Transition: '<elementHyperlink>' */
```

Example

```
/* Transition: '<S5>:12' */
```

This comment refers to a transition, which has the hyperlink `<S5>:12`.

MATLAB Function

Syntax

```
/* MATLAB Function '<Name>': '<elementHyperlink>' */
```

Within the inlined code for a MATLAB function, comments that link to individual lines of the function have this syntax:

```
/* '<elementHyperlink>' */
```

Examples

```
/* MATLAB Function 'test_function': '<S50>:99' */
```

```
/* '<S50>:99:20' */
```

The first comment refers to the MATLAB function named `test_function`, which has the hyperlink `<S50>:99`.

The second comment refers to line 20 of the MATLAB function in your chart.

Truth Table Block

Syntax

```
/* Truth Table Function '<Name>': '<elementHyperlink>' */
```

Within the inlined code for a Truth Table block, comments for conditions, decisions, and actions have this syntax:

```
/* Condition '#<Num>': '<elementHyperlink>' */
/* Decision 'D<Num>': '<elementHyperlink>' */
/* Action '<Num>': '<elementHyperlink>' */
```

`<Num>` is the row or column number that appears in the Truth Table Editor.

Examples

```
/* Truth Table Function 'truth_table_default': '<S10>:100' */
```

```
/* Condition '#1': '<S10>:100:8' */
/* Decision 'D1': '<S10>:100:16' */
/* Action '1': '<S10>:100:31' */
```

The first comment refers to a Truth Table block named `truth_table_default`, which has the hyperlink `<S10>:100`.

The other three comments refer to elements of that Truth Table block. Each condition, decision, and action in the Truth Table block has a unique hyperlink.

Truth Table Function

For syntax and examples, see “Truth Table Block” on page 85-22.

Graphical Function

Syntax

```
/* Graphical Function '<Name>': '<elementHyperlink>' */
```

Example

```
/* Graphical Function 'hello': '<S1>:123' */
```

This comment refers to a graphical function named `hello`, which has the hyperlink `<S1>:123`.

Simulink Function

Syntax

```
/* Simulink Function '<Name>': '<elementHyperlink>' */
```

Example

```
/* Simulink Function 'simfcn': '<S4>:10' */
```

This comment refers to a Simulink function named `simfcn`, which has the hyperlink `<S4>:10`.

Event

Syntax

```
/* Event: '<elementHyperlink>' */
```

Example

```
/* Event: '<S3>:33' */
```

This comment refers to an event, which has the hyperlink `<S3>:33`.

See Also

Related Examples

- “Verify Generated Code by Using Code Tracing” on page 85-2
- “Trace Simulink Model Elements in Generated Code” on page 85-8
- “Trace Data, Events, and Messages” (Stateflow)

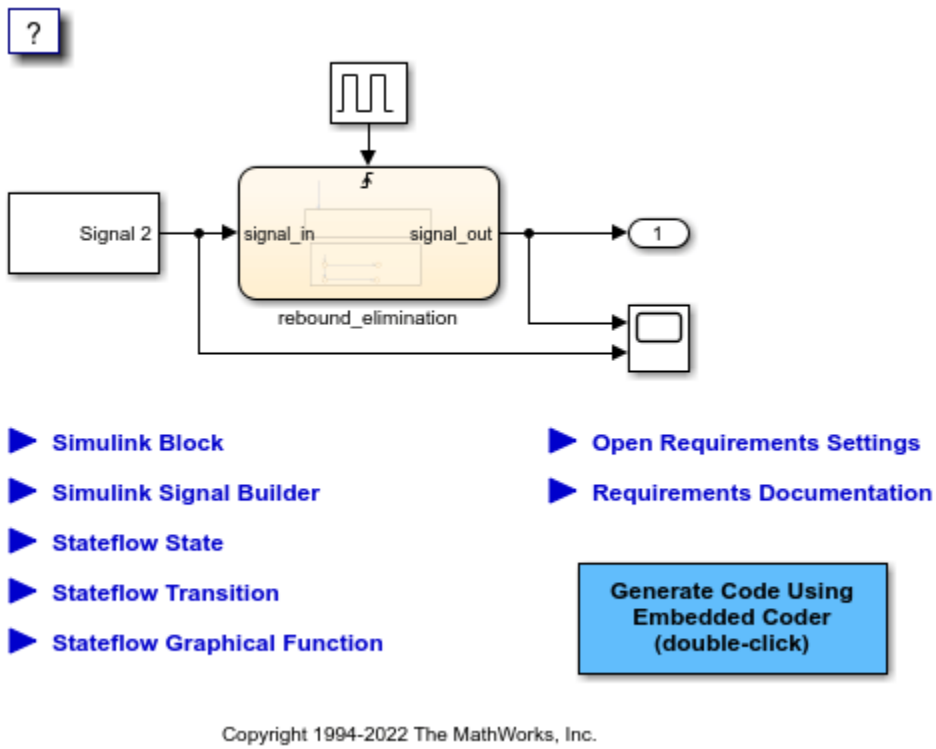
Link Generated Code to Requirements

Link generated code to model element requirements. Using configuration parameters, you can specify whether to include requirement descriptions as comments in the generated code.

Open Model

Open the `rtwdemo_requirements` model. The model contains Simulink® and Stateflow® elements with associated requirements.

```
model='rtwdemo_requirements';
open_system(model);
```



View Requirements

You can view requirements linked to the model by using the Requirements Editor. In the Simulink Editor, open the **Requirements Manager** app. On the **Requirements** tab, click **Requirements Editor**. You can view requirements specific to model objects by using the object context menu. Right-click an element and select **Requirements** > **Open Outgoing Links dialog**. To view the requirements, use these commands:

1. To view the requirements for the `DiscretePulseGenerator` block, right-click the `DiscretePulseGenerator` block and select **Requirements** > **Open Outgoing Links dialog** to open Outgoing Links Editor. Using Outgoing Links Editor, you can create, edit, and delete requirements traceability links.

```
clockblock='rtwdemo_requirements/clock';
clockblockh=get_param(clockblock,'handle');
rmi('edit',clockblockh);
```

2. To view the requirements, open the **Signal Editor** block by double-clicking it.

```
sigeblock='rtwdemo_requirements/Signal Editor';
open_system(sigeblock)
```

3. To view the requirements for the Stateflow® state, open the Outgoing Links Editor.

```
state=find(sfroot,'-isa','Stateflow.State','-and','Tag','req_state');
rmi('edit',state.id);
```

4. To view the requirements for the Stateflow transition, open the Outgoing Links Editor.

```
trans=find(sfroot,'-isa','Stateflow.Transition','-and','Tag','req_trans');
rmi('edit',trans.id);
```

5. To view the requirements for the Stateflow function, open the Outgoing Links Editor.

```
func=find(sfroot,'-isa','Stateflow.Function','-and','Tag','req_function');
rmi('edit',func.id);
```

Close the open windows.

```
close_system(sigeblock);
```

Set Configuration Parameters

Open the Configuration Parameters dialog box **Code Generation > Comments** pane. View the configuration parameter settings.

```
model = bdroot;
configSet = getActiveConfigSet(model);
openDialog(configSet);
```

Generate Code

Generate code for the model.

```
slbuild('rtwdemo_requirements')
```

```
### Starting build procedure for: rtwdemo_requirements
### Successful completion of build procedure for: rtwdemo_requirements

Build Summary

Top model targets built:

Model                 Action                        Rebuild Reason
========================================================================================================
rtwdemo_requirements  Code generated and compiled.  Code generation information file does not ex

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 17.607s
```

In the generated code, view the comments containing the requirements. To view all the requirements, click the hyperlinked requirement comment.

```
rtwdemodbtype('rtwdemo_requirements_ert_rtw/rtwdemo_requirements.c',...
    '/* Function for Chart:','return result;',1,0);
```

```
/* Function for Chart: '<Root>/rebound_elimination' */
static real_T rebound_fcn(real_T prev_in, real_T prev_out, real_T curr_in)
{
  real_T result;

  /* Graphical Function 'rebound_fcn': '<S2>:2':
   *  1. Result Computation
   */
  /* Transition: '<S2>:4' */
  if (prev_in == curr_in) {
    /* Transition: '<S2>:5' */
    result = curr_in;
  } else {
    /* Transition: '<S2>:6' */
    /* Transition: '<S2>:7' */
    result = prev_out;
  }
```

Close Model

```
rtwdemoclean;
close_system('rtwdemo_requirements',0);
```

See Also

Related Examples

- “Link Blocks and Requirements” (Requirements Toolbox)

Reload Existing Traceability Information

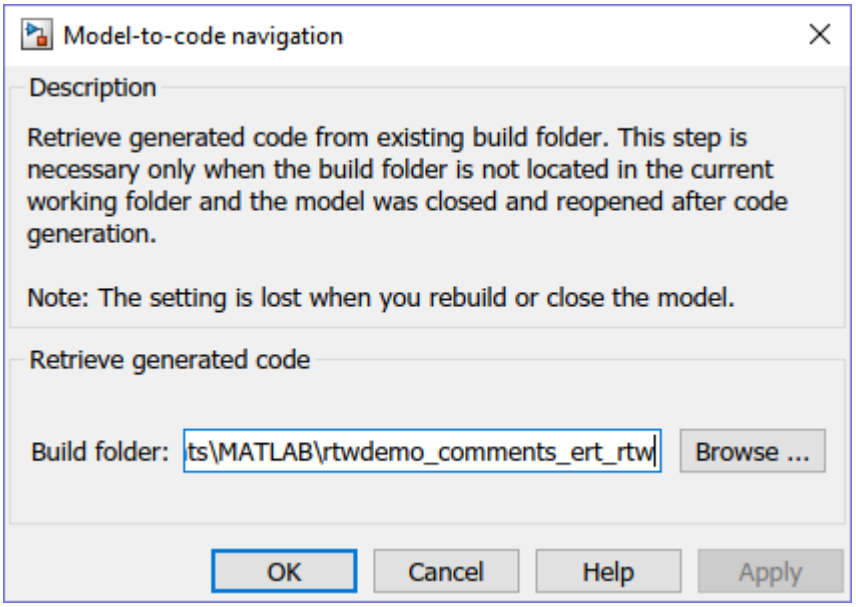
When you build a model, the traceability information is stored in the build folder. The format of the build folder for an ERT-based target is *model*_ert_rtw.

Model-to-code parameter should be selected in the original build to be able to reload the traceability information in a new Simulink session. To reload existing traceability information for a model:

1. Reset the current working folder to the parent folder of the existing build folder.
2. In the Configuration Parameters dialog box, next to **Model-to-code**, click **Configure**.
3. In the Model-to-code navigation dialog box, in the **Build folder** field, type or browse to the build folder that contains the existing traceability information.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available because Embedded Coder cannot find a build folder for your model in the current working folder. In that case, perform these steps:

1. To open the Model-to-code navigation dialog box, click **Configure**.
2. In the Model-to-code navigation dialog box, click **Browse**.
3. Browse to the build folder for your model and select the folder. The build folder path is displayed in the **Build folder** field.



4. If you selected **Model-to-code** for the build, clicking **Apply** or **OK** loads traceability information from the earlier build into your Simulink session.
5. To open the context menu and trace a model element to corresponding code, right-click a model element and select **C/C++ Code** > **Navigate to C/C++ Code**.

See Also

Related Examples

- “Generate Code Generation Report” on page 54-5

Customize Traceability Reports

Even a relatively small model can generate hundreds of lines of C/C++ code. To help you navigate more easily between the generated code and your source model, Embedded Coder provides the traceability report section. When you enable traceability, Embedded Coder display the generated code in Code view or code generation report. You can generate reports in the Configuration Parameters dialog box or at the command line.

In the Configuration Parameters dialog box, the **Code Generation** > **Report** pane lists parameters that you can select and clear to customize the content of your traceability reports.

Select or clear any combination of these parameters. These parameters are on by default.

- **Eliminated / virtual blocks** (account for blocks that are untraceable)
- **Traceable Simulink blocks**
- **Traceable Stateflow elements**
- **Traceable MATLAB functions**

If you select all parameters, you get a complete mapping between model elements and the generated code.

This figure shows the top section of the traceability report, which is generated when you select all traceability content parameters for model `rtwdemo_hyperlinks`. The **Generate Traceability Matrix** button appears only if you have DO Qualification Kit or IEC Certification Kit and are using a Windows host.

Traceability Report for rtwdemo_hyperlinks

Generate Traceability Matrix

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - rtwdemo_hyperlinks
   - rtwdemo_hyperlinks/Chart
   - rtwdemo_hyperlinks/Chart/emfcn

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Unused code path elimination
<Root>/View Code Generation Report	Empty SubSystem

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Root system: rtwdemo_hyperlinks

Object Name	Code Location
<Root>/Chart	rtwdemo_hyperlinks.c:22, 48, 66, 108, 128, 140, 147, 148, 151, 152, 156, 158, 162, 165, 169, 223, 247 rtwdemo_hyperlinks.h:40, 41, 42, 48, 50, 51, 52, 55, 56
<Root>/Constant	rtwdemo_hyperlinks.c:141, 176, 189, 214 rtwdemo_hyperlinks_private.h:26

Generate a Traceability Matrix

The traceability matrix provides traceability among model elements, generated code, and model requirements in a Microsoft Excel® file format.

If you have DO Qualification Kit software or IEC Certification Kit software and are using a Windows host, you can generate a traceability matrix into Microsoft Excel format directly from the traceability report.

Go to the **Traceability Report** section of the HTML code generation report and click **Generate Traceability Matrix**.

To select an existing matrix file to update or specify a matrix file to create, use the options in the Generate Traceability Matrix dialog box. Optionally, you can select and create an order to the columns that appear in the generated matrix. After specifying the location where you want the matrix file, click the **Generate** button.

For more information, see "Generating a Traceability Matrix" in either the DO Qualification Kit documentation (DO Qualification Kit).

See Also

Related Examples

- “Reports for Code Generation” on page 54-2

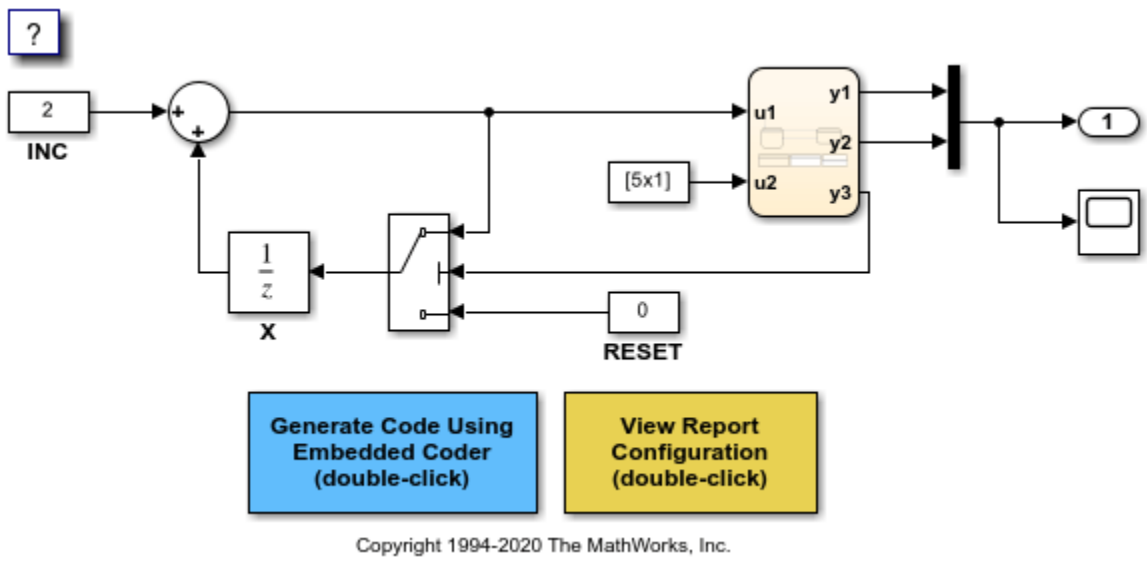
Trace Generated Code to Blocks

Navigate and trace between generated code and its source model for verification.

Open Example Model

Open the example model `rtwdemo_hyperlinks`.

```
model='rtwdemo_hyperlinks';
open_system(model)
```



Code to Model Navigation

Navigate from the generated code to the model.

1. In the Configuration Parameters dialog box, open the **Code Generation > Report** pane.
2. Verify that the following parameters are selected: **Create code generation report**, **Open report automatically** and **Code-to-model**.
3. In the model editor window, press **Ctrl+B** to generate code. After the code generation process is complete, the code generation report appears.
4. In the generated report, click the `rtwdemo_hyperlinks.c` link to see the code with embedded hyperlinks.
5. Click links in the code to trace generated code segments to the highlighted elements in the model. For example, if you click the hyperlink `<Root>/Sum`, the Sum block in the model is highlighted.

Model to Code Navigation

Navigate from the model to the generated code.

1. In the Configuration Parameters dialog box, open the **Code Generation > Report** pane.

2. Verify that the following parameters are selected: **Create code generation report**, **Open report automatically** and **Model-to-code**.
3. In the model editor window, press **Ctrl+B** to generate code. After the code generation process is complete, the code generation report appears.
4. In the model window, right-click any block and select **C/C++ Code** > **Navigate To C/C++ Code**. The software highlights the generated code for this block in the code generation report.
5. Inspect the highlighted lines in the code generation report. The total number of highlighted lines is displayed next to each source file name in the left pane of the report. Use the **Previous** and **Next** buttons to navigate through the highlighted lines.

Note: **Model-to-code** navigation also works for Stateflow® objects and **MATLAB** functions.

Note: The report displays a diagnostic message if there is no code to highlight for the selected block (for example, if you select a virtual block).

Traceability Report

Generate and inspect the **Traceability Report**.

1. In the Configuration Parameters dialog box, open the **Code Generation > Report** pane.
2. Verify that the following parameters are selected: **Create code generation report**, **Open report automatically**
3. Search for the traceability report parameters using keyword `GenerateTraceReport`. Verify all the listed parameters are selected.
4. In the model editor window, press **Ctrl+B** to generate code. After the code generation process is complete, the code generation report appears.
5. To see a report of the untraceable (not in the generated code) and traceable blocks in your model, on the left pane of the generated report click **Traceability Report**.
6. In the **Traceability Report**, inspect the **Eliminated / Virtual Blocks** and **Traceable Blocks** lists. For example, the **Scope** block is an untraceable block. It is listed under **Eliminated / Virtual Blocks** because the code generator does not generate code for this block.

See Also

External Websites

- “Verify Generated Code by Using Code Tracing” on page 85-2
- “Trace Simulink Model Elements in Generated Code” on page 85-8
- “Trace Stateflow Elements in Generated Code” on page 85-12
- “Use Traceability in MATLAB Function Blocks” on page 85-34

Use Traceability in MATLAB Function Blocks

Extent of Traceability in MATLAB Function Blocks

Like other Simulink blocks, MATLAB Function blocks support bidirectional traceability, but extend navigation to lines of source code. That is, you can navigate between a line of generated code and its corresponding line of source code. In other Simulink blocks, you can navigate between a line of generated code and its corresponding object.

In addition, you can select to include the source code as comments in the generated code. When you select **MATLAB source code as comments** parameter, the MATLAB source code appears immediately after the associated traceability tag. For more information, see “Include MATLAB Code as Comments in Generated Code” on page 48-23.

For information about how traceability works in Simulink blocks, see “Verify Generated Code by Using Code Tracing” on page 85-2.

Traceability Requirements

To enable traceability comments in your code, you must have a license for Embedded Coder software. These comments appear only in code that you generate for an Embedded Real-Time (ERT) target.

Note Traceability is not supported for MATLAB files that you call from a MATLAB Function block.

Using Traceability in a MATLAB Function Block

This example shows how to trace between source code and generated code in a MATLAB Function block in the `rtwdemo_dynamicio` model. Follow these steps:

1. Open model `rtwdemo_dynamicio`.
2. Open the **Embedded Coder** app.
3. Change the system target file to `ert.tlc` when prompted. Traceability comments appear hyperlinked in generated code only for embedded real-time (`ert.tlc`) targets.
4. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select these parameters, if not already selected:
   - **Create code generation report**
   - **Code-to-model**
   - **Model-to-code**
5. In the **Code Generation > Comments** pane, select the **MATLAB source code as comments** and **Stateflow object comments** parameters. These parameters control different parts of the traceability comment.
6. In the model window, press **Ctrl+B**.

   This action generates source code and header files for the `rtwdemo_dynamicio` model that contains the `emlHorizontalSum` and `emlVerticalSum` blocks. After the code generation process is complete, the Code view appears automatically to the right of the model and displays the `rtwdemo_dynamicio.c` file.

7 Scroll down through the code to see the traceability comments, which appear as links inside `/*...*/` brackets, as in this example.

```
91  /* Outport: '<Root>/Out1' incorporates:
92   *  Chart: '<Root>/sfVerticalSum'
93   *  MATLAB Function: '<Root>/emlHorizontalSum'
94   */
95  /* MATLAB Function 'emlHorizontalSum': '<S1>:1' */
96  /* '<S1>:1:2' y = sum(u,2); */
97  rtwdemo_dynamicio_Y.Out1 = (tmp[0] * rtwdemo_dynamicio_P.K + tmp[1] *
98    rtwdemo_dynamicio_P.K) + tmp[2] * rtwdemo_dynamicio_P.K;
```

8 Click the `<S1>:1:2` hyperlink in this traceability comment:

```
/* '<S1>:1:2' y = sum(u,2); */
```

Line 2 of the function in the source code appears highlighted in the MATLAB Function Block Editor.

9 You can trace a line in a MATLAB function to lines of generated code. For example, right-click on line 2 of your function and select **Code Generation** > **Navigate to Code** from the context menu.

The code location for line 2 appears highlighted in `rtwdemo_dynamicio.c`.

10 You can trace a line of generated code to a line of source code in a MATLAB function using the line number hyperlinks in the generated code.

See Also

More About

- “Verify Generated Code by Using Code Tracing” on page 85-2
- “Include MATLAB Code as Comments in the Generated Code” on page 48-23

Component Verification in Embedded Coder

Component Verification in Target Environment

After you generate production code for a component design, you need to integrate, compile, link, and deploy the code as a complete application on the embedded system. One approach is to manually integrate the code into an existing software framework that consists of an operating system, device drivers, and support utilities. The algorithm can include externally written legacy or custom code.

An easier approach to verifying a component in a target environment is to use processor-in-the-loop (PIL) simulation. For information about PIL simulations, see “SIL and PIL Simulations” on page 88-2.

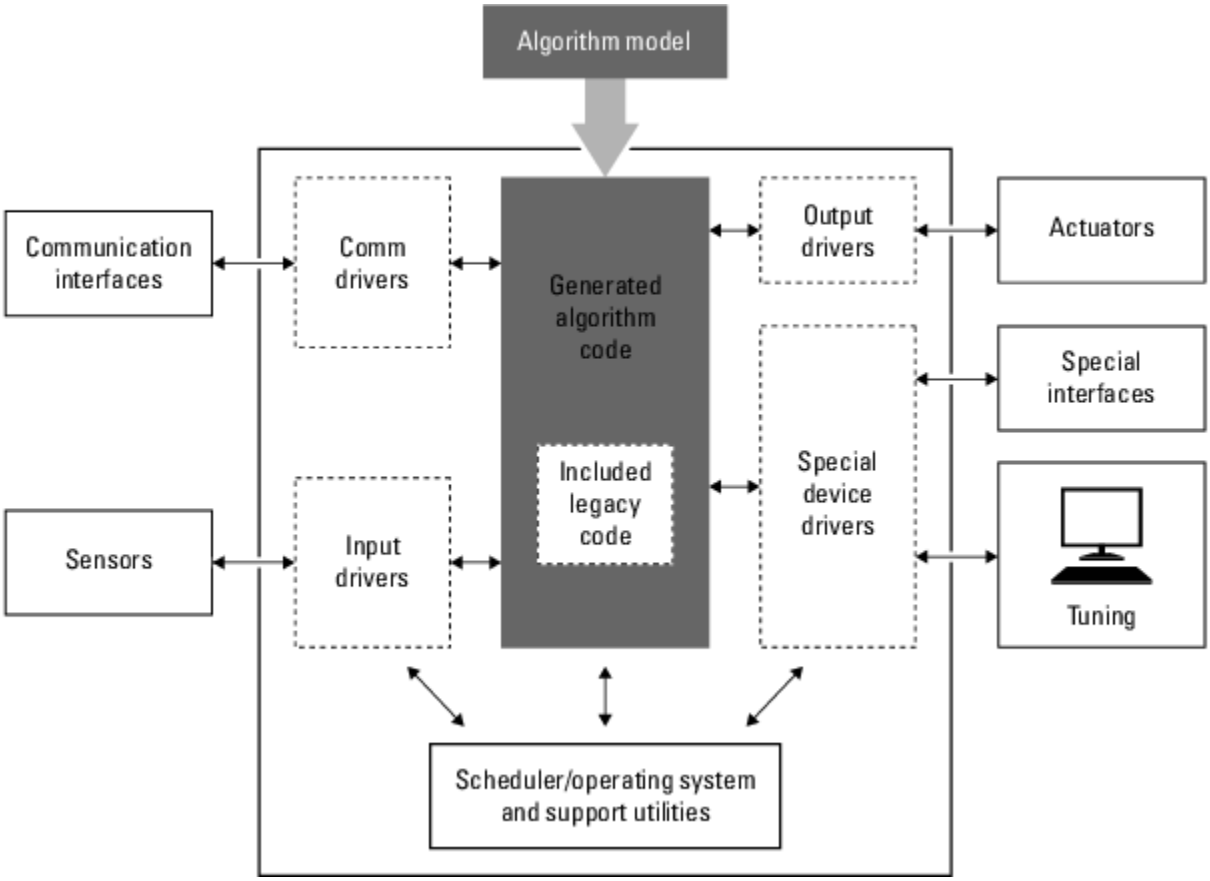
Goals of Component Verification

Assuming that you have generated production source code and integrated required externally written code, such as drivers and a scheduler, you can verify that the integrated software operates as expected by testing it in the target environment. During testing, you can achieve either of the following goals, depending on whether you export code that is strictly ANSI C/C++ or mixes ANSI C/C++ with code optimized for a target environment.

Goal	Type of Code Export
Maximize code portability and configurability	ANSI C/C++
Simplify integration and maximize use of processor resources and code efficiency	Mixed code

Regardless of your goal, you must integrate required external drivers and scheduling software. To achieve real-time execution, you must integrate the real-time scheduling software.

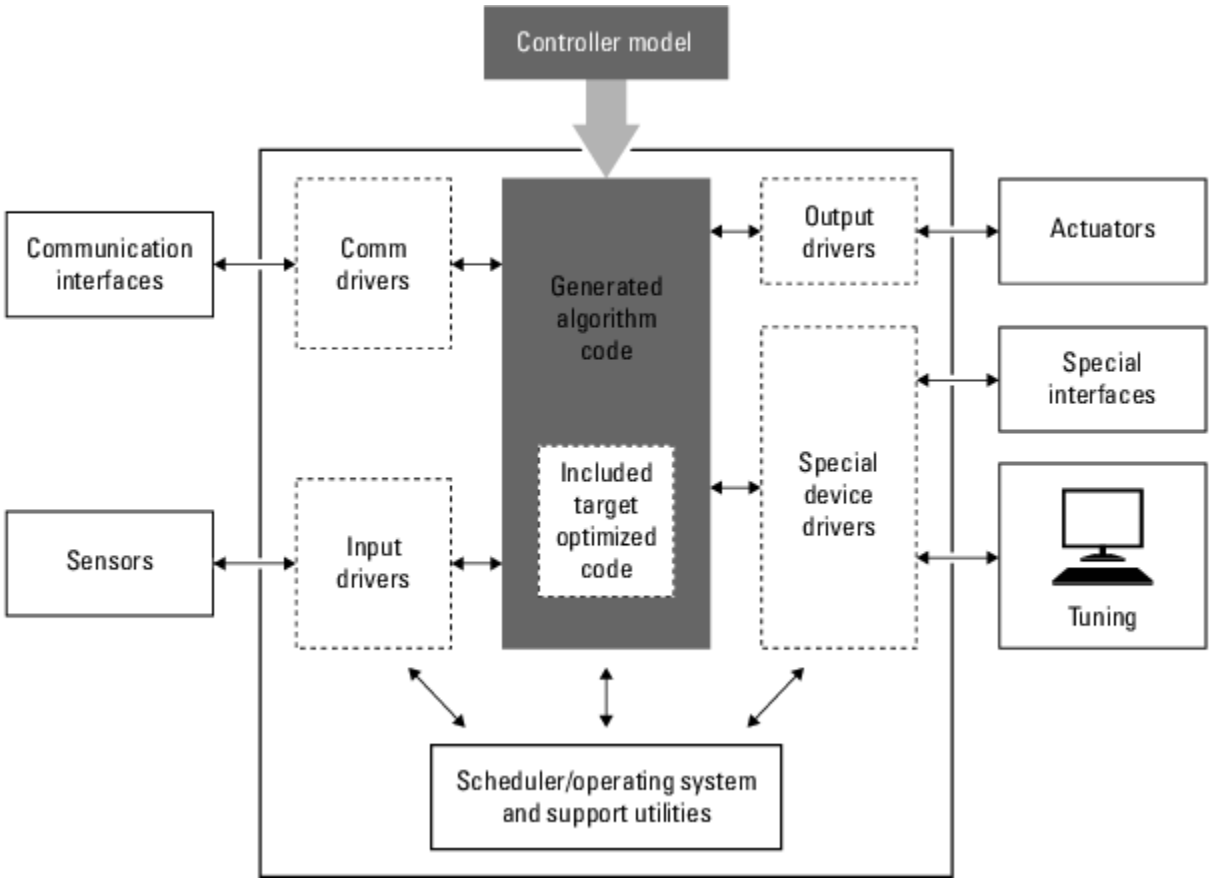
To maximize code portability and configurability, limit the application code to ANSI/ISO C or C++ code only, as the following figure shows.



To simplify code integration and maximize code efficiency for a target environment, use Embedded Coder features for:

- Controlling code interfaces
- Exporting subsystems
- Including target-specific code, including compiler optimizations

The following figure shows a mix of ANSI C/C++ code with code that is optimized for a target environment.



Run Component Tests

The workflow for running software component tests in the target environment is:

1 Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see “S-Functions and Code Generation”. For more specific references that depend on your verification goals, see the following table.

For	See
ANSI C/C++ code integration	“Integrate C Functions Using Legacy Code Tool”. Also, open `rtwdemos` and navigate to the **Custom Code** folder.

For	See
Mixed code integration	• “Generate Component Source Code for Export to External Code Base” on page 58-40 and example `rtwdemo_exporting_functions` • “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114, “Interactively Configure C++ Interface” on page 44-2, and example rtwdemo_fcnprotoctrl • “What Is Code Replacement?” on page 57-2, “What Is Code Replacement Customization?” on page 72-3, and example “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

2. Simulate the integrated component model.
3. Generate code for the integrated component model.
4. Connect to data interfaces for the generated C code data structures. See “Exchange Data Between Generated and External Code Using C API” and “Generate ASAP2 and CDF Calibration Files” on page 64-2.
5. Customize and control the build process, if required. See “Customize Post-Code-Generation Build Processing”, and example `rtwdemo_buildinfo` .
6. Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocate or Share Generated Code”, and example `rtwdemo_buildinfo`.

Component Verification With a Real-Time Target Environment in Embedded Coder

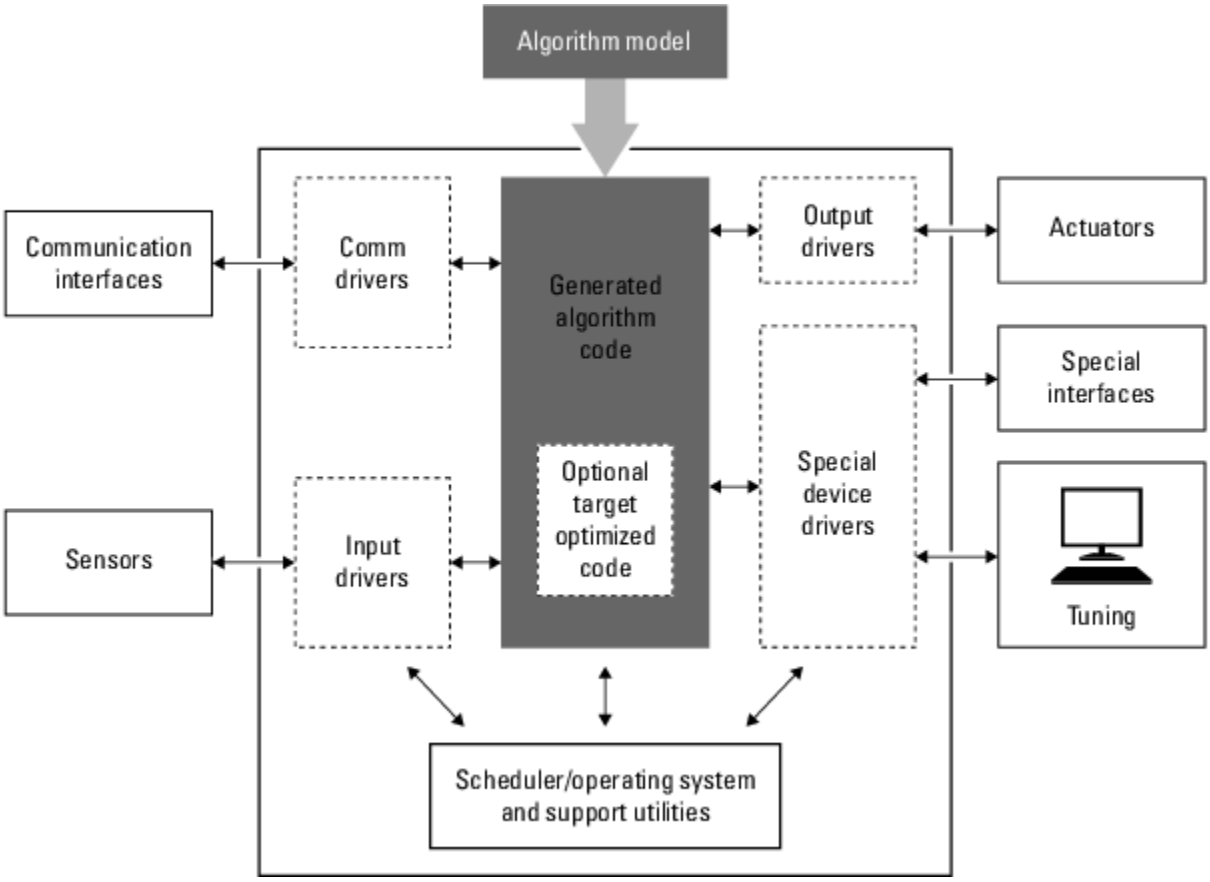
Real-Time Software Component Verification

One approach to verifying a software component is to build the component into a complete software system that can execute in real time in the target environment. A complete software system includes:

- Algorithm for the software component
- Scheduling algorithms
- Calls to drivers for board-specific devices

This single build approach is more time consuming to set up, but makes it easier to get the complete application running in the target environment.

The following figure shows code generated for an algorithm being built into a complete system executable for the target environment.



The workflow for testing component software as part of a complete real-time target environment is:

1. Develop a component model and generate source code for production.

   For information on building in scheduling and real-time system support, see:

   - “Time-Based Scheduling and Code Generation” and “Modeling for Multitasking Execution”. For an example, open `rtwdemos` and navigate to the **Multirate Support** folder.

- “Asynchronous Events” and example `rtwdemo_async`
- “Deploy Applications to Target Hardware” on page 71-2
- “Workflows for AUTOSAR” (AUTOSAR Blockset) and example “Develop a Model That Complies with the AUTOSAR Standard” on page 23-17.

2. Optimize generated code for a specific run-time environment, using specialized function libraries. For more information, see “What Is Code Replacement?” on page 57-2, “What Is Code Replacement Customization?” on page 72-3, and “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31.
3. Customize post code generation build processing to accommodate third-party tools and processes, as required. See “Customize Post-Code-Generation Build Processing” and example `rtwdemo_buildinfo`.
4. Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see “S-Functions and Code Generation”. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	“Integrate C Functions Using Legacy Code Tool”. Also, open `rtwdemos` and navigate to the **Custom Code** folder.
Mixed code integration	• “Generate Component Source Code for Export to External Code Base” on page 58-40 and example `rtwdemo_exporting_functions` • “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114, “Interactively Configure C++ Interface” on page 44-2, and example rtwdemo_fcnprotoctrl • “What Is Code Replacement?” on page 57-2, “What Is Code Replacement Customization?” on page 72-3, and example “Optimize Generated Code by Developing and Using Code Replacement Libraries - Simulink” on page 79-31

5. Simulate the integrated model.
6. Generate code for the integrated model.
7. Connect to data interfaces for the generated C code data structures. See “Exchange Data Between Generated and External Code Using C API” and “Generate ASAP2 and CDF Calibration Files” on page 64-2.
8. Customize and control the build process, as required. See “Customize Post-Code-Generation Build Processing”, and example `rtwdemo_buildinfo` .
9. Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocate or Share Generated Code”, and example `rtwdemo_buildinfo`.

Numerical Equivalence Checking in Embedded Coder

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Configure and Run SIL Simulation” on page 88-15
- “Configure and Run PIL Simulation” on page 88-24
- “Unit Test Subsystem Code with SIL/PIL Manager” on page 88-31
- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 88-33
- “Debug Generated Code During SIL or PIL Simulation” on page 88-35
- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Host-Target Communication for Simulink PIL Simulation” on page 88-42
- “Specify Hardware Timer for Simulink” on page 88-45
- “Set Up PIL Connectivity by Using target Package” on page 88-48
- “Custom Toolchain Directives Required for Code Coverage and Execution Profiling” on page 88-68
- “PIL Simulation Sequence” on page 88-69
- “Verification of Code Generation Assumptions” on page 88-71
- “View SIL and PIL Files in Code Generation Report” on page 88-74
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “Security for PIL Simulations” on page 88-82
- “SIL and PIL Limitations” on page 88-83
- “Test Generated Code with SIL and PIL Simulations” on page 88-94
- “Create a Target Communication Channel for Processor-in-the-Loop (PIL) Simulation” on page 88-108
- “Configure Processor-In-The-Loop (PIL) for a Custom Target” on page 88-113
- “Field-Oriented Control of Permanent Magnet Synchronous Machine” on page 88-118
- “Check Configuration” on page 88-135
- “Verify Numerical Equivalence with CGV” on page 88-136
- “Verify Numerical Equivalence Between Two Modes of Execution of a Model” on page 88-137

SIL and PIL Simulations

In this section...

"What Are SIL and PIL Simulations?" on page 88-2
"Why Use SIL and PIL" on page 88-2
"How SIL and PIL Simulations Work" on page 88-3
"Comparison of SIL and PIL Simulations" on page 88-4
"Code Interfaces for SIL and PIL" on page 88-4
"Scheduling Considerations" on page 88-6
"Imported Data and Additional Generated Code" on page 88-6

What Are SIL and PIL Simulations?

With Embedded Coder, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations of your model. These simulations generate source code for either the top model or part of the model. A SIL simulation compiles and runs the generated code on your development computer. A PIL simulation cross-compiles source code on your development computer, and then downloads and runs the object code on a target processor or an equivalent instruction set simulator.

With SIL and PIL simulations, you can:

- Test whether your model and generated code are numerically equivalent.
- Observe code coverage.
- Perform code execution profiling.

Why Use SIL and PIL

Through SIL and PIL, you can early on test and fix defects. For example, you can model and test a system component in normal mode. Then, you can reuse your test suites in a SIL or PIL simulation that runs compiled generated code. To check numerical equivalence, you compare normal and SIL or PIL simulation results. You do not have to leave the Simulink environment to test generated code on a separate infrastructure.

This table describes situations where you can use SIL and PIL.

Situation	Use
Test numerical equivalence between model and generated code by reusing test vectors developed for normal mode simulation to verify numerical output of generated (or legacy) code. See "SIL/PIL Manager Verification Workflow" on page 88-76 and "Test Two Simulations for Equivalence" (Simulink Test)	SIL and PIL
Collect metrics for generated code: • Code coverage. See "Code Coverage". • Execution profiling. See "Create Execution-Time Profile for Generated Code" on page 81-2	SIL and PIL

Situation	Use
(Requires an IEC Certification Kit license.) Perform verification and validation activities on the generated C/C++ code in accordance to the requirements of the overall software safety lifecycle as defined by the ISO 26262-6, IEC 61508-3, IEC 62304, EN 50128, and EN 50657 functional safety standards. For detailed information about the SIL and PIL verification activities in the IEC Certification Kit Model-Based Design workflow, see the *Reference Workflow for Embedded C/C++ Applications* (`certkitiec_ecoder_workflow.docx/pdf`) artifact in the Artifacts Explorer.	SIL and PIL
(Requires a DO Qualification Kit license.) Test the output of the integration process to satisfy relevant objectives in the DO-178C and DO-333 safety standards. For additional information, see Testing of Outputs of Integration Process (DO Qualification Kit). This information is available also in the *Model-Based Design Workflow for DO-178C* (`qualkitdo_do178_workflow.pdf`) artifact in the Artifacts Explorer.	SIL and PIL
Without target hardware, get a convenient alternative to PIL.	SIL
With target hardware, for example, an evaluation board or instruction set simulator: • Verify behavior of target-specific code, for example, code replacement optimizations, and legacy code. See “What Is Code Replacement?” on page 57-2 and “What Is Code Replacement Customization?” on page 72-3. • Optimize the execution speed and memory footprint of your code. In this table, see the information about collecting execution profiling and stack profiling metrics. • Investigate effects of compiler settings and optimizations, for example, deviation from ANSI C overflow behavior. Normal simulation techniques do not account for restrictions and requirements that the hardware imposes, such as limited memory resources or behavior of target-specific optimized code. For information about running PIL simulations on specific targets, see “Sample Custom Targets” in the Simulink Coder documentation.	PIL

Note The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see “What Is Acceleration?”.

How SIL and PIL Simulations Work

In a SIL or PIL simulation, code is generated for either the top model or part of the model. With SIL, this code is compiled for and executed on your development computer. With PIL, the code is cross-compiled for the target hardware and runs on the target processor.

Through a communication channel, Simulink sends stimulus signals to the code on your computer or target processor for each sample interval of the simulation.

- For a top model, Simulink uses stimulus signals from the base or model workspace.
- If you have designated only part of the model to simulate in SIL or PIL mode, then a part of the model remains in Simulink and code is not generated for this part of the model. Typically, you configure this part of the model to provide test vectors for the software executing on the hardware. This part of the model can represent other parts of the algorithm or the environment in which the algorithm operates.

When your computer or target processor receives signals from Simulink, the processor executes the SIL or PIL algorithm for one sample step. The SIL or PIL algorithm returns output signals calculated during this step to Simulink through a communication channel. One sample cycle of the simulation is complete, and Simulink proceeds to the next sample interval. The process keeps repeating itself and the simulation progresses. SIL and PIL simulations do not run in real time. In each sample period, Simulink and the object code exchange I/O data.

Comparison of SIL and PIL Simulations

Type of SIL or PIL Simulation	What Happens in SIL Simulation	What Happens in PIL Simulation
Specify through: • Top-model simulation mode • Model block **Simulation mode** parameter	• Test behavior of generated source code on development computer. Simulation does not test code compiled for target hardware because code is compiled for the development computer (different compiler and different processor architecture than the target). • Generated production code compiled and executed on development computer as separate process, independent of MATLAB process. • Execution is host/host and nonreal time.	• Test object code that you intend to deploy in production on either real target hardware or an instruction set simulator. • On development computer, generated production code cross-compiled for target. Object code downloaded and executed on target processor or instruction set simulator. • Execution is host/target and nonreal time.
Use SIL or PIL block created from subsystem.	• Simulation runs compiled object code through S-function. S-function communicates with object code executing as standalone application on development computer. SIL block execution is independent of the MATLAB process. • Execution is host/host and nonreal time.	• Simulation runs cross-compiled object code through S-function on development computer. S-function communicates with object code executing as standalone application on target processor or instruction set simulator. • Execution is host/target and nonreal time.

Code Interfaces for SIL and PIL

You generate standalone code when you perform, for example, a top-model or right-click subsystem build for a single deployable component. You can compile and link standalone code into a standalone executable or integrate it with other code. For more information on the standalone code interface, see “Configure Generated C Function Interface for Model Entry-Point Functions”.

When you generate code for a referenced model hierarchy, the software generates standalone executable code for the top model and a library module called a *model reference target* for each referenced model. When the code executes, the standalone executable invokes the applicable model reference targets to compute the referenced model outputs. For more information, see “Build Model Reference Targets”.

To integrate generated code with legacy code, use standalone code because the standalone code interface is documented.

Note SIL and PIL simulations do not provide direct support for custom code interfaces. You can incorporate these interfaces into Simulink as an S-function, for example, using the Legacy Code Tool, S-Function Builder, or handwritten code. Then, you can verify the custom code by using SIL and PIL simulations.

This table provides the interfaces that SIL and PIL simulations generate.

SIL/PIL Simulation	Code Interface
Top-model	SIL/PIL simulation generates the standalone code interface. If the code exists, simulation calls the standalone code for the model . If the code does not exist, simulation generates the standalone code.
Model block	If you set the **Code interface** block parameter to `Top model`, SIL/PIL simulation generates the standalone code interface. Simulation calls the standalone code for the model if it exists. Otherwise, simulation generates standalone code by using the `slbuild('model')` command. If you set the **Code interface** block parameter to `Model reference`, SIL/PIL simulation generates the model reference code interface. Simulation calls the model reference target for the Model block if it exists. Otherwise, simulation generates the model reference target by using the `slbuild('model', 'ModelReferenceCoderTarget')` command.
SIL or PIL block	The block uses the standalone code interface.

Scheduling Considerations

Item	Information
Algebraic loops	There are algebraic loops that occur in SIL and PIL simulations but not in normal mode simulations: • **Single output/update function** in code generation optimization can introduce algebraic loops because the option introduces direct feedthrough via a combined output and update function. **Single output/update function** is not compatible with **Minimize algebraic loop occurrences** (in the Subsystem Parameters dialog box and **Configuration Parameters** > **Model Referencing** pane). **Minimize algebraic loop occurrences** allows code generation to remove algebraic loops by partitioning generated code between output and update functions to avoid direct feedthrough. • If you generate code for a virtual subsystem, code generation treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and introduce inconsistencies to the simulation behavior. To enable consistent simulation and execution behavior for your model, declare virtual subsystems as atomic subsystems. For more information, see: • “Algebraic Loop Concepts” • “Algebraic Loops” • “Simulink Modeling Components”
Exported functions in feedback loops	If your model has function-call subsystems and you export a subsystem that has context-dependent inputs (for example, feedback signals), the results of a SIL/PIL simulation with the generated code and the results of the normal mode simulation of your model can differ. One approach to make SIL/PIL and normal mode simulations yield identical results is to use Function-Call Feedback Latch blocks in your model. You can make context-dependent inputs become context-independent. If you set the **Context-dependent inputs** configuration parameter to `warning`, Embedded Coder generates a warning that identifies context-dependent inputs of exported function-call subsystems. For more information, see: • “Simulink Modeling Components” • Function-Call Feedback Latch • “Context-dependent inputs”

Imported Data and Additional Generated Code

A SIL or PIL simulation does not modify the code generated from the model. If you use imported data, the SIL or PIL simulation generates additional code that is required to support compiling and linking

of the SIL or PIL target application. The additional generated code corresponds to interfaces that you specify in the model.

This section describes:

- For imported data, when the SIL or PIL simulation defines storage and when you must define storage.
- Interfaces for which a SIL or PIL simulation generates additional code

Imported Data in SIL or PIL Simulations

In SIL and PIL simulations, you can use signals, parameters, and data stores that specify storage classes with imported data definitions. The simulations define storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary).
- Parameters in the base workspace or a data dictionary. For parameters in the model workspace:
  - Top-model SIL/PIL and SIL/PIL block simulations define storage.
  - Model block SIL/PIL simulations do not define storage. You must define storage and specify initial values that match MATLAB values.
- Global data stores.

SIL and PIL simulations do not define storage for other imported data. For example, the simulations do not define storage for imported data associated with:

- Internal signals (not on the I/O boundary). Note that, if the data for internal signals is in structures that are imported through the use of pointers, the simulations define storage.
- Local data stores.

In these cases, define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API.

See also “Tunable Parameters and SIL/PIL” on page 88-83.

Interfaces That Produce Additional Generated Code

This table describes scenarios where a SIL or PIL simulation generates additional code for interfaces.

Scenario	Details
`GetSet` custom storage class	SIL and PIL simulations support the `GetSet` custom storage class. The SIL/PIL test harness provides C definitions of the `Get` and `Set` functions that are used during simulations. For more information, see “Access Data Through Functions with Storage Class GetSet” on page 40-25.

Scenario	Details

Custom storage class of type `Other`	To enable SIL and PIL support for the custom storage class where **Type** is set to `Other` , create a custom attributes class for the custom storage class and associate the custom attributes class with a Boolean property, `SupportSILPIL`, set to `true`.
	
	`classdef CSCOtherAttributes < Simulink.CustomStorageClassAttributes`
	`  properties(PropertyType = 'logical scalar')`
	`  SupportSILPIL = true;`
	`  end`
	`end`
	
	For more information about custom attributes, see “Further Customize Generated Code by Writing TLC Code” on page 40-22 and “Finely Control Data Representation by Writing TLC Code for a Storage Class” on page 40-23.
	
	To build the SIL or PIL application interface, the code generator calls the `DataAccess` and `ClassAccess` functions in the associated custom TLC file to get required information. The code generator stores the extracted information in build artifacts in the build folder.
	
	For a custom storage class that is not grouped:
	
	• The code generator calls `DataAccess` with the request argument taking values of `define`, `declare`, `layout`, `contents`, `address`, or `set`.
	• The code generator uses the string returned by `DataAccess(record, "define", "", "")` to define the variable in the SIL or PIL application if one of these is true:
	  • The signal or parameter has an `Imported` data scope.
	  • The model uses a model reference code interface.
	  • The model uses a top model code interface, `EnableDataOwnership` is `on`, and the `Owner` attribute for the custom storage class is not empty and not equal to the name of the current model
	• The code generator uses the string returned by `DataAccess(record, "declare", "", "")` to declare the variable in the SIL or PIL application if the following are true:
	  • The model uses a top model code interface.
	  • The signal or parameter uses an `Exported` storage class.
	  • `EnableDataOwnership` is `off`, or `EnableDataOwnership` is `on` and the `Owner` attribute for the custom storage class is empty or equal to the model name.
	  • Code packaging is configured so that the variable is not declared in *`model`*`.h` or a header file that is included by *`model`*`.h`
	
	For a custom storage class that is grouped:
	
	• The code generator calls `DataAccess` with the `request` argument taking values of `layout`, `address`, or `set`.

Scenario	Details

	• The code generator calls `ClassAccess` with the `request` argument taking the value of `groupTypeDeclDefn`.
	• You must provide the grouped type (`struct`) definition and the `extern` declaration of the grouped variable if one of these is true:
	  • The signal or parameter has an `Imported` data scope.
	  • The model uses a model reference code interface.
	  • The model uses a top model code interface, `EnableDataOwnership` is `on`, and the `Owner` attribute for the custom storage class is not empty and not equal to the name of the current model
	  Provide the definition and declaration in a header file associated with the custom storage class by using the `HeaderFile` attribute or custom code that you include through the *`model`*`.h` file. To define the variable in the SIL or PIL application, the code generator uses the string returned by `ClassAccess(record, "groupTypeDeclDefn")`.
	• Static initialization can assume an order of `struct` elements that is different from the order that is generated if the data scope is `Exported`. When the code generator queries `ClassAccess(record, "groupTypeDeclDefn")`, it temporarily overrides the data initialization attribute of the custom storage class with the value `None`.
	
	To determine whether the SIL or PIL application can access the variable in the code by address, the code generator uses the elements returned by `DataAccess(record, "layout", "", "")`. To create the functionality in the application that transfers input or output port, tunable parameter, or global data store memory values between the development computer and target hardware, the code generator uses the output from:
	• `DataAccess(record, "address", idx, reim)` if the first returned element is `scalar`, `vector`, `row-mat`, or `col-mat`.
	• `DataAccess(record, "contents", idx, reim)` (or `DataAccess(record, "set", idx, reim)`) if the first returned element is `other`.
	
	The code generator assumes that for `row-mat` and `col-mat`, matrices are stored in row-major format respectively. The assumption is independent of the array layout for the rest of the model. The code generator assumes that if the array layout of the storage implemented by the custom storage class differs from the rest of the model, the TLC file associated with the custom storage file performs the required transformations.
	
	You can construct the custom TLC file associated with a custom storage class of type `Other` to perform other functions (in addition to returning the requested code fragments). For example, write directly to a custom file or call MATLAB functions that change the state of the base workspace. If you do not always want to execute these functions when `DataAccess` or `ClassAccess` are called, use the `LibIsAccessingCustomDataForSILPIL(record)` TLC function to distinguish between target code generation and requests for code fragments for the construction of the SIL or PIL application. For example:

Scenario	Details

	``
	...
	%case "contents"
	  %if !LibIsAccessingCustomDataForSILPIL(record)
	  %matlab functionWithSideEffects()
	  %endif
	%return LibDefaultCustomStorageContents(record, idx, reim)
	...
	``
	
	See also Other custom storage class limitations on page 88-87.

AUTOSAR Runtime Environment	You can use top-model and Model block SIL/PIL and SIL/PIL block simulations to perform model-based testing of an AUTOSAR software component. The software links generated code for the AUTOSAR software component with a basic component-specific AUTOSAR Runtime Environment (RTE) to create a test application. This application tests AUTOSAR API calls made by the AUTOSAR software component. For a top-level AUTOSAR software component that contains referenced models, you can run top-model or Model block (**Code interface** set to `Top model`) SIL or PIL simulations. In the simulations, the software: • Before compilation of referenced models, generates AUTOSAR RTE header files. • Provides the RTE include path for referenced model compilation. You can also run Model block (**Code interface** set to `Model reference`) SIL or PIL simulations for a referenced model within the top-level AUTOSAR SWC. In this case, before you run a simulation, you must build the parent component to generate the RTE header files. If you do not build the parent component, the SIL or PIL simulation fails.

See Also

Related Examples

- “Test Generated Code with SIL and PIL Simulations” on page 88-94
- “Choose a SIL or PIL Approach” on page 88-11
- “Configure and Run SIL Simulation” on page 88-15
- “Check Configuration” on page 88-135

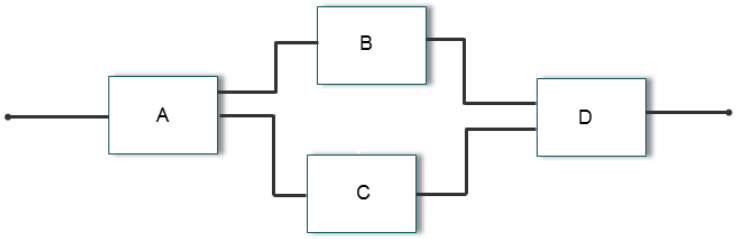
Choose a SIL or PIL Approach

In this section...

"Test Top-Model Code" on page 88-12
"Test Referenced Model Code" on page 88-12
"Test Subsystem Code" on page 88-12
"Summary" on page 88-13

Consider a top model that consists of components A, B, C, and D:

- A and B are existing components for which code has previously been generated and tested.
- C, a referenced model, and D, a subsystem, are new components.



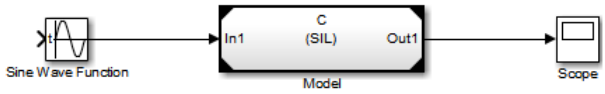
With software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations, you can use the following approaches to numerical equivalence testing:

- Test code from all components together. See "Test Top-Model Code" on page 88-12.
- Test new components separately (before testing code from all components). See "Test Referenced Model Code" on page 88-12 and "Test Subsystem Code" on page 88-12.

For some forms of testing, you require a test harness model. The test harness model:

- Generates test vectors or stimulus inputs that feed the block under test.
- Makes it possible for you to observe or capture output from the block.

The following example shows a simple test harness model.



The block under test is a Model block. The Sine Wave block generates the input for the Model block. Through the Scope block, you can observe the output from the Model block. For information about creating and using test harnesses, see "Create or Import Test Harnesses and Select Properties" (Simulink Test) and "Code Generation Verification Workflow with Simulink Test" (Simulink Test).

Test Top-Model Code

To test code generated from the top-model components together (`A`, `B`, `C`, and `D`), you can use top-model SIL/PIL or Model block SIL/PIL.

- Top-model SIL/PIL:
  1. Create test vectors or stimulus inputs in the MATLAB workspace.
  2. Run the top model in normal, SIL, and PIL simulation modes. The software loads the test vectors or stimulus inputs from the MATLAB workspace.
  3. For each simulation mode, observe or capture outputs.
  4. Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.
- Model block SIL/PIL:
  1. Create a Model block that contains the top-model components.
  2. Insert the Model block in a simulation model, for example, your test harness model.
  3. Run simulations, switching the Model block between normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code interface** Model block parameter to `Top model`.
  4. Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.

For more information, see “Simulation with Top Model” on page 88-15 and “SIL/PIL Manager Verification Workflow” on page 88-76.

Test Referenced Model Code

To test code generated from the component `C` as part of a model reference hierarchy, use the Model block SIL/PIL approach:

- Insert the Model block `C` in a simulation model, for example, your test harness model.
- Run simulations, switching the Model block between normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code interface** Model block parameter to `Model reference`.
- Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.

For more information, see “Simulation with Model Blocks” on page 88-16 and “SIL/PIL Manager Verification Workflow” on page 88-76.

Test Subsystem Code

To test code that is generated from the subsystem `D`, use one of these workflows.

Workflow	Description
Simulink Test harness with SIL/PIL Manager	Perform unit tests on subsystem code that is part of the code generated from the parent model. **1** Generate code for the model that contains the subsystem by running `slbuild(model)` or `slbuild(model, 'RTWModelReferenceTargetOnly')`. **2** In the model, right-click the subsystem and then create a test harness for the subsystem. **3** Open the SIL/PIL Manager. If required, enable code coverage analysis. Then run back-to-back model and SIL or PIL simulations. **4** If required, export an equivalence test case. For more information, see “Unit Test Subsystem Code with SIL/PIL Manager” on page 88-31. If the subsystem is not supported by this workflow, use the SIL or PIL block workflow. For more information about unsupported subsystems and other limitations, see “Atomic Subsystem Workflow Limitations” on page 88-92.
SIL or PIL block	Generate and test new standalone code from the subsystem. **1** Insert the subsystem in a simulation model, for example, your test harness model. **2** Run a normal mode simulation, capturing the outputs. **3** Create a SIL or PIL block from the subsystem. **4** In the model, replace the subsystem with the SIL or PIL block. **5** Run a simulation of the model, capturing the outputs. **6** Verify numerical equivalence by comparing normal mode subsystem outputs against SIL or PIL block outputs. For more information, see “SIL or PIL Block Simulation” on page 88-18.

Summary

Component From Which Code Is Generated	Simulation Uses	Steps	Generated Code Interface	Test Signal Source
Top model	Top-model SIL/PIL	In SIL/PIL Manager: **1** Select **SIL/PIL Simulation Only** mode. **2** In **System Under Test**, select `Top model`.	Standalone	MATLAB workspace

Component From Which Code Is Generated	Simulation Uses	Steps	Generated Code Interface	Test Signal Source
Model referenced by Model block	Model block SIL/PIL	1 In SIL/PIL Manager: a Select **SIL/PIL Simulation Only** mode. b In **System Under Test**, select `Model blocks in SIL/PIL mode`. 2 In Model blocks, set **Simulation mode** to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)`.	Determined by Model block parameter **Code interface** –– standalone or model reference.	Simulation model, for example, test harness model
Subsystem	Simulink Test harness and SIL/PIL Manager	1 Generate code for model that contains the atomic subsystem. 2 Create a test harness for the subsystem. 3 From SIL/PIL Manager, run back-to-back model and SIL or PIL simulations. 4 If required, export equivalence test case.	Determined by generated parent model code –– standalone or model reference.	Simulink Test harness
Subsystem	SIL or PIL block	Manual block substitution	Standalone	Simulation model, for example, test harness model.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Configure and Run SIL Simulation” on page 88-15
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “Test Generated Code with SIL and PIL Simulations” on page 88-94
- “Code Interfaces for SIL and PIL” on page 88-4

Configure and Run SIL Simulation

In this section...

"Simulation with Top Model" on page 88-15
"Simulation with Model Blocks" on page 88-16
"Simulation with Subsystem Blocks" on page 88-17
"SIL or PIL Block Simulation" on page 88-18
"Configure Hardware Implementation Settings" on page 88-18
"Log Signals of a Component" on page 88-20
"Prevent Code Changes in Multiple Simulations" on page 88-21
"Speed Up Testing" on page 88-22
"Simulation with Function Calls" on page 88-23

There are three ways of running SIL and PIL simulations. You can use:

- The top model.
- Model blocks.
- SIL and PIL blocks that you create from subsystems.

Simulation with Top Model

To configure and run a top-model SIL or PIL simulation:

1. In the Simulink Editor, open your model.
2. On the **Apps** tab, click **SIL/PIL Manager**.
3. In the **Mode** section, select **SIL/PIL Simulation Only**.
4. In the **Prepare** section, set **System Under Test** to `Top model`.
5. In the **SIL/PIL Mode** field, select either `Software-in-the-Loop (SIL)` or `Processor-in-the-Loop (PIL)`. The option supports only ERT, GRT, or AUTOSAR system target files. See "Model Configuration Parameters: Code Generation" and "Configure AUTOSAR Code Generation" (AUTOSAR Blockset) for configuration information.
6. To monitor component signals and state data and compare values from the model and SIL or PIL simulations:
   a. For each signal that you want to log:
      i. In the Simulink Editor, select the signal.
      ii. On the **SIL/PIL** tab, click **Monitor Signals**. From the gallery, select these options:
         - **Log Selected Signals**
         - **Make Selected Signals Testpoints**
         - **Signal Logging**
   b. To log state data, from the **Monitor Signals** gallery, select **State Logging**.
   c. If the **Language** configuration parameter is `C++`, in the **Code Mappings** editor, set the **Data Visibility** for the signals, states, and internal data model element categories to `public`.

7. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. In the Configuration Parameters dialog box, select the **Enable portable word sizes** check box.
8. If required, configure:
   - Code coverage.
   - Code execution profiling on page 81-2.
   - Creation of code generation report and static code metrics on page 88-74.
9. In the **Run** section, in the **Stop Time** field, specify simulation time.
10. Click **Run SIL/PIL**.
11. At the end of the simulation, in the **Results** section, click **Data Inspector** to view simulation results.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation with the command `sim(model)`. The software supports the `sim` command option `SrcWorkspace` for the value `'base'`.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

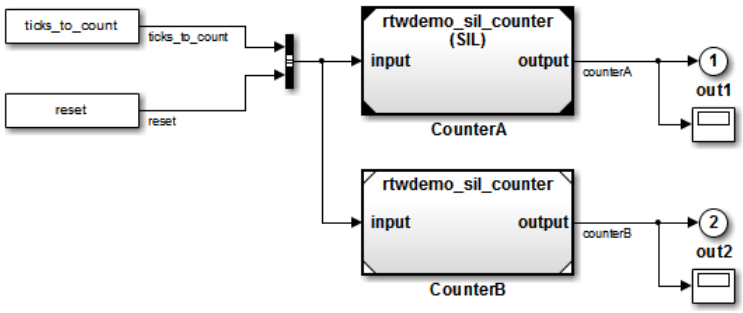
Simulation with Model Blocks

To configure a Model block for a SIL or PIL simulation:

1. Open your model, for example, `SILModelBlock`:

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILModelBlock.slx')
   ```

2. Right-click your Model block, for example, `Counter A`. In the context menu, select **Block Parameters (ModelReference)**, which opens the Function Block Parameters dialog box.
3. From the **Simulation Mode** drop-down list, select the required mode, for example, `Software-in-the-loop (SIL)`.
4. From the **Code interface** drop-down list, specify the code that you want to test, for example, `Model reference`.
5. Click **OK**. The software displays the simulation mode as a block label.



If you select `Top model`, the software displays the block label `(SIL: Top)`.

6 If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. In the Configuration Parameters dialog box, select the **Enable portable word sizes** check box.

7 On the **Apps** tab, click **SIL/PIL Manager**.

8 In the **Mode** section, select **SIL/PIL Simulation Only**.

9 In the **Prepare** section, set **System Under Test** to `Model blocks in SIL/PIL mode`.

10 In the **Top Model Mode** field, select either `Normal` or `Accelerator`.

11 If required, configure:

- “Code Coverage”.
- Code execution profiling on page 81-2 for your Model block, by configuring execution profiling for the top model.
- Creation of code generation report and static code metrics on page 88-74.

12 In the **Run** section:

- In the **Stop Time** field, specify simulation time.
- If you want to avoid rebuilding code after the initial build, click **Fast Restart**. See “Prevent Code Changes in Multiple Simulations” on page 88-28.
- Click **Run SIL/PIL**.

13 At the end of the simulation, in the **Results** section, click Data Inspector to view simulation results.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Simulation with Subsystem Blocks

You can use one of these workflows:

- Simulink Test harness with the SIL/PIL Manager—If you have a model that contains subsystems, you can use Simulink Test and the SIL/PIL Manager to perform unit tests on code generated from

the subsystems. This workflow tests the generated subsystem code as part of the code generated from the parent model. For workflow details, see “Unit Test Subsystem Code with SIL/PIL Manager” on page 88-31.

If this workflow does not support a subsystem, use the SIL or PIL block workflow as an alternative.

- SIL or PIL block—Create a SIL or PIL block from the subsystem, and then run the block in an environment or test harness model that supplies test vectors or stimulus input. This workflow generates and tests new standalone code from the subsystem. For more information, see “SIL or PIL Block Simulation” on page 88-27.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

SIL or PIL Block Simulation

To create a SIL or PIL block from a subsystem and use this block to test the code generated from the subsystem:

1. From the **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced Parameters** > **Create block** drop-down list, select either `SIL` or `PIL`.
2. If required, configure code execution profiling.
3. Click **OK**.
4. In your model window, right-click the subsystem that you want to simulate.
5. Select **C/C++ Code** > **Build This Subsystem**, which starts the subsystem build process that creates a SIL or PIL block for the generated subsystem code.
6. Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.
7. Run simulations with the environment or test harness model.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot create a SIL or PIL block if you do one of the following:

- Disable the `CreateSILPILBlock` property.
- Select a code coverage tool.

Create block appears dimmed.

Configure Hardware Implementation Settings

For a SIL simulation, you must configure hardware implementation settings, which enables generated code compilation for your development computer. These settings can differ from the hardware implementation settings that you use when building the model for your production hardware. Use one of these approaches.

Approach	Details
Portable word sizes	Switch between SIL and PIL modes without regenerating code. You use the same generated source code files for the SIL simulation on your development computer and for production deployment on the target platform. To configure a model to use portable word sizes, set: • `ProdEqTarget` to `'on'`. • `PortableWordSizes` to `'on'`.

	When you generate code for a model with portable word sizes specified, the code generator conditionalizes data type definitions in `rtwtypes.h`:
	
	`#ifdef PORTABLE_WORDSIZES /* PORTABLE_WORDSIZES defined */`
	
	…
	
	`#else /* PORTABLE_WORDSIZES not defined */`
	
	…
	
	`#endif /* PORTABLE_WORDSIZES */`
	
	The template makefile that you use to build code for your target must not contain the `PORTABLE_WORDSIZES` definition.
	
	For the template makefile and toolchain approaches to building code, the software specifies `-DPORTABLE_WORDSIZES` for the compiler only for host-based builds.
	
	For information about the template makefile and toolchain approaches to building code, see “Configure Toolchain (ToolchainInfo) or Template Makefile Build Process” on page 59-20.

	Consider the case where your target uses code that your development computer cannot compile. When you switch from the PIL mode to the SIL mode and try to simulate the model, you see compilation errors. You can try to work around this problem by adding the source code files to the `SkipForSil` group in the build information object `RTW.BuildInfo`. The SIL build on the host platform does not compile source files present in the `SkipForSil` group. For information about how you add source code files to a group in the build information object, see: • `addSourceFiles` in the Simulink Coder documentation • “Customize Post-Code-Generation Build Processing” in the Simulink Coder documentation

Approach	Details
	Numerical results can differ between generated code executing in a SIL simulation and generated code executing on the production hardware under one of these conditions: • Your model contains blocks implemented in TLC, for which C integral promotion in expressions can behave differently between the MATLAB host and the production hardware target. Normal and PIL simulation results match, but SIL simulation results can differ. • Your production hardware implements rounding to `Floor` for signed integer division, and divisions in your model use rounding mode `Ceiling`, `Floor`, `Simplest`, or `Zero`. Normal and PIL simulation results match, but SIL simulation results can differ. • The byte ordering for your production hardware is `Big Endian`. Normal and PIL simulation results match, but SIL simulation results can differ. For example, when generated code depends on byte order *and* generated production code is implemented with the aim that its behavior matches normal simulation behavior. • You use custom code with the Stateflow product. In this case, type conversion statements are not inserted into the custom code, which target overflow behavior on the host can require. Normal and PIL simulation results match, but SIL simulation results can differ.
Test hardware	Use this approach only when you want to work around a limitation of portable word sizes. Set: • `PortableWordSizes` to `'off'`. • `ProdEqTarget` to `'off'`. • `TargetHWDeviceType` to `'Custom Processor->MATLAB Host Processor'`.
Production hardware	Use this approach only when the production hardware settings match your development computer architecture. Set: • `PortableWordSizes` to `'off'`. • `ProdEqTarget` to `'on'`. • `ProdHWDeviceType` to match your development computer architecture.

For information about test and production targets, see “Configure Run-Time Environment Options” in the Simulink Coder documentation.

Log Signals of a Component

SIL and PIL component outputs are available for observation and comparison with other simulation mode outputs. If you want to examine an internal signal, you can enable internal signal logging for top-model or Model block SIL or PIL. With signal logging, you can:

- Collect signal logging outputs during SIL/PIL simulations, for example, `logsout`.
- Log the internal signals and the root-level outputs of a SIL/PIL component.

- Manage the SIL/PIL signal logging settings with the Simulink Signal Logging Selector.
- Use the Simulation Data Inspector to:
  - Observe streamed signals during normal, SIL, and PIL simulations.
  - Compare logged signals from normal, SIL, and PIL simulations.

To enable signal logging to the MATLAB workspace and signal streaming to the Simulation Data Inspector during SIL or PIL simulations:

1. For each signal that you want to monitor:
   a. In the Simulink Editor, select the signal.
   b. On the **SIL/PIL** tab, click **Monitor Signals**. From the gallery, select these options:
      - **Log Selected Signals**
      - **Make Selected Signals Testpoints**
      - **Signal Logging**
2. If the **Language** configuration parameter is `C++`, in the **Code Mappings** editor, set the **Data Visibility** for the signals, states, and internal data model element categories to `public`.

You can use other methods to examine internal signals of the SIL or PIL component:

- Manually route the signal to the top level.
- Use global data stores to access internal signals:
  1. Inside the component, connect a Data Store Write block to the required signal.
  2. Outside the component, use a Data Store Read block to access the signal value.
- Use MAT-file logging. Note that:
  - MAT-file logging does not support signal logging. If signal logging is enabled, `logsout` is generated but not stored in the MAT-file.
  - For PIL, the target environment must support MAT-file logging.

For more information, see:

- “Configure Signals as Test Points”
- “Save Signal Data Using Signal Logging”
- “Local and Global Data Stores”
- “Global Data Store Example”
- “Log Program Execution Results”

Prevent Code Changes in Multiple Simulations

Use Model block SIL/PIL or the SIL/PIL block with fast restart when you want to run multiple SIL or PIL simulations with:

- Varying test vectors (parameter sets and input data).
- Unchanged generated code, that is, none of the simulations regenerate or rebuild code after the initial build. For example, you want to avoid the incremental code generation that an initial value change can trigger.

For Model block SIL/PIL, you can also use one of these methods:

- In your test harness model, in the Configuration Parameters dialog box, set **Rebuild** to `Never`. If the Model block **Code interface** parameter is `Model reference`, the software does not rebuild the referenced model code. (If the **Code interface** parameter is `Top model`, the software ignores the **Rebuild** setting.)
- Create a protected model and generate source or binary code. Then, insert the protected model in your test harness model. With this method, you can verify top-model code (with the standalone code interface) or model reference code.

For the alternative methods of running Model block SIL/PIL, the following table summarizes code generation behavior after the initial build.

SIL and PIL Approach		Code Generation Behavior After Initial Build
Model block	**Rebuild** configuration parameter of test harness model set to `Never`.	**1** Component (algorithm) code from initial build is not regenerated. **2** Component code makefile is not called. **3** SIL/PIL application files from initial build are not regenerated. **4** SIL/PIL application makefile is called.
Model block (protected model)	Source code from protected model.	You observe the same behavior except for feature 2. In this case, the component code makefile is run. The component code is recompiled and linked to produce new object code.
	Binary code from protected model.	You observe features 1–4.

For more information, see:

- “Model Configuration Parameters: Model Referencing”
- “Protect Models to Conceal Contents”

Speed Up Testing

If your model has SIL/PIL blocks or Model blocks in SIL/PIL mode, you can speed up SIL/PIL testing by:

- Running the top-model simulation in accelerator mode. This mode accelerates the simulation of model components that are not in SIL or PIL mode.
- Turning on fast restart. After the first simulation, you can tune parameters and rerun simulations without model recompilation. The SIL/PIL Manager provides a **Fast Restart** button.

Note The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see “What Is Acceleration?”.

Simulation with Function Calls

Use the Simulink Function block and Simulink Function block when you want to:

- Generate code that makes a function-call to external code, for example, driver or legacy code.
- Provide a subsystem that behaves like the external code in normal, SIL, or PIL simulations.

The example in “Configure Calls to AUTOSAR NVRAM Manager Service” (AUTOSAR Blockset) shows how you can configure client calls to Basic Software (BSW) NVRAM Manager (NvM) service interfaces from your AUTOSAR software component. In a simulation, Simulink implements the BSW NvM calls through Simulink Function and preconfigured Function Caller blocks. For the final system, you link function-call stubs with external BSW function code that runs in the AUTOSAR Runtime Environment (RTE).

For more information, see “Simulink Function Blocks and Code Generation”.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Test Generated Code with SIL and PIL Simulations” on page 88-94
- “Debug Generated Code During SIL or PIL Simulation” on page 88-35
- “View SIL and PIL Files in Code Generation Report” on page 88-74
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “Run Simulations Programmatically”
- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 88-33
- “SIL and PIL Limitations” on page 88-83
- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “Create Execution-Time Profile for Generated Code” on page 81-2

Configure and Run PIL Simulation

In this section...

"Simulation with Top Model" on page 88-24
"Simulation with Model Blocks" on page 88-25
"Simulation with Subsystem Blocks" on page 88-26
"SIL or PIL Block Simulation" on page 88-27
"Log Signals of a Component" on page 88-27
"Prevent Code Changes in Multiple Simulations" on page 88-28
"Speed Up Testing" on page 88-29
"Simulation with Function Calls" on page 88-30

There are three ways of running SIL and PIL simulations. You can use:

- The top model.
- Model blocks.
- SIL and PIL blocks that you create from subsystems.

Simulation with Top Model

To configure and run a top-model SIL or PIL simulation:

1. In the Simulink Editor, open your model.
2. On the **Apps** tab, click **SIL/PIL Manager**.
3. In the **Mode** section, select **SIL/PIL Simulation Only**.
4. In the **Prepare** section, set **System Under Test** to `Top model`.
5. In the **SIL/PIL Mode** field, select either `Software-in-the-Loop (SIL)` or `Processor-in-the-Loop (PIL)`. The option supports only ERT, GRT, or AUTOSAR system target files. See "Model Configuration Parameters: Code Generation" and "Configure AUTOSAR Code Generation" (AUTOSAR Blockset) for configuration information.
6. To monitor component signals and state data and compare values from the model and SIL or PIL simulations:
   a. For each signal that you want to log:
      i. In the Simulink Editor, select the signal.
      ii. On the **SIL/PIL** tab, click **Monitor Signals**. From the gallery, select these options:
         - **Log Selected Signals**
         - **Make Selected Signals Testpoints**
         - **Signal Logging**
   b. To log state data, from the **Monitor Signals** gallery, select **State Logging**.
   c. If the **Language** configuration parameter is `C++`, in the **Code Mappings** editor, set the **Data Visibility** for the signals, states, and internal data model element categories to `public`.

7. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. In the Configuration Parameters dialog box, select the **Enable portable word sizes** check box.
8. If required, configure:
   - Code coverage.
   - Code execution profiling on page 81-2.
   - Creation of code generation report and static code metrics on page 88-74.
9. In the **Run** section, in the **Stop Time** field, specify simulation time.
10. Click **Run SIL/PIL**.
11. At the end of the simulation, in the **Results** section, click **Data Inspector** to view simulation results.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation with the command `sim(model)`. The software supports the `sim` command option `SrcWorkspace` for the value `'base'`.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

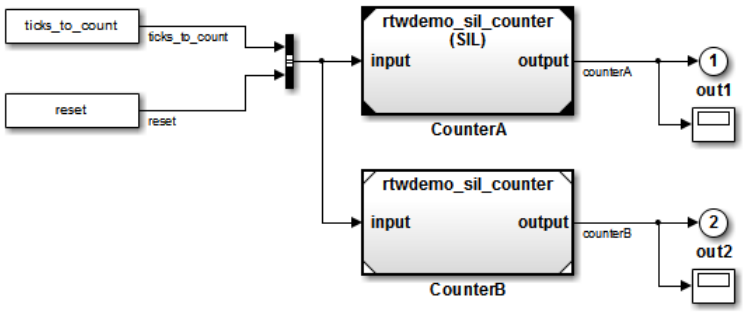
Simulation with Model Blocks

To configure a Model block for a SIL or PIL simulation:

1. Open your model, for example, `SILModelBlock`:

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILModelBlock.slx')
   ```

2. Right-click your Model block, for example, `Counter A`. In the context menu, select **Block Parameters (ModelReference)**, which opens the Function Block Parameters dialog box.
3. From the **Simulation Mode** drop-down list, select the required mode, for example, `Software-in-the-loop (SIL)`.
4. From the **Code interface** drop-down list, specify the code that you want to test, for example, `Model reference`.
5. Click **OK**. The software displays the simulation mode as a block label.



If you select `Top model`, the software displays the block label `(SIL: Top)`.

6 If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. In the Configuration Parameters dialog box, select the **Enable portable word sizes** check box.

7 On the **Apps** tab, click **SIL/PIL Manager**.

8 In the **Mode** section, select **SIL/PIL Simulation Only**.

9 In the **Prepare** section, set **System Under Test** to `Model blocks in SIL/PIL mode`.

10 In the **Top Model Mode** field, select either `Normal` or `Accelerator`.

11 If required, configure:

- "Code Coverage".
- Code execution profiling on page 81-2 for your Model block, by configuring execution profiling for the top model.
- Creation of code generation report and static code metrics on page 88-74.

12 In the **Run** section:

- In the **Stop Time** field, specify simulation time.
- If you want to avoid rebuilding code after the initial build, click **Fast Restart**. See "Prevent Code Changes in Multiple Simulations" on page 88-28.
- Click **Run SIL/PIL**.

13 At the end of the simulation, in the **Results** section, click Data Inspector to view simulation results.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Simulation with Subsystem Blocks

You can use one of these workflows:

- Simulink Test harness with the SIL/PIL Manager—If you have a model that contains subsystems, you can use Simulink Test and the SIL/PIL Manager to perform unit tests on code generated from

the subsystems. This workflow tests the generated subsystem code as part of the code generated from the parent model. For workflow details, see “Unit Test Subsystem Code with SIL/PIL Manager” on page 88-31.

If this workflow does not support a subsystem, use the SIL or PIL block workflow as an alternative.

- SIL or PIL block—Create a SIL or PIL block from the subsystem, and then run the block in an environment or test harness model that supplies test vectors or stimulus input. This workflow generates and tests new standalone code from the subsystem. For more information, see “SIL or PIL Block Simulation” on page 88-27.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

SIL or PIL Block Simulation

To create a SIL or PIL block from a subsystem and use this block to test the code generated from the subsystem:

1. From the **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced Parameters** > **Create block** drop-down list, select either `SIL` or `PIL`.
2. If required, configure code execution profiling.
3. Click **OK**.
4. In your model window, right-click the subsystem that you want to simulate.
5. Select **C/C++ Code** > **Build This Subsystem**, which starts the subsystem build process that creates a SIL or PIL block for the generated subsystem code.
6. Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.
7. Run simulations with the environment or test harness model.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL simulation. To allow the simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot create a SIL or PIL block if you do one of the following:

- Disable the `CreateSILPILBlock` property.
- Select a code coverage tool.

Create block appears dimmed.

Log Signals of a Component

SIL and PIL component outputs are available for observation and comparison with other simulation mode outputs. If you want to examine an internal signal, you can enable internal signal logging for top-model or Model block SIL or PIL. With signal logging, you can:

- Collect signal logging outputs during SIL/PIL simulations, for example, `logsout`.

- Log the internal signals and the root-level outputs of a SIL/PIL component.
- Manage the SIL/PIL signal logging settings with the Simulink Signal Logging Selector.
- Use the Simulation Data Inspector to:
  - Observe streamed signals during normal, SIL, and PIL simulations.
  - Compare logged signals from normal, SIL, and PIL simulations.

To enable signal logging to the MATLAB workspace and signal streaming to the Simulation Data Inspector during SIL or PIL simulations:

1. For each signal that you want to monitor:
   a. In the Simulink Editor, select the signal.
   b. On the **SIL/PIL** tab, click **Monitor Signals**. From the gallery, select these options:
      - **Log Selected Signals**
      - **Make Selected Signals Testpoints**
      - **Signal Logging**
2. If the **Language** configuration parameter is `C++`, in the **Code Mappings** editor, set the **Data Visibility** for the signals, states, and internal data model element categories to `public`.

You can use other methods to examine internal signals of the SIL or PIL component:

- Manually route the signal to the top level.
- Use global data stores to access internal signals:
  1. Inside the component, connect a Data Store Write block to the required signal.
  2. Outside the component, use a Data Store Read block to access the signal value.
- Use MAT-file logging. Note that:
  - MAT-file logging does not support signal logging. If signal logging is enabled, `logsout` is generated but not stored in the MAT-file.
  - For PIL, the target environment must support MAT-file logging.

For more information, see:

- “Configure Signals as Test Points”
- “Save Signal Data Using Signal Logging”
- “Local and Global Data Stores”
- “Global Data Store Example”
- “Log Program Execution Results”

Prevent Code Changes in Multiple Simulations

Use Model block SIL/PIL or the SIL/PIL block with fast restart when you want to run multiple SIL or PIL simulations with:

- Varying test vectors (parameter sets and input data).

- Unchanged generated code, that is, none of the simulations regenerate or rebuild code after the initial build. For example, you want to avoid the incremental code generation that an initial value change can trigger.

For Model block SIL/PIL, you can also use one of these methods:

- In your test harness model, in the Configuration Parameters dialog box, set **Rebuild** to `Never`. If the Model block **Code interface** parameter is `Model reference`, the software does not rebuild the referenced model code. (If the **Code interface** parameter is `Top model`, the software ignores the **Rebuild** setting.)
- Create a protected model and generate source or binary code. Then, insert the protected model in your test harness model. With this method, you can verify top-model code (with the standalone code interface) or model reference code.

For the alternative methods of running Model block SIL/PIL, the following table summarizes code generation behavior after the initial build.

SIL and PIL Approach		Code Generation Behavior After Initial Build
Model block	**Rebuild** configuration parameter of test harness model set to `Never`.	**1** Component (algorithm) code from initial build is not regenerated. **2** Component code makefile is not called. **3** SIL/PIL application files from initial build are not regenerated. **4** SIL/PIL application makefile is called.
Model block (protected model)	Source code from protected model.	You observe the same behavior except for feature 2. In this case, the component code makefile is run. The component code is recompiled and linked to produce new object code.
	Binary code from protected model.	You observe features 1–4.

For more information, see:

- “Model Configuration Parameters: Model Referencing”
- “Protect Models to Conceal Contents”

Speed Up Testing

If your model has SIL/PIL blocks or Model blocks in SIL/PIL mode, you can speed up SIL/PIL testing by:

- Running the top-model simulation in accelerator mode. This mode accelerates the simulation of model components that are not in SIL or PIL mode.

- Turning on fast restart. After the first simulation, you can tune parameters and rerun simulations without model recompilation. The SIL/PIL Manager provides a **Fast Restart** button.

Note The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see “What Is Acceleration?”.

Simulation with Function Calls

Use the Simulink Function block and Simulink Function block when you want to:

- Generate code that makes a function-call to external code, for example, driver or legacy code.
- Provide a subsystem that behaves like the external code in normal, SIL, or PIL simulations.

The example in “Configure Calls to AUTOSAR NVRAM Manager Service” (AUTOSAR Blockset) shows how you can configure client calls to Basic Software (BSW) NVRAM Manager (NvM) service interfaces from your AUTOSAR software component. In a simulation, Simulink implements the BSW NvM calls through Simulink Function and preconfigured Function Caller blocks. For the final system, you link function-call stubs with external BSW function code that runs in the AUTOSAR Runtime Environment (RTE).

For more information, see “Simulink Function Blocks and Code Generation”.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Test Generated Code with SIL and PIL Simulations” on page 88-94
- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “Create Execution-Time Profile for Generated Code” on page 81-2
- “View SIL and PIL Files in Code Generation Report” on page 88-74
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “Run Simulations Programmatically”
- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 88-33
- “SIL and PIL Limitations” on page 88-83

Unit Test Subsystem Code with SIL/PIL Manager

If you have a model that contains subsystems, you can use Simulink Test and the **SIL/PIL Manager** to perform unit tests on code generated from the subsystems. This workflow tests the generated subsystem code as part of the code generated from the parent model.

To perform unit tests on code generated from an atomic subsystem:

1. Open the model that contains the atomic subsystem. For example, in the Command Window, enter:

   ```
   openExample('ecoder/SILPILVerificationExample', ...
               supportingFile='SILBlock.slx')
   ```

2. In a new folder, save the model as `plant_model`.
3. Modify configuration parameter settings that are not supported for this workflow. In the Command Window, enter:

   ```
   set_param('plant_model', ...
             'CreateSILPILBlock','None', ...
             'PreserveStaticInFcnDecls', 'Off');

   % For this example, use these values for block parameters.
   set_param('plant_model/Controller', ...
             'RTWSystemCode','Nonreusable function', ...
             'RTWFileNameOpts', 'Auto');
   ```

   For more information about limitations, see “Atomic Subsystem Workflow Limitations” on page 88-92.
4. Generate code for the model that contains the atomic subsystem. In the Command Window, enter:

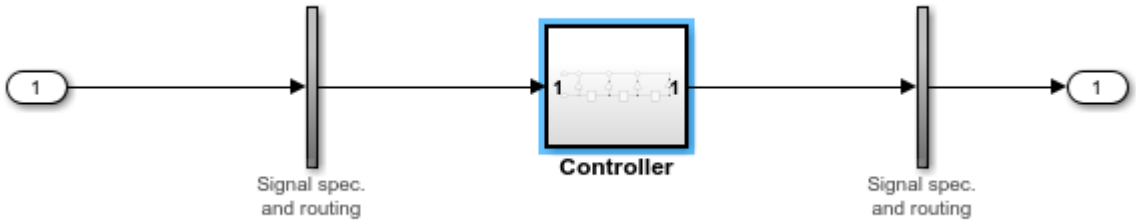
   ```
   slbuild('plant_model')
   ```

5. Create a test harness for the subsystem. Right-click the `Controller` subsystem and select **Test Harness > Create for 'Controller'**.
6. Set the harness properties:

   - **Name** –– `controller_subsystem_harness`
   - **Sources and Sinks**: `Inport` and `Outport`
   - Select **Open harness after creation**.

   Click **OK**.

   Simulink Test creates a test harness for the `Controller` subsystem.



7. Specify format for saving data. In the Command Window, enter:

```
set_param('controller_subsystem_harness', ...
          'SaveFormat','Dataset')
```

8. In the Test Harness window, open the SIL/PIL Manager. On the **Apps** tab, click **SIL/PIL Manager**.
9. If required, configure code coverage.
10. Run back-to-back model and SIL simulations:
    a. In the **Mode** section, select **Automated Verification**.
    b. In the **Run Automated Verification** section, click **Run Verification**.
11. From the **Results** gallery, you can:
    - Use the Simulation Data Inspector to analyze results. For example, compare the `controller_model_output` signals from the model and SIL simulations.
    - Export an equivalence test case.

    For more information, see “Analyze Results and Export Test Cases” on page 88-79.

For more information about limitations, see “Subsystem Block SIL/PIL Limitations” on page 88-92.

See Also

Related Examples

- “Choose a SIL or PIL Approach” on page 88-11
- “Configure and Run SIL Simulation” on page 88-15
- “Configure and Run PIL Simulation” on page 88-24

Simulation Mode Override Behavior in Model Reference Hierarchy

When the top model contains a Model block, the simulation mode of the top model can override the simulation mode of the Model block. The model referenced by the Model block can be a parent model that contains Model blocks at lower levels of the model hierarchy. When you simulate the model hierarchy, the simulation mode of a parent model can override the simulation mode of Model blocks that the parent model contains.

You can specify the simulation mode of a top model to be normal, accelerator, rapid accelerator, SIL, or PIL. With a Model block, you can specify all modes *except* rapid accelerator. This table shows how the software determines the effective simulation mode of a Model block in a model hierarchy.

Simulation Mode Used by Parent Model	**Simulation Mode of Model Block**			
	Normal	**Accelerator**	**SIL**	**PIL**
Normal	Compatible	Compatible	Compatible	Compatible
Accelerator	Overridden	Compatible	Compatible if top model mode is accelerator. Error if parent block mode is accelerator.	Compatible if top model mode is accelerator. Error if parent block mode is accelerator.
Rapid accelerator (top model only)	Overridden	Compatible	Error	Error
SIL	Overridden	Overridden	Compatible	Error
PIL	Overridden	Overridden	Error	Compatible

The different types of behavior are:

- Compatible — The software simulates the Model block in the mode specified for it.
- Error — The simulation produces an error. For example, if a top model specifies rapid accelerator mode but contains a Model block in SIL or PIL mode, then running a simulation produces an error: the rapid accelerator mode cannot override the SIL and PIL mode of Model blocks. This behavior avoids the risk of false positives, that is, the simulation of a model in rapid accelerator mode does not lead to the conclusion that generated source or object code of Model blocks is tested or verified.
- Overridden — The simulation mode of the top model or parent model overrides the simulation mode of the Model block. For example, if a top model or parent model that you configured for a SIL simulation contains a Model block with normal or accelerator simulation mode, then the software simulates the referenced model in SIL mode. The override behavior:
  - Allows a Model block in the model hierarchy to have SIL or PIL mode.
  - Makes lower-level referenced models execute in SIL or PIL mode if you simulate the top model or parent model in SIL or PIL mode. You do not have to switch the simulation mode of every referenced model in the hierarchy.

Multiple Model blocks, starting at the top of a model hierarchy, can execute in SIL or PIL mode. However, if code coverage or code execution profiling is enabled, only one Model block can execute at a time in SIL or PIL mode.

Note You can view your model hierarchy in the Dependency Analyzer. In the Model Instances view, the software displays Model blocks differently to indicate their simulation modes, for example, normal, accelerator, SIL, and PIL. In this view, the software does not indicate the simulation mode of the top model.

See Also

More About

- “What Is Acceleration?”
- “SIL and PIL Simulations” on page 88-2
- “Model Block SIL/PIL Limitations” on page 88-91

Debug Generated Code During SIL or PIL Simulation

In this section...
“SIL Debugging” on page 88-35 “PIL Debugging” on page 88-37

If a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation fails or you notice differences between the outputs of your original functions and the generated code, you can rerun the simulation with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which can help you to understand the cause of the issue.

For a SIL or PIL simulation failure, you can also view information from the standard output and standard error streams in the Diagnostic Viewer. For example:

- Output from `printf` statements in your code.
- Error messages sent to `stderr`.
- Some low-level system messages.

During a SIL simulation, the target application redirects the `stdout` and `stderr` streams. When the application terminates, the Diagnostic Viewer displays the information from the redirected streams. The target application also provides a basic signal handler, which captures the POSIX® signals `SIGFPE`, `SIGILL`, `SIGABRT`, and `SIGSEV`. For this signal handler, the target application includes the file `signal.h`.

SIL Debugging

This table gives information about debugger support.

Operating System	Supported Debuggers
Windows	Microsoft Visual Studio IDE.
	Visual Studio Code with MinGW GDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.
macOS	LLDB.
	Visual Studio Code with LLDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.
Linux	GNU Data Display Debugger (DDD).

Operating System	Supported Debuggers
	Visual Studio Code with GDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.

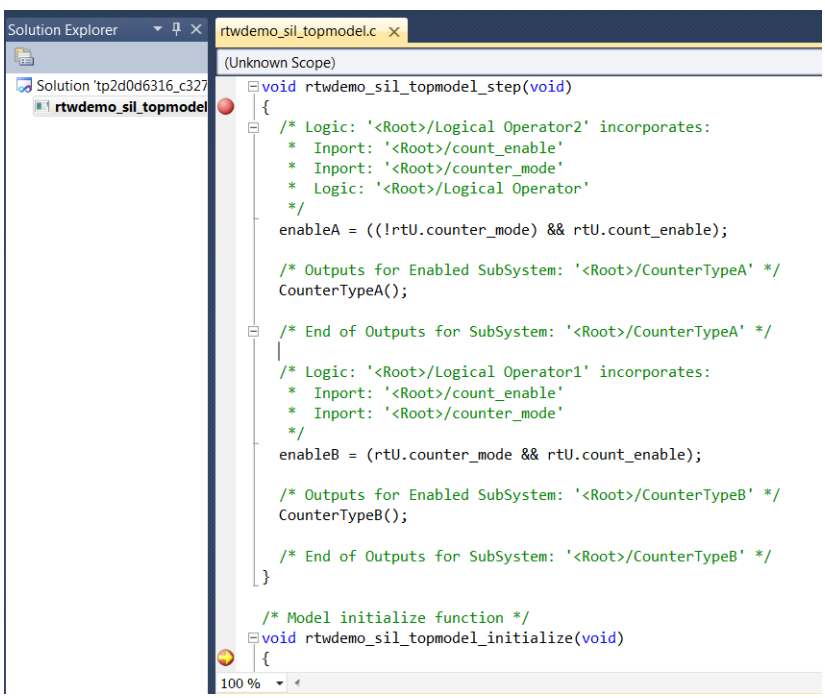
Note You can perform SIL debugging only if the Simulink product family supports your Microsoft Visual C++, Xcode, or GNU GCC compiler. For more information, see supported compilers.

To enable your debugger for a SIL simulation, perform one of these actions:

- On the **SIL/PIL** tab, in the **Settings** gallery, under **SIL/PIL Settings**, click **Debugging**.
- In the Configuration Parameters dialog box, select the **Enable source-level debugging for SIL or PIL** check box.

If your top model has Model blocks, the **Enable source-level debugging for SIL or PIL** parameter for the top model overrides the corresponding parameter for referenced models.

When you run the SIL simulation, your *model*`.c` or *model*`.cpp` file opens, for example, in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the *model*`_initialize` and *model*`_step` functions.



You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

1. Remove all breakpoints.
2. Click the `Continue` button (**F5**).

   The SIL simulation runs to completion and the Microsoft Visual Studio IDE closes.

Note In the Microsoft Visual Studio IDE, if you select **Debug > Stop Debugging**, the SIL simulation times out with the following error message:

```
The timeout of 1 seconds for receiving data from the rtiostream
interface has been exceeded. There are multiple possible causes
for this failure.
...
...
```

PIL Debugging

When you set up PIL connectivity between Simulink and the target hardware, specify a debugger by using `target.ExecutionService` and `target.DebugExecutionTool` objects — see “Support PIL Debugging” on page 88-51 and “DebugExecutionTool Template” on page 88-52.

To enable your debugger for a PIL simulation, perform one of these actions:

- On the **SIL/PIL** tab, in the **Settings** gallery, under **SIL/PIL Settings**, click **Debugging**.
- In the Configuration Parameters dialog box, select the **Enable source-level debugging for SIL or PIL** check box.

If your top model has Model blocks, the **Enable source-level debugging for SIL or PIL** parameter for the top model overrides the corresponding parameter for referenced models.

See Also

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “Set Up PIL Connectivity by Using target Package” on page 88-48

Create PIL Target Connectivity Configuration for Simulink

In this section...

"Target Connectivity Configurations for PIL" on page 88-38
"Create a Target Connectivity API Implementation" on page 88-38
"Register a Connectivity API Implementation" on page 88-40
"Verify Target Connectivity Configuration" on page 88-40
"Target Connectivity API Examples" on page 88-40

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) simulation for your target environments.

Through a target connectivity configuration, you specify:

- A configuration name for a target connectivity API implementation.
- Settings that define the set of compatible Simulink models. For example, the set of models that have a particular system target file, template makefile, and hardware implementation.

A PIL simulation requires a target connectivity API implementation that integrates third-party tools for:

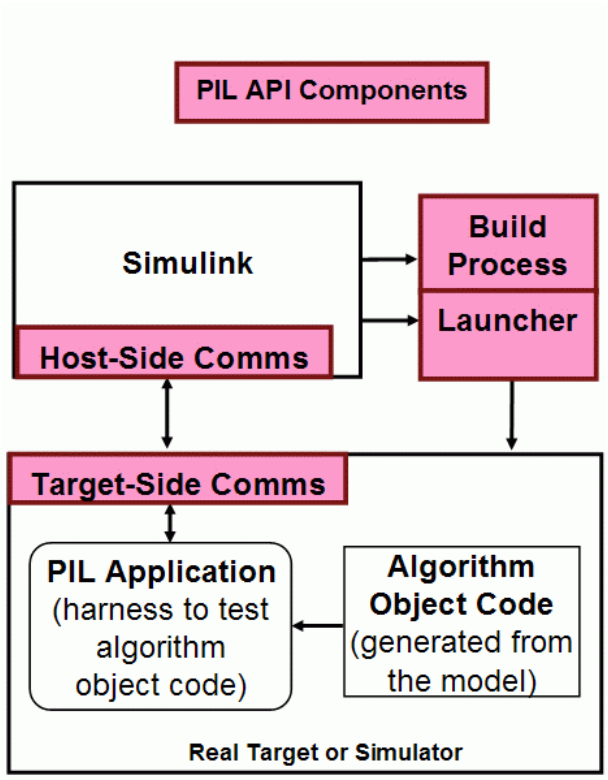
- Cross-compiling generated code, creating the PIL application that runs on the target hardware.
- Downloading, starting, and stopping the application on the target.
- Communicating between Simulink and the target.

You can have many different target connectivity configurations for PIL simulation. Register a connectivity configuration with Simulink by creating an `sl_customization.m` file and placing it on the MATLAB search path.

When you run a PIL simulation, the software determines which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the model under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

Create a Target Connectivity API Implementation

This diagram shows the components of the PIL target connectivity API.



You must provide implementations of the three API components:

- Build API — Specify the Simulink Coder toolchain or template makefile approach for building generated code.
- Launcher API — Control how Simulink starts and stops the PIL executable.
- Communications API — Customize connectivity between Simulink and the PIL target. Embedded Coder provides host-side support for TCP/IP and serial communications, which you can adapt for other protocols.

These steps outline how you create a target connectivity API implementation. The example code shown in the steps is taken from `ConnectivityConfig.m` in “Configure Processor-In-The-Loop (PIL) for a Custom Target” on page 88-113.

1. Create a subclass of `rtw.connectivity.Config`.

   ```
   ConnectivityConfig < rtw.connectivity.Config
   ```

2. In the subclass:

   - Instantiate `rtw.connectivity.MakefileBuilder`, which configures the build process.

     ```
     builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
                                                targetApplicationFramework, ...
                                                exeExtension);
     ```

   - Create a subclass of `rtw.connectivity.Launcher`, which downloads and executes the application using a third-party tool.

     ```
     launcher = mypil.Launcher(componentArgs, builder);
     ```

3. Configure your `rtiostream` API implementation of the host-target communications on page 88-42 channel.

- For the target side, you must provide the driver code for communications, for example, TCP/IP or serial communications. To integrate this code into the build process, create a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
- For the host side, you can use a supplied library for TCP/IP or serial communications. Instantiate `rtw.connectivity.RtIOStreamHostCommunicator`, which loads and initializes the library that you specify.

```
hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, ...
                                                               launcher, ...
                                                               rtiostreamLib);
```

4. If you require execution-time profiling for generated code, create a timer object that provides details of the hardware-specific timer and associated source files. For more information, see "Specify Hardware Timer for Simulink" on page 88-45.
5. If you require stack usage profiling for generated code, specify a driver implementation that obtains stack usage data from the target hardware. The driver must return the value of the stack register. If you do not specify a driver, the PIL simulation tries to use a default generic driver. For more information, see "Implement Driver to Obtain Stack Usage Data During PIL Simulation" on page 81-62.

Note Each time you modify a connectivity implementation, close and reopen the models to refresh them.

Register a Connectivity API Implementation

To register a target connectivity API implementation as a target connectivity configuration in Simulink:

1. Create or update an `sl_customization.m` file. In this file:
   - Create a target connectivity configuration object that specifies, for example, the configuration name for a target connectivity API implementation and compatible models.
   - Invoke `registerTargetInfo`.
2. Add the folder containing `sl_customization.m` to the search path and refresh your customizations.

```
addpath(sl_customization_path);
sl_refresh_customizations;
```

For more information, see `rtw.connectivity.ConfigRegistry`.

Verify Target Connectivity Configuration

To verify your target connectivity configuration early on and independently of your model development and code generation, use the supplied `piltest` function. With the function, you can run a suite of tests. In the tests, the function runs various normal, SIL, and PIL simulations. The function compares results and produces errors if it detects differences between simulation modes.

Target Connectivity API Examples

For step-by-step examples, see:

- “Configure Processor-In-The-Loop (PIL) for a Custom Target” on page 88-113

  This example shows you how to create a custom PIL implementation using the target connectivity APIs. You can examine the code that configures the build process to support PIL, a downloading and execution tool, and a communication channel between host and target. To activate a full host-based PIL configuration, follow the steps in the example.

- “Create a Target Communication Channel for Processor-in-the-Loop (PIL) Simulation” on page 88-108

  This example shows you how to implement a communication channel for use with the Embedded Coder product and your embedded target. This communication channel enables exchange of data between different processes. PIL simulation requires exchange of data between the Simulink software running on your development computer and deployed code executing on target hardware.

  The `rtiostream` interface provides a generic communication channel that you can implement in the form of target connectivity drivers for a range of connection types. The example shows how to configure your own target-side driver for TCP/IP, to operate with the default host-side TCP/IP driver. The default TCP/IP communications allow high-bandwidth communication between host and target, which you can use for transferring data such as video.

  Note If you customize the `rtiostream` TCP/IP implementation for your PIL simulations, you must turn off Nagle's algorithm for the server side of the connection. If Nagle's algorithm is not turned off, your PIL simulations can run at a significantly slower speed. The *`matlabroot/`*`toolbox/coder/rtiostream/src/rtiostreamtcpip/rtiostream_tcpip.c` file shows how you can turn off Nagle's algorithm:

  ```
  /* Disable Nagle's Algorithm*/
  option = 1;
  sockStatus = setsockopt(lFd,IPPROTO_TCP,TCP_NODELAY,(char*)&option,sizeof(option));
  ```

  The code for your custom TCP/IP implementation can require modification.

  The example also shows how to implement custom target connectivity drivers, for example, using serial, CAN, or USB for both host and target sides of the communication channel.

See Also

`piltest` | `rtw.connectivity.Config` | `rtw.connectivity.ConfigRegistry` | `rtw.connectivity.Launcher` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.pil.RtIOStreamApplicationFramework`

Related Examples

- “Host-Target Communication for Simulink PIL Simulation” on page 88-42
- “Specify Hardware Timer for Simulink” on page 88-45
- “Design Subclass Constructors”
- “Processor-in-the-Loop Verification of Simulink Models” (Embedded Coder Support Package for ARM Cortex-A Processors)

Host-Target Communication for Simulink PIL Simulation

In this section...

"Communications rtiostream API" on page 88-42
"Synchronize Host and Target" on page 88-43
"Test an rtiostream Driver" on page 88-43
"Troubleshooting Host-Target Communication" on page 88-43
"Word Addressable Target Hardware" on page 88-44

This section provides information about using the `rtiostream` API for communication between your development computer and target hardware during a processor-in-the-loop (PIL) simulation.

Communications rtiostream API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL simulation requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of target-side and host-side functions that this driver code must implement.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation and a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see "Create a Target Connectivity API Implementation" on page 88-38.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see "Test an rtiostream Driver" on page 88-43 and `rtiostreamtest`.

Synchronize Host and Target

If you use the `rtiostream` API to implement the communications channel, the host and target must be synchronized, which prevents Simulink from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP rtiostream implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator`. This approach works well for connection-oriented TCP/IP `rtiostream` implementations because Simulink automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the Simulink side of the `rtiostream` connection opens without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher` / `rtiostream` implementation that confirms completion of target initialization.

Test an rtiostream Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite can help you to:

- Reduce the time required for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduce the time required for testing custom `rtiostream` drivers.
- Analyze the performance of custom `rtiostream` drivers.

The test suite consists of two parts. One part is an application that runs on the target hardware. The other part runs in MATLAB. For details, see `rtiostreamtest`.

Troubleshooting Host-Target Communication

To display communication I/O information in the Diagnostic Viewer during the PIL simulation, in the Command Window, enter:

```
set_param(model, 'SILPILVerboseOutput', 'on');
```

By default, the configuration parameter `SILPILVerboseOutput` is set to `off`.

If you run a SIL simulation with `SILPILVerboseOutput` set to `on`, the simulation displays I/O information for communication that occurs between Simulink and the target application that runs as a separate process on your development computer.

Word Addressable Target Hardware

In the Configuration Parameters dialog box, when you specify settings for **Device vendor** and **Device type**, you provide PIL simulations with memory addressing information about the target hardware.

For example, consider the case when **Device vendor** is set to `Texas Instruments` and **Device type** is set to `C5000`.

The settings specify that the target hardware uses 16-bit word addresses and big-endian word order. The rtiostream implementation operates with 16-bit words.

On the target hardware, if an `rtIOStream` function specifies a size of 1, then the target hardware must send or receive one 16-bit word (2 bytes). `sizeof(char)` and `sizeof(short)` return 1, which corresponds to one 16-bit word. The `rtIOStream` function expects the byte order within the word to be little-endian, that is, less significant bytes are transmitted before more significant bytes.

`rtiostreamtest` identifies the target hardware and handles data in terms of 16-bit words, for example, a byte value is transferred as a 16-bit word value.

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose` | `rtiostream_wrapper` | `rtiostreamtest` | `rtw.connectivity.RtIOStreamHostCommunicator`

Related Examples

- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38

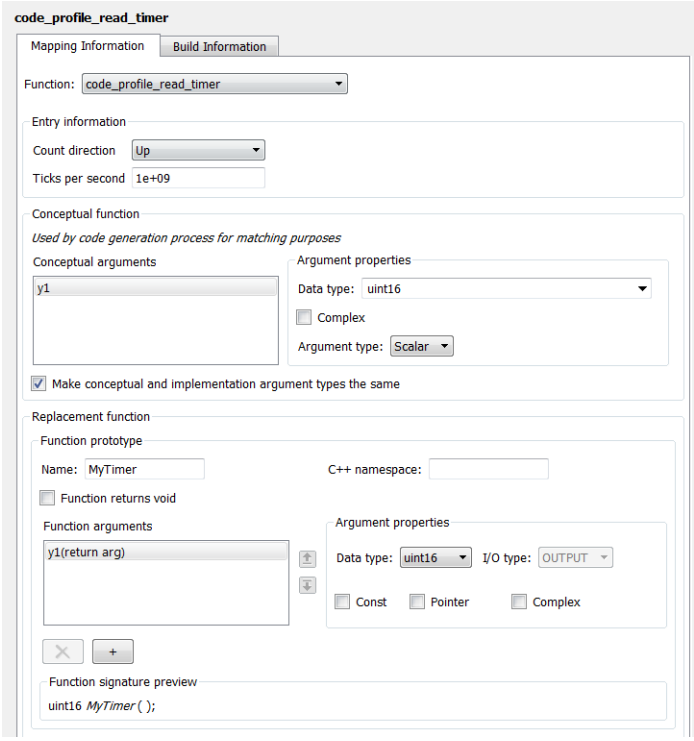
Specify Hardware Timer for Simulink

For processor-in-the-loop (PIL) code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the **Code Replacement Tool** or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

1. Open the Code Replacement Tool. In the Command Window, enter `crtool`.
2. Create a new code replacement table. Select **File** > **New table**.
3. Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry** > **Function**.
4. In the middle view, select the new unnamed function.
5. On the **Mapping Information** pane:
   a. From the **Function** drop-down list, select `code_profile_read_timer`.
   b. Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select `Up`.
   c. In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

      The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.
   d. In the **Conceptual function** section, from the **Data type** list, select `uint8`, `uint16`, `uint32`, or `uint64`.
   e. In the **Function prototype** section, in the **Name** field, specify a replacement function name, for example, `MyTimer`.
   f. Click **Apply**.



g To validate the function entry, click **Validate entry**.

6 On the **Build Information** pane, specify the required build information. See Specify build information in “Define Code Replacement Library Optimizations” on page 72-36.

7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

8 Assuming you save the table as *`MyCrlTable`*`.m`, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCrlTable)
```

See Also

Related Examples

- “Create a Target Connectivity API Implementation” on page 88-38
- “Create Execution-Time Profile for Generated Code” on page 81-2
- “Define Code Replacement Library Optimizations” on page 72-36

More About

- “What Is Code Replacement?” on page 57-2

- “What Is Code Replacement Customization?” on page 72-3

Set Up PIL Connectivity by Using target Package

Before you can run processor-in-the-loop (PIL) simulations, you must set up connectivity between Simulink and your target hardware. The connectivity enables the PIL simulation to:

- Build the target application.
- Download, start, and stop the application on the target hardware.
- Support communication between Simulink and the target hardware.

Use one of these workflows.

Workflow	When to use
`target` package with `rtiostream` API	Use this workflow if your target application has large I/O requirements that need high-bandwidth communication. See “Use rtiostream API for PIL Target Connectivity” on page 88-48. You can specify a debugger. See “Support PIL Debugging” on page 88-51 and “DebugExecutionTool Template” on page 88-52.
`target` package with debugger	Use this workflow if: • You want a single debugger integration to provide PIL connectivity for target hardware systems that are supported by the debugger. This workflow requires little or no target-specific code. • The I/O requirements of the target application are small. For example, unit tests on a component. If you use this workflow for a target application that has large I/O requirements, the execution of the application is slower. See “Use Debugger for PIL Target Connectivity” on page 88-57 and “DebugIOTool Template” on page 88-58.

Use rtiostream API for PIL Target Connectivity

To provide PIL connectivity between Simulink and the target hardware, you can use the `target Package` with the rtiostream API on page 88-42. In this example, your development computer is the target hardware.

1 Create the board description

Create a `target.Board` object that provides MATLAB with a description of your target hardware.

```
hostTarget = target.create('Board', ...
                           'Name', 'Example Intel Board');
```

To support code generation and data I/O in PIL simulations, associate the object with processor, connection, and communication descriptions.

2 **Associate the board with a processor**

To support code generation, associate the board with a `target.Processor` object that contains a language implementation. For this example, create a `target.Processor` object and reuse an existing `target.LanguageImplementation` object. For information about setting up a custom `target.LanguageImplementation` object, see “Register New Hardware Devices” on page 10-13.

```
processor = target.create('Processor', 'Name', 'ExampleProcessor');
processor.LanguageImplementations = target.get('LanguageImplementation', ...
                                               'Intel-x86-64 (Windows64)');
```

Associate the `target.Board` object with the new `target.Processor` object by using the `Processors` property

```
hostTarget.Processors = processor;
```

3 **Specify execution information for the target hardware**

Create an object that contains details about executing the target application. The object describes the tool that is required to run the target application on the target hardware. To capture system commands for starting and stopping the target application, you can use the `HostProcessExecutionTool` or `SystemCommandExecutionTool` class .

```
applicationExecution = target.create('HostProcessExecutionTool');
applicationExecution.Name = 'Windows Application';
```

Create a `Command` object for downloading and running the target application. Assign the string variable `'$(EXE)'` to the `String` property as a place holder for the target application name, which is not known until execution.

```
runCommand = target.create('Command');
runCommand.String = '$(EXE)';
applicationExecution.StartCommand = runCommand;
hostTarget.Tools.ExecutionTools = applicationExecution;
```

4 **Create the communication interface for the target hardware**

Create a `CommunicationInterface` object that provides the target hardware with details of the communication channel and the rtiostream API on page 88-42 implementation.

Use:

- The shipped TCP/IP `rtiostream` implementation source file.
- A `BuildDependencies` object to specify, for the `rtiostream` API, the source files that are compiled with the target application.
- A `MainFunction` object to pass arguments to the target application

```
comms = target.create('CommunicationInterface');
comms.Name = 'Windows TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
                                         'Name', 'x86 RTIOStream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'RTIO Stream');
comms.APIImplementations.BuildDependencies = target.create('BuildDependencies');
comms.APIImplementations.BuildDependencies.SourceFiles = ...
                                           {fullfile('$(MATLAB_ROOT)', ...
                                                     'toolbox', ...
                                                     'coder', ...
                                                     'rtiostream', ...
```

```
                                                        'src', ...
                                                        'rtiostreamtcpip', ...
                                                        'rtiostream_tcpip.c')};
comms.APIImplementations.MainFunction = target.create('MainFunction', ...
                                                       'Name', 'TCP RtIOStream Main');
comms.APIImplementations.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
hostTarget.CommunicationInterfaces = comms;
```

5 **Specify PIL protocol information**

This step is optional. A PIL simulation uses a communication protocol for the transfer of data between Simulink and the target hardware. The `target.PILProtocol` class describes the parameters of the protocol. For this example, you can improve target run-time performance by increasing the I/O buffer size that the protocol uses.

Create a `target.PILProtocol` object and specify I/O buffer size.

```
pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Windows PIL Protocol';
pilProtocol.SendBufferSize = 50000;
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

If you do not perform this step, default `target.PILProtocol` values are used.

6 **Add profiling timer**

You can configure a PIL simulation to produce execution-time profiles for the generated code. To support code execution profiling, you must create a timer object that describes the retrieval of current time from generated code running on the target hardware. The timer object description must include a description of the function that retrieves time and its implementation.

This example uses a C function, `timestamp_x86`, which returns the current time as a `uint64` data type.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

Capture the function in an API object.

```
timerApi = target.create('API');
timerApi.Functions = timerSignature;
timerApi.Language = target.Language.C;
timerApi.Name = 'Windows Timer API';
```

Capture the dependencies of the function, that is, the source and header files that are required to run the function.

```
timerDependencies = target.create('BuildDependencies');
timerDependencies.IncludeFiles = {'host_timer_x86.h'};
timerDependencies.IncludePaths = {'$(MATLAB_ROOT)/toolbox/coder/profile/src'};
timerDependencies.SourceFiles = {'host_timer_x86.c'};
```

Create an object that combines the API and dependencies.

```
timerImplementation = target.create('APIImplementation');
timerImplementation.API = timerApi;
timerImplementation.BuildDependencies = timerDependencies;
timerImplementation.Name = 'Windows Timer Implementation';
```

Create the timer object and associate it with the timer information.

```
timer = target.create('Timer');
timer.APIImplementation = timerImplementation;
timer.Name = 'Windows Timer';
```

Assign the timer to the processor object.

```
processor.Timers = timer;
```

7 Specify connection between development computer and target hardware

The previous steps created target hardware support for communications and running the target application. Now, set up a connection between your development computer and the target hardware by creating a `TargetConnection` object. Specify:

- The communication channel, which is the same channel specified in the target hardware communication interface –– see step 4.
- The connection properties.
- The target, which is the board description specified in previous steps.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = 'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

8 Make board and connection objects persist across MATLAB sessions

To register the connectivity in MATLAB, add the target hardware and connection information to an internal database by using the `target.add` function. By default, the information is only available for the current MATLAB session. To make the registration persist across MATLAB sessions, specify the name-value pair `'UserInstall', true`.

```
target.add([hostTarget connection], 'UserInstall', true);
```

9 Specify board for model

You can now specify your development computer as the target hardware for PIL simulations. Before you run a PIL simulation, in the Configuration Parameters dialog box, set **Hardware board** to `Example Intel Board`.

Support PIL Debugging

The workflow in “Use rtiostream API for PIL Target Connectivity” on page 88-48 provides PIL connectivity between Simulink and the target hardware. To also support PIL debugging, implement the following steps after or instead of step 3 (*Specify execution information for the target hardware*):

1 Describe the debugger execution service implementation

Create a `target.ExecutionService` object that references the debugger abstraction interface.

```
debugExecutionTool = target.create('ExecutionService', ...
                                   'Name', 'GDB Launch');
debugExecutionTool.APIImplementation = ...
                                   target.create('APIImplementation', ...
                                   'Name', 'myServiceImplementation');
debugExecutionTool.APIImplementation.BuildDependencies = ...
                                   target.create('MATLABDependencies');
debugExecutionTool.APIImplementation.BuildDependencies.Classes = ...
                                   {'myImplementationClass'};
debugExecutionTool.APIImplementation.API = ...
                                   target.get('API', ...
                                   'DebugExecutionTool');
```

2 **Associate the debugger execution service with the board**

Associate the `target.ExecutionService` object with the `target.Board` object.

```
hostTarget.Tools.DebugTools = debugExecutionTool;
```

DebugExecutionTool Template

This section provides a pseudocode example of `target.DebugExecutionTool` debugger abstraction. Use a `target.DebugExecutionTool` object to:

- Control the lifetime of an application running in the debugger.
- Automate applying breakpoints.

For interactions with the debugger, you can use MATLAB external language interfaces to APIs, which the debugger vendor supplies.

Section Descriptions for `target.DebugExecutionTool` Debugger Abstraction Example

`target.DebugExecutionTool` Object	Comments
`classdef DebugExecutionToolTemplate < target.DebugExecutionTool`	Example implementation of `target.DebugExecutionTool` debugger abstraction.

`properties (Access=private)`
`DebuggerHandle;`
`end`
	Optional.
	
	Add any state required to interact with the debugger or application being run in the debugger as a property of `DebugExecutionTool`. Apply private access so that only this file can access the state. For example, the state can be a handle to the debugger application provided by the debugger external API.

`methods`	Method implementations that perform debugger interactions. Each method returns a `'errFlag'` logical flag. If the method executes, `'errFlag'` is set to `false`. Consider also using the MATLAB error method to report custom error information.

`function this = DebugExecutionToolTemplate()`
 	
`end`
	Optional.
	
	Constructor for the `target.DebugExecutionTool`. The constructor creates an object of the tool allowing its methods to be called from that object.
	
	Perform basic initialization here. For example, setting state for calls to `open`.

target.DebugExecutionTool Object	Comments

`function errFlag = open(this)`
 	
`this.DebuggerHandle = this.callExternalDebugger('open');`
`errFlag = false;`
`end`
	Optional.
	
	Add logic here to open the debugger. Typically, use system commands or the external API provided by the debugger.
	
	If you cannot open the debugger without also loading the application into the debugger, do not implement this method. Use the `'loadApplication'` method to implement the logic.
	
	If you cannot open the debugger without also starting execution of the application, do not implement this method. Use the `'startApplication'` method to implement the logic.

`function errFlag = loadApplication(this)`
 	
`this.callExternalDebugger('load application', ...`
`this.Application, ...`
`this.ApplicationCommandLineArguments);`
 	
`this.callExternalDebugger('set breakpoints', this.Breakpoints);`
 	
`errFlag = false;`
`end`
	Optional.
	
	Load the application into the debugger.
	
	Add logic here to load the application into the debugger. For example, load a project into an IDE debugger or load the application executable file into the debugger harness.
	
	Also add logic to set breakpoints in the debugger to break execution.
	
	The path to the application file is stored in the `Application` property of this class which can you can access by using `this.Application`.
	
	Additional command line arguments for the application are stored in `this.ApplicationCommandLineArguments`.
	
	Breakpoints to set are stored in `this.Breakpoints`.
	
	If you cannot load the application without also starting execution of the application, do not implement this method. Implement the logic in the `'startApplication'` method.

target.DebugExecutionTool Object	Comments

`function errFlag = startApplication(this)`
 	
`this.callExternalDebugger('start application');`
 	
`errFlag = false;`
`end`
	Required.
	
	Start the application execution in the debugger.
	
	Add logic here to start the application execution in the debugger.
	
	The path to the application to load is stored in the `Application` property of this class, which you can access by using `this.Application`.
	
	Additional command line arguments to pass to the application are stored in `this.ApplicationCommandLineArguments`.
	
	Breakpoints to set are stored in `this.Breakpoints`.

`function errFlag = stopApplication(this)`
 	
`this.callExternalDebugger('stop application');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Stop the application execution in the debugger.
	
	Add logic here to stop the application execution in the debugger.
	
	If your `'startApplication'` method opened the debugger, perform the opposite action here, i.e. close the debugger.
	
	If your `'startApplication'` method loaded the application, perform the opposite action here, i.e. unload the application from the debugger.

`function errFlag = unloadApplication(this)`
 	
`this.callExternalDebugger('unload application');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Unloads the application from the debugger.
	
	If you implemented the `'loadApplication'` method, then add logic here to unload the application from the debugger.
	
	If your `'loadApplication'` method opened the debugger, perform the opposite action here, i.e. close the debugger.

`function errFlag = close(this)`
 	
`this.callExternalDebugger('close');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Closes the debugger.
	
	Add logic here to close the debugger. If you implemented the `'open'` method, you must also implement the `'close'` method.

`end`	`end` statement for `methods`.

target.DebugExecutionTool Object	Comments

`methods (Access=private)`
 	
`function output = callExternalDebugger(~, varargin)`
 	
`output = [];`
`fprintf("\nInstructed the debugger to '%s'\n", varargin{1});`
 	
`if nargin > 2`
`disp('Using arguments:');`
`disp(varargin(2:end));`
`end`
`end`
`end`
	Private methods.

`end`	`end` statement for `classdef`.

Pseudocode for `target.DebugExecutionTool` Debugger Abstraction Example

```
classdef DebugExecutionToolTemplate < target.DebugExecutionTool
% DEBUGEXECUTIONTOOLTEMPLATE Example implementation of target.DebugExecutionTool
% debugger abstraction.

%   Copyright 2023 The MathWorks, Inc.

    properties (Access=private)

        % Optional
        % Add any state required to interact with the debugger or
        % application being run in the debugger as a property of
        % DebugExecutionTool. Apply private access so that only this file
        % can access the state.
        %
        % For example, the state could be a handle to the debugger
        % application provided by the debugger external API.
        DebuggerHandle;
    end

    % Method implementations that perform debugger interactions.
    % Each method will return a 'errFlag' logical flag. If the method
    % executes, 'errFlag' is set to false.
    % Consider also using the MATLAB error method to report custom
    % error information.
    methods

        % Constructor for the target.DebugExecutionToolTemplate. The constructor
        % creates an instance of the tool allowing its methods to be called
        % on that instance. Optional.
        function this = DebugExecutionToolTemplate()

            % Basic initialization can be performed here. For example
            % setting state ready for calls to open
        end

        % Open the debugger. Optional.
        function errFlag = open(this)

            % Add logic here to open the debugger. Typically, use system
            % commands or the external API provided by the debugger.
            %
            % If you cannot open the debugger without also loading the
            % application into the debugger, this method should not be
            % implemented and the logic should be performed in the
            % 'loadApplication' method.
            %
            % If you cannot open the debugger without also starting
            % execution of the application, this method should not be
            % implemented and the logic should be performed in the
            % 'startApplication' method.

            this.DebuggerHandle = this.callExternalDebugger('open');
```

```
        errFlag = false;
    end

    % Load the application into the debugger. Optional.
    function errFlag = loadApplication(this)

        % Add logic here to load the application into the debugger.
        % This could be loading a project into an IDE debugger or
        % loading the application executable into the debugger harness.
        %
        % Additionally add logic to set breakpoints in the debugger to
        % break execution.
        %
        % The path to the application to load is stored in the
        % Application property of this class which can be accessed
        % using this.Application in this method.
        %
        % Additional command line arguments to pass to the the
        % application are stored in
        % this.ApplicationCommandLineArguments.
        %
        % Breakpoints to set are stored in this.Breakpoints
        %
        % If you cannot load the application without also starting
        % execution of the application, this method should not be
        % implemented and the logic should be performed in the
        % 'startApplication' method.

        this.callExternalDebugger('load application', ...
                                  this.Application, ....
                                  this.ApplicationCommandLineArguments);

        this.callExternalDebugger('set breakpoints', this.Breakpoints);

        errFlag = false;
    end

    % Start the application execution in the debugger. Required.
    function errFlag = startApplication(this)

        % Add logic here to start the application execution in the
        % debugger.
        %
        % The path to the application to load is stored in the
        % Application property of this class which can be accessed
        % using this.Application in this method.
        %
        % Additional command line arguments to pass to the
        % application are stored in
        % this.ApplicationCommandLineArguments.
        %
        % Breakpoints to set are stored in this.Breakpoints

        this.callExternalDebugger('start application');

        errFlag = false;
    end

    % Stop the application execution in the debugger. Optional.
    function errFlag = stopApplication(this)

        % Add logic here to stop the application execution in the
        % debugger.

        % If your 'startApplication' method opened the debugger,
        % perform the opposite action here, i.e. close the
        % debugger.
        %
        % If your 'startApplication' method loaded the application,
        % perform the opposite action here, i.e. unload the
        % application from the debugger.
```

```
            this.callExternalDebugger('stop application');

            errFlag = false;
        end

        % Unloads the application from the debugger. Optional.
        function errFlag = unloadApplication(this)

            % If you implemented the 'loadApplication' method,
            % then add logic here to unload the application from
            % the debugger.

            % If your 'loadApplication' method opened the debugger,
            % perform the opposite action here, i.e. close the
            % debugger.

            this.callExternalDebugger('unload application');

            errFlag = false;
        end

        % Closes the debugger. Optional.
        function errFlag = close(this)

            % Add logic here to close the debugger. If you implemented the
            % 'open' method, you should implement this method.

            this.callExternalDebugger('close');

            errFlag = false;
        end

    end

    methods (Access=private)

        function output = callExternalDebugger(~, varargin)

            output = [];
            fprintf("\nInstructed the debugger to '%s'\n", varargin{1});

            if nargin > 2
                disp('Using arguments:');
                disp(varargin(2:end));
            end
        end
    end
end
```

Use Debugger for PIL Target Connectivity

To provide PIL connectivity between Simulink and the target hardware, you can use the `target Package` with a debugger, for example, the GNU Debugger.

1 Implement the `target.DebugIOTool` debugger abstraction interface

Implement the interface in a class called `myImplementationClass`. For an example pseudocode implementation, see “DebugIOTool Template” on page 88-58.

```
classdef myImplementationClass < target.DebugIOTool
    properties (Access=private)
    …
    end

    methods
    …
    end

    methods (Access=private)
```

```
    …
    end
end
```

2 **Create the board description**

Create `target.Board` object that provides MATLAB with a description of your target hardware.

```
hostTarget = target.create('Board', ...
                           'Name', 'GDB Debug PIL IO');
```

3 **Associate the board with a processor**

To support code generation, associate the board with a `target.Processor` object that contains a language implementation. For this example, create a `target.Processor` object and reuse an existing `target.LanguageImplementation` object. For information about setting up a custom `target.LanguageImplementation` object, see “Register New Hardware Devices” on page 10-13.

```
hostTarget.Processors = target.get('Processor', ...
                                   'Intel-x86-64 (Linux 64)');
```

4 **Describe the debugger execution service implementation**

Create a `target.ExecutionService` object that references the debugger abstraction interface.

```
matlabExecution = target.create('ExecutionService', ...
                                'Name', 'GDB Launch');
matlabExecution.APIImplementation = ...
                                target.create('APIImplementation', ...
                                'Name', 'myServiceImplementation');
matlabExecution.APIImplementation.BuildDependencies = ...
                                target.create('MATLABDependencies');
matlabExecution.APIImplementation.BuildDependencies.Classes = ...
                                {'myImplementationClass'};
matlabExecution.APIImplementation.API = ...
                                target.get('API', ...
                                'DebugIOTool');
```

5 **Associate debugger execution service with the board**

Associate the `target.ExecutionService` object with the `target.Board` object.

```
hostTarget.Tools.DebugTools = matlabExecution;
```

6 **Make board object persist across MATLAB sessions**

Add the board object to an internal database and make the object persist across MATLAB sessions.

```
target.add(hostTarget, 'UserInstall', true);
```

7 **Specify board for model**

Before you run a PIL simulation, in the Configuration Parameters dialog box, set **Hardware board** to `GDB Debug PIL IO`.

DebugIOTool Template

This section provides a pseudocode example of `target.DebugIOTool` debugger abstraction. Use a `target.DebugIOTool` object to:

- Control the lifetime of an application running in the debugger.

- Automate common debugger actions. For example, applying breakpoints and continuing execution from a paused state.
- Read and write to memory that the application uses.

For interactions with the debugger, you can use MATLAB external language interfaces to APIs, which the debugger vendor supplies.

Section Descriptions for `target.DebugIOTool` Debugger Abstraction Example

`target.DebugIOTool` Object	Comments
`classdef DebugIOToolTemplate < target.DebugIOTool`	Example implementation of `target.DebugIOTool` debugger abstraction.

`properties (Access=private)`
 	
`DebuggerHandle;`
`end`
	Optional.
	
	Add any state required to interact with the debugger or application being run in the debugger as a property of `DebugIOTool`. Apply private access so that only this file can access the state. For example, the state can be a handle to the debugger application provided by the debugger external API.

`methods`	Method implementations that perform debugger interactions. Each method returns a `'errFlag'` logical flag. If the method executes, `'errFlag'` is set to `false`. Consider also using the MATLAB error method to report custom error information.

`function this = DebugIOToolTemplate()`
 	
`end`
	Optional.
	
	Constructor for the `target.DebugIOTool`. The constructor creates an object of the tool allowing its methods to be called from that object.
	
	Perform basic initialization here. For example, setting state for calls to `open`.

`function errFlag = open(this)`
 	
`this.DebuggerHandle = this.callExternalDebugger('open');`
`errFlag = false;`
`end`
	Optional.
	
	Add logic here to open the debugger. Typically, use system commands or the external API provided by the debugger.
	
	If you cannot open the debugger without also loading the application into the debugger, do not implement this method. Use the `'loadApplication'` method to implement the logic.
	
	If you cannot open the debugger without also starting execution of the application, do not implement this method. Use the `'startApplication'` method to implement the logic.

target.DebugIOTool Object	Comments

`function errFlag = loadApplication(this)`
 	
`this.callExternalDebugger('load application', ...`
`this.Application, ...`
`this.ApplicationCommandLineArguments);`
 	
`this.callExternalDebugger('set breakpoints', this.Breakpoints);`
 	
`errFlag = false;`
`end`
	Optional.
	
	Load the application into the debugger.
	
	Add logic here to load the application into the debugger. For example, load a project into an IDE debugger or load the application executable file into the debugger harness.
	
	Also add logic to set breakpoints in the debugger to break execution.
	
	The path to the application file is stored in the `Application` property of this class which can you can access by using `this.Application`.
	
	Additional command line arguments for the application are stored in `this.ApplicationCommandLineArguments`.
	
	Breakpoints to set are stored in `this.Breakpoints`.
	
	If you cannot load the application without also starting execution of the application, do not implement this method. Implement the logic in the `'startApplication'` method.

`function errFlag = startApplication(this)`
 	
`this.callExternalDebugger('start application');`
 	
`errFlag = false;`
`end`
	Required.
	
	Start the application execution in the debugger.
	
	Add logic here to start the application execution in the debugger.
	
	The path to the application to load is stored in the `Application` property of this class, which you can access by using `this.Application`.
	
	Additional command line arguments to pass to the application are stored in `this.ApplicationCommandLineArguments`.
	
	Breakpoints to set are stored in `this.Breakpoints`.

target.DebugIOTool Object	Comments

`function errFlag = stopApplication(this)`
 	
`this.callExternalDebugger('stop application');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Stop the application execution in the debugger.
	
	Add logic here to stop the application execution in the debugger.
	
	If your `'startApplication'` method opened the debugger, perform the opposite action here, i.e. close the debugger.
	
	If your `'startApplication'` method loaded the application, perform the opposite action here, i.e. unload the application from the debugger.

`function errFlag = unloadApplication(this)`
 	
`this.callExternalDebugger('unload application');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Unloads the application from the debugger.
	
	If you implemented the `'loadApplication'` method, then add logic here to unload the application from the debugger.
	
	If your `'loadApplication'` method opened the debugger, perform the opposite action here, i.e. close the debugger.

`function errFlag = close(this)`
 	
`this.callExternalDebugger('close');`
 	
`errFlag = false;`
`end`
	Optional.
	
	Closes the debugger.
	
	Add logic here to close the debugger. If you implemented the `'open'` method, you must also implement the `'close'` method.

`function [atBreakpoint, errFlag] = ...`
`stoppedAtBreakpoint(this, breakpoint)`
 	
`result = this.callExternalDebugger('determine breakpoint hit', ...`
`breakpoint.File, ...`
`breakpoint.Line, ...`
`breakpoint.Function);`
 	
`atBreakpoint = ~isempty(result);`
 	
`errFlag = false;`
`end`
	Required.
	
	Asks if the debugger is currently stopped at the specified breakpoint. Returns `true` if at the specified breakpoint, `false` otherwise.
	
	Add logic to determine if the debugger has broken execution at the breakpoint passed into the function.
	
	The `'breakpoint'` input variable is of type `target.Breakpoint`, which provides the description of the breakpoint. It contains properties `Function`, `File`, and `Line` which describes the location of the breakpoint.
	
	Set the `atBreakpoint` return value to `true` if the debugger has stopped at the breakpoint, or `false` otherwise.

target.DebugIOTool Object	Comments

`function errFlag = continueExecution(this)`
 	
`this.callExternalDebugger('continue execution');`
`errFlag = false;`
`end`
	Required.
	
	Continues execution if application is paused in the debugger.

`function errFlag = write(this, variableName, memoryStream)`
 	
`sizeOfMemoryVariable = uint64(numel(memoryStream));`
`arrayFormat = repmat('%d,', [1 sizeOfMemoryVariable]);`
`arrayFormat(end) = [];`
 	
`this.callExternalDebugger(sprintf(['execute (void*)memcpy(%s, ...`
`'(unsigned char[%d])%s, ...`
`variableName, ...`
`sizeOfMemoryVariable, ...`
`sprintf(arrayFormat, memoryStream), ...`
`sizeOfMemoryVariable]));`
 	
`errFlag = false;`
`end`
	Write the memory stream to target platform memory specified by the variable name.
	
	Add logic to write `memoryStream` data to the variable `variableName`. The function assumes that code execution has reached the breakpoint that requires this action.
	
	`memoryStream` is a vector of `uint64` values. The `unit64` values act as a container for the smallest addressable integer on the target platform, which in most cases is the size of char, typically 8 bits.
	
	The variable represents a pointer to the smallest addressable integer type. For example, write operations on a target platform where `char` represents the smallest addressable integer are equivalent to the `memcpy` call in C code:
	
	`(void*)memcpy(<variableName>,`
	`(unsigned char[<sizeOfMemoryStream>])`
	`{<memoryStream(1)>,<memoryStream(2)>},`
	`<sizeOfMemoryStream>);`
	
	If your debugger allows immediate execution of code when execution is broken, use the `memcpy` approach to write data to the variable.

target.DebugIOTool Object	Comments

`function [memoryStream, errFlag] = read(this, variableName, sizeToRead)`
 	
`memoryStream = this.callExternalDebugger(...`
`sprintf("read (unsigned char [%d])*%s", ...`
`sizeToRead, variableName));`
 	
`memoryStream = uint64(memoryStream);`
 	
`errFlag = false;`
`end`
	Read the memory stream from target platform memory specified by variable name.
	
	Add logic to read from the `variableName` variable on the target platform. The function assumes that code execution has reached the breakpoint that requires this action.
	
	The `memoryStream` return value is a vector of `uint64` values. The `unit64` values act as a container for the smallest addressable integer on the target platform, which in most cases is the size of `char`, typically 8 bits.
	
	The variable represents a pointer to the smallest addressable integer type. Read operations are equivalent to this C code:
	
	`(unsigned char [<sizeToRead>])*<variableName>`
	
	If your debugger allows immediate execution of code when execution is broken, use the C code approach to inspect the value of the buffer and return it to MATLAB.

`end`	`end` statement for `methods`.

`methods (Access=private)`
 	
`function output = callExternalDebugger(~, varargin)`
 	
`output = [];`
`fprintf("\nInstructed the debugger to '%s'\n", varargin{1});`
 	
`if nargin > 2`
`disp('Using arguments:');`
`disp(varargin(2:end));`
`end`
`end`
`end`
	Private methods.

`end`	`end` statement for `classdef`.

Pseudocode for `target.DebugIOTool` Debugger Abstraction Example

```
classdef DebugIOToolTemplate < target.DebugIOTool
% DEBUGIOTOOLTEMPLATE Example implementation of target.DebugIOTool
% debugger abstraction.

%   Copyright 2020 The MathWorks, Inc.

    properties (Access=private)

        % Optional
        % Add any state required to interact with the debugger or
        % application being run in the debugger as a property of
        % DebugIOTool. Apply private access so that only this file
        % can access the state.
        %
        % For example, the state could be a handle to the debugger
        % application provided by the debugger external API.
        DebuggerHandle;
```

```
end

% Method implementations that perform debugger interactions.
% Each method will return a 'errFlag' logical flag. If the method
% executes, 'errFlag' is set to false.
% Consider also using the MATLAB error method to report custom
% error information.
methods

    % Constructor for the target.DebugIOTool. The constructor
    % creates an instance of the tool allowing its methods to be called
    % on that instance. Optional.
    function this = DebugIOToolTemplate()

        % Basic initialization can be performed here. For example
        % setting state ready for calls to open
    end

    % Open the debugger. Optional.
    function errFlag = open(this)

        % Add logic here to open the debugger. Typically, use system
        % commands or the external API provided by the debugger.
        %
        % If you cannot open the debugger without also loading the
        % application into the debugger, this method should not be
        % implemented and the logic should be performed in the
        % 'loadApplication' method.
        %
        % If you cannot open the debugger without also starting
        % execution of the application, this method should not be
        % implemented and the logic should be performed in the
        % 'startApplication' method.

        this.DebuggerHandle = this.callExternalDebugger('open');
        errFlag = false;
    end

    % Load the application into the debugger. Optional.
    function errFlag = loadApplication(this)

        % Add logic here to load the application into the debugger.
        % This could be loading a project into an IDE debugger or
        % loading the application executable into the debugger harness.
        %
        % Additionally add logic to set breakpoints in the debugger to
        % break execution.
        %
        % The path to the application to load is stored in the
        % Application property of this class which can be accessed
        % using this.Application in this method.
        %
        % Additional command line arguments to pass to the the
        % application are stored in
        % this.ApplicationCommandLineArguments.
        %
        % Breakpoints to set are stored in this.Breakpoints
        %
        % If you cannot load the application without also starting
        % execution of the application, this method should not be
        % implemented and the logic should be performed in the
        % 'startApplication' method.

        this.callExternalDebugger('load application', ...
                                  this.Application, ....
                                  this.ApplicationCommandLineArguments);

        this.callExternalDebugger('set breakpoints', this.Breakpoints);

        errFlag = false;
    end
```

```
% Start the application execution in the debugger. Required.
function errFlag = startApplication(this)

    % Add logic here to start the application execution in the
    % debugger.
    %
    % The path to the application to load is stored in the
    % Application property of this class which can be accessed
    % using this.Application in this method.
    %
    % Additional command line arguments to pass to the
    % application are stored in
    % this.ApplicationCommandLineArguments.
    %
    % Breakpoints to set are stored in this.Breakpoints

    this.callExternalDebugger('start application');

    errFlag = false;
end

% Stop the application execution in the debugger. Optional.
function errFlag = stopApplication(this)

    % Add logic here to stop the application execution in the
    % debugger.

    % If your 'startApplication' method opened the debugger,
    % perform the opposite action here, i.e. close the
    % debugger.
    %
    % If your 'startApplication' method loaded the application,
    % perform the opposite action here, i.e. unload the
    % application from the debugger.

    this.callExternalDebugger('stop application');

    errFlag = false;
end

% Unloads the application from the debugger. Optional.
function errFlag = unloadApplication(this)

    % If you implemented the 'loadApplication' method,
    % then add logic here to unload the application from
    % the debugger.

    % If your 'loadApplication' method opened the debugger,
    % perform the opposite action here, i.e. close the
    % debugger.

    this.callExternalDebugger('unload application');

    errFlag = false;
end

% Closes the debugger. Optional.
function errFlag = close(this)

    % Add logic here to close the debugger. If you implemented the
    % 'open' method you should implement this method.

    this.callExternalDebugger('close');

    errFlag = false;
end

% Asks if the debugger is currently stopped at the specified
% breakpoint. Returns true if at the specified breakpoint, false
% otherwise. Required.
function [atBreakpoint, errFlag] = ...
                            stoppedAtBreakpoint(this, breakpoint)
```

```
    % Add logic to determine if the debugger has broken execution
    % at the breakpoint passed into the function.
    %
    % The 'breakpoint' input variable is of type target.Breakpoint
    % which provides the description of the breakpoint. It contains
    % properties Function, File and Line which describes the
    % location of the breakpoint.
    %
    % Set the atBreakpoint return value to true if the debugger is
    % stopped at the breakpoint, or false if not.

    result = this.callExternalDebugger('determine breakpoint hit', ...
                                       breakpoint.File, ...
                                       breakpoint.Line, ...
                                       breakpoint.Function);
    atBreakpoint = ~isempty(result);

    errFlag = false;
end

% Continues execution if application execution is paused in the
% debugger. Required.
function errFlag = continueExecution(this)

    this.callExternalDebugger('continue execution');
    errFlag = false;
end

% Write the specified memory stream to the target platform memory
% specified by the given variable name.
function errFlag = write(this, variableName, memoryStream)

    % Add logic to write the memoryStream data to the variableName
    % variable in the target. This function should assume that the
    % appropriate breakpoint has been hit to perform such an
    % action.
    %
    % memoryStream is a vector of uint64 values to be written to
    % the target platform. The unit64 values act as a container for
    % the smallest addressable integer on the target platform. In
    % most cases this will be a container for the size of char on
    % the target platform, which will commonly be 8-bit.
    %
    % The specified variable will represent a pointer to the
    % smallest addressable integer type.

    % As an example write operations on a target platform where
    % char represents the smallest addressable integer should be
    % equivalent to the following memcpy call in C code:
    %
    % (void*)memcpy(<variableName>,
    %               (unsigned char[<sizeOfMemoryStream>])
    %               {<memoryStream(1)>,<memoryStream(2)>, ...},
    %               <sizeOfMemoryStream>);
    %
    % A lot of debuggers allow for immediate execution of code when
    % execution is broken, if so, attempt to use the above command
    % to write data to the variable like the below example:

    sizeOfMemoryVariable = uint64(numel(memoryStream));
    arrayFormat = repmat('%d,', [1 sizeOfMemoryVariable]);
    arrayFormat(end) = [];

    this.callExternalDebugger(sprintf(['execute (void*)memcpy(%s,' ...
                            '(unsigned char [%d]){%s}, %d)'], ...
                            variableName, ...
                            sizeOfMemoryVariable, ...
                            sprintf(arrayFormat, memoryStream), ...
                            sizeOfMemoryVariable));

    errFlag = false;
```

```
        end

        % read the memory stream from the target platform memory specified
        % by the given variable name.
        function [memoryStream, errFlag] = read(this, ...
                                                variableName, ...
                                                sizeToRead)

            % Add logic to read from the variableName variable on the
            % target. This function should assume that the appropriate
            % breakpoint has been hit to perform such an action.
            %
            % The memoryStream return value should be a vector of uint64
            % values. The unit64 values act as a container for the smallest
            % addressable integer on the target platform. In most cases
            % this will be a container for the size of char on the target
            % platform, which will commonly be 8-bit.
            %
            % The specified variable will represent a pointer to the
            % smallest addressable integer type.

            % Read operations should be equivilant to inspecting the result
            % of the following C code:
            % (unsigned char [<sizeToRead>])*<variableName>
            %
            % A lot of debuggers allow for immediate execution of code when
            % execution is broken, if so, attempt to use the above command
            % to inspect the value of the buffer and return it to MATLAB.

            memoryStream = this.callExternalDebugger(...
                           sprintf("read (unsigned char [%d])*%s", ...
                           sizeToRead, variableName));

            % The memory stream must be returned as a vector of uint64
            % values
            memoryStream = uint64(memoryStream);

            errFlag = false;
        end
    end

    methods (Access=private)

        function output = callExternalDebugger(~, varargin)

            output = [];
            fprintf("\nInstructed the debugger to '%s'\n", varargin{1});

            if nargin > 2
                disp('Using arguments:');
                disp(varargin(2:end));
            end
        end
    end
end
```

See Also

`target Package`

More About

- “SIL and PIL Simulations” on page 88-2
- “Configure and Run PIL Simulation” on page 88-24
- “Debug Generated Code During SIL or PIL Simulation” on page 88-35

Custom Toolchain Directives Required for Code Coverage and Execution Profiling

If you use a custom toolchain for building a processor-in-the-loop (PIL) target application that supports Simulink Coverage analysis or Embedded Coder code execution profiling, in the toolchain definition file, you must provide these compiler directives:

- `CompileFlag` — Specify the flag that C or C++ compiler uses to compile source files without linking. For example, if the compiler in the toolchain is GNU C or C++, set the directive to -c.
- `PreprocessFile` — Specify the flag that C or C++ compiler uses to preprocess the source files without running other compiler stages. For example, if the compiler in the toolchain is GNU C or C ++, set the directive to -E.

This section from a toolchain definition file example shows how you can use the `setDirective` method to provide the directives.

```
tc = coder.make.ToolchainInfo('BuildArtifact','nmake makefile');
tc.Name ='My Toolchain Name';
tc.Platform ='win64';
tc.SupportedVersion ='14';
...
% ------------------------------
% C Compiler
% ------------------------------
tool = tc.getBuildTool('C Compiler');
tool.setName('My C Compiler');
...
tool.setDirective('CompileFlag','-c');
tool.setDirective('PreprocessFile','-E');
...
```

See Also

More About

- “Custom Toolchain Registration”
- “Add Custom Toolchains to MATLAB® Coder™ Build Process”

PIL Simulation Sequence

A processor-in-the-loop (PIL) simulation cross-compiles production source code, and then downloads and runs object code on your target hardware. The connectivity configuration that you create controls the way code is compiled and executed on the target. This table describes the sequence of stages in a PIL simulation.

Stage		Description
1	Start	For top-model PIL, on the Simulink Editor toolbar, you select the `Processor-in-the-Loop (PIL)` mode, and then click the Run button. For Model block PIL, you set the **Simulation mode** parameter of the Model block to `Processor-in-the-loop (PIL)`, and then run a simulation of the harness model that contains the Model block. For the PIL block, you run a simulation of the harness model that contains the PIL block.
2	Validate target connectivity	The software verifies that a target connectivity configuration is registered for PIL. Otherwise, the software produces an error.
3	Generate production source code and build object code for target	The generated source code is identical to the code that is produced when you run the `slbuild` command. • For top-model PIL or Model block PIL with block parameter **Code interface** set to `Top model`, the generated code is identical to the code produced when you run `slbuild('model')`. • For Model block PIL with block parameter **Code interface** set to `Model reference`, the generated code is identical to the code produced when you run `slbuild('model','ModelReferenceCoderTarget')`. The software builds object code for the target by using the template makefile or toolchain that you specify.
4	Create instances of PIL API components	The software instantiates your `rtw.connectivity.Config` class, which creates instances of `rtw.connectivity.MakefileBuilder`, `rtw.connectivity.Launcher`, `rtw.pil.RtIOStreamApplicationFramework`, and `rtw.connectivity.RtIOStreamHostCommunicator`.
5	Generate PIL files	The generated PIL files are in the `pil` folder. At the end of the simulation, use the code generation report to view the files.

Stage		Description
6	Build target application	The software: • Uses your instance of `rtw.connectivity.MakefileBuilder` to build the target application. • Compiles the PIL interface file, `xil_interface.c`, and other PIL files into the target executable file. On a Windows system, for example, this file is called *`modelName`*`.exe`. The object code, including the executable file, is in the `pil` folder. • If configured, produces the code generation report.
7	Start target application	The software uses `rtw.connectivity.Launcher` to start the application on the target.
8	Simulink engine interacts with PIL S-function	The Simulink engine interacts with the PIL S-function in the same way that it interacts with a C S-function. From the host-side, the PIL S-function communicates with the target executable code through `rtIOStream` commands. On the target side, `xil_interface` executes generated code.
9	Stop target application	The software uses `rtw.connectivity.Launcher` to stop the application on the target.
10	End PIL simulation	For top-model PIL, at the end of the simulation, the software destroys the `rtw.connectivity.Config` instance. For Model block PIL and PIL block, the block creates and owns the `rtw.connectivity.Config` instance, which is not destroyed at the end of the simulation. You can rerun the simulation, which now does not require the creation of another `rtw.connectivity.Config` instance. If you want to destroy the instance, close the parent model.

See Also

`rtw.connectivity.Config` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher` | `rtw.pil.RtIOStreamApplicationFramework` | `rtw.connectivity.RtIOStreamHostCommunicator`

Related Examples

- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “View SIL and PIL Files in Code Generation Report” on page 88-74

More About

- “SIL and PIL Simulations” on page 88-2
- “Simulink Engine Interaction with C S-Functions”

Verification of Code Generation Assumptions

At the start of a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, the software verifies some configuration parameter settings with reference to the target hardware.

For example, in the Configuration Parameters dialog box, the **Hardware Implementation**settings specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

The software checks:

- The correctness of settings. For example, the integer bit length in the **Number of bits: int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

The simulation generates a Coder Assumptions page on page 58-13 for the code generation report, which provides a list of:

- Code generation assumptions that are checked
- Expected results for the assumption checks

This table shows when checks occur and outcomes when code generation assumptions are incorrect.

Stage	What is Checked	Outcome
Preprocessor	**Number of bits** per: • **char** (`ProdBitPerChar`) • **short** (`ProdBitPerShort`) • **int** (`ProdBitPerInt`) • **long** (`ProdBitPerLong`) If **Support long long** (`ProdLongLongMode`) is selected, number of bits per `long long` For each data type, the preprocessor check is not performed if the number of bits exceeds these settings for the target C preprocessor: • For signed integer math, `TargetPreprocMaxBitsSint`. • For unsigned integer math, `TargetPreprocMaxBitsUint`.	Error if data type sizes for model and target hardware do not match.

Stage	What is Checked	Outcome
Run-time	**Number of bits** per: • **char** (`ProdBitPerChar`) • **short** (`ProdBitPerShort`) • **int** (`ProdBitPerInt`) • **long** (`ProdBitPerLong`)	Error if data type sizes for model and target hardware do not match.
	If **Support long long** (`ProdLongLongMode`) is selected, number of bits per `long long`	Error if: • Target hardware does not support `long long`. • Data type sizes for model and target hardware do not match.
	Size of: • `float` • `double` You cannot configure these data type sizes through the **Hardware Implementation** pane. The check is performed only if **Support: floating-point numbers** is selected.	Warning if data type sizes for model and target hardware do not match. For `double`, warning is generated if size of target hardware data type is not 32 or 64 bits.
	Number of bits per: • **pointer** (`ProdBitPerPointer`) • **size_t** (`ProdBitPerSizeT`) • **ptrdiff_t** (`ProdBitPerPtrDiffT`)	Error if data type sizes for model and target hardware do not match.
	Signed integer division rounds to (`ProdIntDivRoundTo`) setting	Warning if model parameter setting is `Undefined`. Error if target hardware behavior is undefined and model parameter setting is not `Undefined`. Error if target hardware behavior is defined but settings for model and target hardware do not match.
	Byte ordering (`ProdEndianess`) setting	Warning if setting is `Unspecified`. Otherwise, error if settings for model and target hardware do not match.

Stage	What is Checked	Outcome
	Shift right on a signed integer as arithmetic shift (`ProdShiftRightIntArith`) setting	Error if settings for model and target hardware do not match.
	Remove root level I/O zero initialization (`ZeroExternalMemoryAtStartup`) setting	Warning if `ZeroExternalMemoryAtStartup` is `'off'` and initial values of global variables in the target application are not zero.
	Remove internal data zero initialization (`ZeroInternalMemoryAtStartup`) setting	Warning if `ZeroInternalMemoryAtStartup` is `'off'` and initial values of global variables in the target application are not zero.
	Use dynamic memory allocation for model initialization (`GenerateAllocFcn`) setting	Warning if `GenerateAllocFcn` is `'on'` and initial values of dynamically allocated memory in the target application are not zero.
	Simulation behavior for denormal numbers (`DenormalBehavior`) setting	Error if behavior for model and target hardware does not match.

See Also

“Hardware Implementation Pane”

More About

- “SIL and PIL Simulations” on page 88-2
- “Check Code Generation Assumptions” on page 58-12

View SIL and PIL Files in Code Generation Report

With top-model and Model block SIL and PIL simulations, you can produce a code generation report and static code metrics that cover SIL and PIL files. The information helps you to:

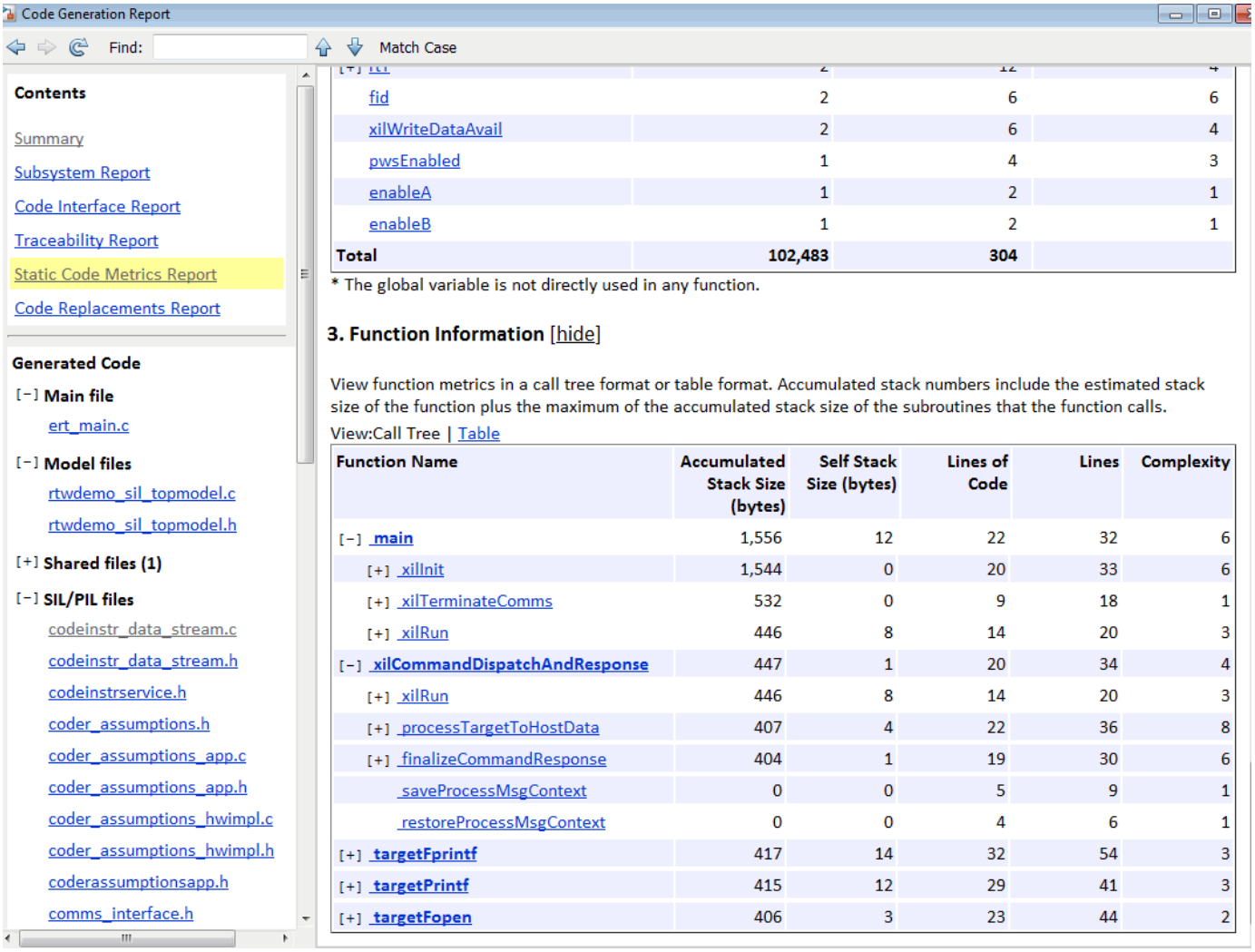
- Understand and review the SIL and PIL testing process.
- See how your registered custom target connectivity files fit into the target application that runs during a SIL or PIL simulation.

This capability is not supported for simulations that you run with the PIL block.

To configure the creation of a code generation report and static code metrics, in the Configuration Parameters dialog box, select the **Create code generation report**, **Open report automatically**, and **Generate static code metrics** check boxes. Then click **OK**.

At the end of the simulation, in the Code Generation Report window:

- To review code metrics, in the **Contents** view, click **Static Code Metrics Report**.
- To review SIL and PIL files, in the **Generated Code** view, expand the **SIL/PIL files** node. For example:
  - To review the S-function that runs on the host, click `*modelName*_sbs.c` or `*modelName*_pbs.c`.
  - To view the SIL or PIL interface that runs on the target, click `xil_interface.c`.



Note Do not use the SIL or PIL files in code development as these files can change over releases. Use supplied APIs for code development.

See Also

Related Examples

- “Static Code Metrics” on page 54-26

More About

- “HTML Code Generation Report Extensions” on page 54-3

SIL/PIL Manager Verification Workflow

The **SIL/PIL Manager** is an app that provides a simplified workflow for verifying generated model code.

Open the App

To open the SIL/PIL Manager in the Simulink Editor, on the **Apps** tab, click **SIL/PIL Manager**. Or, on the **C Code** tab in the Embedded Coder app, select **Verify Code > SIL/PIL Manager**.

Select the Mode

In the **Mode** section, specify the simulations that you want the app to run. You can select:

- **Automated Verification** –– The app runs two simulations back-to-back. The first simulation is a model simulation in normal, accelerator, or rapid accelerator mode. The second simulation is a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. After the second simulation is complete, the app uses the Simulation Data Inspector to compare the results of the simulations.

  For the first simulation, if the model contains a Model block with the **Simulation mode** block parameter set to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)`, at the start of the simulation, the app changes the block parameter to `Normal`. At the end of the simulation, the app reverts the block parameter to the previous value.
- **Simulation Only** –– The app runs a model simulation in normal, accelerator, or rapid accelerator mode.

  If the model contains a Model block with the **Simulation mode** block parameter set to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)`, at the start of the simulation, the app changes the block parameter to `Normal`. At the end of the simulation, the app reverts the block parameter to the previous value.
- **SIL/PIL Simulation Only** –– The app runs a SIL or PIL simulation.

Prepare the Model

In the **Prepare** section:

- Specify the numeric testing approach.
- Specify the main configuration parameters.
- Enable or disable code execution profiling and code coverage analysis.

This table shows how the field settings in the **Prepare** section control simulations that the app runs.

Workflow	Field Settings	Model Simulation	SIL or PIL Simulation
Top model	**System Under Test** — `Top model` **Simulation Mode** — `Normal`, `Accelerator`, or `Rapid Accelerator` **SIL/PIL Mode** — `Software-in-the-Loop (SIL)` or `Processor-in-the-Loop (PIL)`	Top model simulation mode specified by **Simulation Mode** setting. Model blocks in SIL or PIL mode are set to normal mode.	Top model simulation mode specified by **SIL/PIL Mode** setting. Model blocks use specified simulation modes.
Model block	**System Under Test** — Model blocks in SIL/PIL mode **Top Model Mode** — `Normal` or `Accelerator`	Top model simulation mode specified by **Top Model Mode** setting. Model blocks in SIL or PIL mode are set to normal mode.	Top model simulation mode specified by **Top Model Mode** setting. Model blocks use specified simulation modes.
Test harness The app supports test harnesses that you create from models and Model blocks. For information about creating test harnesses, see “Create or Import Test Harnesses and Select Properties” (Simulink Test).	**System Under Test** — Harness owner. Not editable. **Simulation Mode** — `Normal`, `Accelerator`, or `Rapid Accelerator` **SIL/PIL Mode** — `Software-in-the-Loop (SIL)` or `Processor-in-the-Loop (PIL)`	Harness model in normal mode. Model block mode for harness owner specified by **Simulation Mode** setting. Internal Model blocks of harness owner in SIL or PIL mode set to normal mode.	Harness model in normal mode. Model block for harness owner specified by SIL/PIL Mode setting. Internal Model blocks of harness owner use specified simulation modes.

If you click **Settings**, the app provides these controls:

- Under **SIL/PIL Settings**:
  - **Settings** — Display **Verification** parameters in the Configuration Parameters dialog box.
  - **Portable Word Sizes** — Select or clear **Enable portable word sizes**. For information about how you can generate code that supports compilation on your development computer, see “Configure Hardware Implementation Settings” on page 88-18.
  - **Debugging** — Select or clear **Enable source-level debugging for SIL or PIL**.
- Under **Coverage**:
  - **Coverage Settings** — Display **Coverage** parameters in the Configuration Parameters dialog box.

  - **Coverage Collection** — Enable or disable code coverage. If code coverage with Simulink Coverage is enabled, clicking the button toggles `CovEnable` between `'on'` and `'off'`. For more information, see “Code Coverage”.
- Under **Time Profiling**:
  - **Task Profiling** — Select or clear **Measure task execution time**.
  - **Save Options** — If **Task Profiling** is selected, clicking the button sets the **Save options** configuration parameter to `Summary data only`, `All data`, and `Metrics only` cyclically. If **Task Profiling** is not selected, the button is dimmed.
  - **Functions** — Clicking the button sets the **Measure function execution times** configuration parameter to `Off`, `Coarse (referenced models and subsystems only)`, and `Detailed (all function call sites)` cyclically.

  For more information, see “Configure Code Execution Profiling” on page 81-2 and “Control Profiling Granularity” on page 81-3.
- Under **Stack Profiling**, **Stack Profiling**— Select or clear **Measure task stack usage**. For more information, see “Stack Usage Profiling for Code Generated from Simulink Models” on page 81-53.

If you click **Monitor Signals**, the app provides these controls:

- Under **Monitor Signals**:
  - **Log Selected Signals** — Enable signal logging for model simulation.
  - **Make Selected Signals Testpoints** — Required for signal observation in SIL or PIL simulations.
- Under **Logging Options**:
  - **Signal Logging** — Set the `SignalLogging` configuration parameter to `'on'` or `'off'`, which enables or disables signal logging to the MATLAB workspace and signal streaming to the Simulation Data Inspector.
  - **State Logging** — Set the `SaveState` configuration parameter to `'on'` or `'off'`, which enables or disables state data logging to the MATLAB workspace and state data streaming to the Simulation Data Inspector.
  - **Signal Table** — Display the **Signals** tab in the Model Data Editor.
  - **Configure Logging** — Display the **Data Import/Export** parameters in the Configuration Parameters dialog box.
  - **Normal Mode Visibility** — Open the Model Block Normal Mode Visibility dialog box.

To monitor component signals and state data and compare values from the model and SIL or PIL simulations:

1. For each signal that you want to log:
   a. In the Simulink Editor, select the signal.
   b. On the **SIL/PIL** tab, click **Monitor Signals**. From the gallery, select these options:
      - **Log Selected Signals**
      - **Make Selected Signals Testpoints**
      - **Signal Logging**

2. To log state data, from the **Monitor Signals** gallery, select **State Logging**.
3. If the **Language** configuration parameter is `C++`, in the **Code Mappings** editor, set the **Data Visibility** for the signals, states, and internal data model element categories to `public`.

For more information, see “Log Signals of a Component” on page 88-27.

Run Automated Verification, Model Simulation, or SIL/PIL Simulation

With one click, you can run back-to-back model and SIL or PIL simulations. You can also run individual simulations.

If you select the **Automated Verification** mode, the **SIL/PIL** tab displays the **Run Automated Verification** section. To run back-to-back simulations:

1. In the **Stop Time** field, specify the simulation time. For the **Automated Verification** mode, you cannot specify `inf`.
2. Click **Run Verification**.

If you select the **Simulation Only** or **SIL/PIL Simulation Only** mode, the **SIL/PIL** tab displays the **Run** section. In this case, the toolstrip provides additional controls:

- **Fast Restart** — Click to turn on fast restart. For the **Simulation Only** mode, the compiled model does not contain components in SIL or PIL mode.
- **Step Back and Step Forward** — You can configure stepping through a simulation by using the **Simulation Stepping Options** tool. To open the tool, click **Step Back**, and then select **Configure Simulation Stepping**.

  Simulation stepping is not available if the mode is **SIL/PIL Simulation Only** and **System Under Test** is `Top model`.

To run an individual simulation:

1. In the **Stop Time** field, specify the simulation time.
2. Click **Run Simulation** or **Run SIL/PIL**.

Analyze Results and Export Test Cases

In the app **Results** section, you can analyze results. If you click **Compare Runs** or **Data Inspector**, the app provides these controls in the Results gallery.

Section	Control	Description
Numerical Equivalence Results	**Compare Runs**	Compare results from the last model and SIL/PIL simulations by using the Simulation Data Inspector. Use the comparison to decide whether the results are numerically consistent. If the mode is **Automated Verification**, at the end of back-to-back simulations, the Simulation Data Inspector compares logged signals from the model and SIL or PIL simulations.

Section	Control	Description
	Data Inspector	Open the Simulation Data Inspector.
Code	**Code View**	By default, at the end of SIL or PIL simulations, the SIL/PIL Manager displays the Code view, which enables you to: • Analyze generated code. • See code metrics. • Trace between model elements and generated code. If the view is closed and you want to reopen it, click **Code View**.
Execution Profiling Results	**Highlight Profiling**	Highlight blocks profiled in the last SIL or PIL simulation.
	Generate Report	Open code execution profiling report for the last SIL or PIL simulation. For more information, see “View and Compare Code Execution Times” on page 81-6.
	Generate Schedule	In the Simulation Data Inspector, show task scheduling for the last SIL or PIL simulation. For more information, see “Visualize Task Scheduling in SIL Simulation”.
Stack Profiling Results	**Generate Report**	Open stack usage profiling report for the last SIL or PIL simulation. For more information, see “View Stack Usage Metrics for Tasks and Functions” on page 81-54.
Coverage Analysis Results (Controls available only if you have a Simulink Coverage license)	**Coverage Highlighting**	Highlight blocks that are analyzed for coverage and provide summary results.
	Coverage Details	Display the code coverage report in the Simulink Editor.
	Results Explorer	Open Results Explorer. For more information, see “Access, Manage, and Aggregate Coverage Results” (Simulink Coverage).
	Generate Report	Generate the code coverage report in a web browser.

Section	Control	Description
Simulink Test	**Export to Test Manager**	If you have Simulink Test, using the current app settings, you can create and export an equivalence test case. The test case enables you to run to back-to-back simulations and check whether model components and the generated code are numerically equivalent. **1** Click **Export to Test Manager**. **2** In the Export SIL/PIL Test Cases dialog box: **a** Click **Use a new test file** or **Reuse an existing file**. **b** In the **Test file** field, specify the name of the test file. By default, the dialog box specifies: `currentWorkingFolderPath\modelName_Testfile.mldatx` If the test file exists and you click **Use a new test file**, the app overwrites the existing file. If you click **Reuse an existing file**, the app adds the test case to the existing file. **3** Click **OK**. The app creates an equivalence test case in the specified test file. For information about using exported test cases, see “Import Test Cases for Equivalence Testing” (Simulink Test).

See Also

SIL/PIL Manager

More About

- “Choose a SIL or PIL Approach” on page 88-11
- “Code Coverage”
- “Code Execution Profiling”

Security for PIL Simulations

Ensure that access to your development computer and the target hardware is secure.

Provide a secure communication channel between your development computer and the target application by applying the security measures listed in this table.

Communication Protocol	Security Measure
TCP/IP	Run your development computer and target application within a trusted private network or virtual private network (VPN).
Serial	Use a point-to-point link between your development computer and target application, for example, a serial cable.

SIL and PIL Limitations

In this section...
“About SIL and PIL Limitations” on page 88-83 “General SIL and PIL Limitations” on page 88-83 “Top-Model SIL/PIL Limitations” on page 88-90 “Model Block SIL/PIL Limitations” on page 88-91 “Subsystem Block SIL/PIL Limitations” on page 88-92

About SIL and PIL Limitations

With Embedded Coder, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations by using:

- The top-model.
- Model blocks.
- Blocks from subsystems.

The following sections describe modeling and code generation features that are either unsupported or partially supported by SIL and PIL simulations.

General SIL and PIL Limitations

Tunable Parameters and SIL/PIL

For Model block SIL/PIL and SIL/PIL block simulations, you can tune tunable *workspace* parameters but not tunable *dialog box* parameters. For information about tuning parameters, see “Tune and Experiment with Block Parameter Values”.

For a top model with tunable parameters, you can run a SIL/PIL simulation but you cannot tune the parameters during the simulation.

The software cannot define, initialize, or tune the following types of tunable workspace parameters.

Parameter description	Software response		
	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Parameters with storage class that applies `"static"` scope or `"const"` keyword. For example, `Custom`, `Const`, or `ConstVolatile`	Warning	Warning	Warning
Parameters with multiword, fixed-point data types	Warning	Error	Warning
Parameters with data types that have different sizes on host and target	Warning	Error	Warning

For C++ class code, SIL/PIL you can tune tunable workspace parameters when **Parameter visibility** is `public`. If **Parameter visibility** is `private` or `protected`, tuning is supported only if **Parameter access** is `Method` or `Inlined method`.

For top-model SIL/PIL and the SIL/PIL block, consider the case where all of the following conditions apply:

- **Code Generation** > **Interface** > **Code interface packaging** is `Reusable function`.
- **Code Generation** > **Interface** > **Use dynamic memory allocation for model initialization** is not selected.
- **Optimization** > **Default parameter behavior** is `Tunable`.
- In the Code Mapping Editor for the model, the default storage class for the **Global parameters** or **Local parameters** category is set to `Default`, and the model contains corresponding parameters that use the storage classes `Auto` or `Model default`.

If the SIL/PIL component cannot dynamically initialize tunable parameters in the `rtP` model parameter structure, you see an error message like the following:

```
Parameter Dialog:InitialOutput in 'SILTopModel/CounterTypeA/count'
is part of the imported "rtP" structure in the generated code but cannot be
initialized by SIL or PIL. To avoid this error, make sure the parameter
corresponds to a tunable workspace variable. Alternatively, set
"Configuration Parameters > Code Generation > Interface > Code interface packaging"
to ''Nonreusable function'', or search for ''Use dynamic memory allocation
for model initialization'' in the Configuration Parameters dialog box
and select the checkbox.
```

If you select **Code Generation** > **Interface** > **Use dynamic memory allocation for model initialization**, this limitation does not apply.

For Model block SIL/PIL, if you specify **Code interface** to be `Top model`, you can tune parameters while a simulation runs. If you tune parameters between successive runs of the simulation, the software generates new code for the later run. The new code uses your latest settings as initial parameter values.

For top model or Model block SIL/PIL, if you change the value of a MATLAB variable or parameter object (such as `Simulink.Parameter`) that you store in a workspace, Simulink Coder regenerates the C code.

For Model block SIL/PIL, you can tune model workspace parameters between simulations, including when fast restart is enabled. You cannot tune the parameters during a simulation. You can use:

- Model workspace parameters with imported storage classes.
- Exported model workspace parameters with no data initialization.
- Model workspace parameters that map to AUTOSAR shared parameters.

Automatic definition or initialization of model workspace parameters for Model block SIL/PIL is supported only for model workspace parameters at the top-level of the SIL/PIL component under test. The support is not available for model workspace parameters at a lower level in the model reference hierarchy, which can lead to a mismatch between normal and SIL/PIL simulation results.

Global and Local Data Stores

SIL/PIL supports global data stores. For components that are not export-function models, SIL/PIL block simulations that access global data stores must be single rate. Otherwise, the software produces an error.

SIL/PIL does not support local data stores.

Model block SIL/PIL does not support local Data Store Memory blocks that have these parameter settings:

- **Share across model instances** – Selected.
- **Storage class** – `Auto` or `Model default`.

You cannot create SIL/PIL blocks from models that use local Data Store Memory blocks with the **Share across model instances** parameter selected.

SIL/PIL Does Not Check Simulink Coder Error Status

SIL/PIL does not check the Simulink Coder error status of the generated code under test. This error status flags exceptional conditions during execution of the generated code.

Blocks in the model can also set the Simulink Coder error status, for example, custom blocks that you create. SIL/PIL does not check this error status and report errors.

Missing Code Interface Description File Errors

SIL/PIL requires a code interface description file, which is created during code generation for the component under test. If the code interface description file is missing, the SIL/PIL simulation cannot proceed. You see an error reporting that the file does not exist. If you select the unsupported option **Classic call interface**, this error can occur. Therefore, do not select the option.

To Workspace Block

If you enable MAT-file logging, top-model SIL/PIL and SIL/PIL blocks support To Workspace blocks.

Model block SIL/PIL does not support To Workspace blocks.

Cannot Connect SIL/PIL Outputs to Merge Block

If you connect Model block SIL/PIL or SIL/PIL block outputs to a Merge block, you see an error because S-function memory is not reusable.

Unsupported Blocks

SIL/PIL does not support the following blocks:

- Scope blocks, and all types of run-time display. For example, display of port values and signal values.
- Stop blocks. SIL/PIL ignores the Stop Simulation block and continues simulating.

Multiword Fixed-Point I/O

You cannot run SIL and PIL simulations of models that have multiword, fixed-point signals across component boundaries.

Fixed-Point Data Types Wider Than 32 Bits

SIL/PIL supports fixed-point data types that are wider than 32 bits. For example:

- 64-bit `long` and `long long`
- 64-bit execution profiling timer data type
- `int64` and `uint64` in MATLAB Coder SIL execution.

The following constraints apply:

- For 64-bit data type support, the data type must be representable as `long` or `long long` on the MATLAB host *and* the target. Otherwise, the software uses the multiword, fixed-point approach, which SIL/PIL does not support.
- The software does not support the 40-bit `long` data type of the TI's C6000™ target.

  Through the **Configuration** > **Hardware Implementation** pane, you can enable support for the 64-bit `long long` data type. For data types with widths between 33 and 40 bits (inclusive), the software implements the data types using the 40-bit `long` data type, which SIL/PIL does not support.

Data Type Replacement

For SIL/PIL, the data type replacement name for `boolean` must resolve to `boolean`, `uint8`, or `int8`.

Simulink.ImageType Data Type

SIL/PIL does not support Simulink Function and Data Store Memory blocks that require the transfer of `Simulink.ImageType` data between Simulink and the target application. At the interface:

- For Simulink Function blocks, the data type of function arguments must not be `Simulink.ImageType`.
- Data Store Memory blocks must not use `Simulink.ImageType` data.

If the Simulink Function and Data Store Memory blocks are not at the interface but inside the target application, the limitation does not apply.

Continuous Sample Times

Top-model SIL/PIL and SIL/PIL block do not support continuous sample times at the SIL or PIL component boundary. However, they support continuous sample times within the component.

Model block SIL/PIL does not support continuous sample times.

Variable-Size Signals

Model block SIL/PIL simulations support variable-size signals only if **Diagnostics** > **Model Referencing** > **Propagate sizes of variable-size signals** is `During execution`.

Top-model SIL/PIL and SIL/PIL block simulations treat variable-size signals at the I/O boundary of the SIL/PIL component as fixed-size signals, which can lead to errors during propagation of signal sizes. To avoid such errors, use only fixed-size signals at the I/O boundary of the SIL/PIL component.

There can be cases where no error occurs during propagation of signal sizes. In these cases, the software treats variable-size input signals as zero-size signals.

Internal Signal Logging

SIL/PIL blocks do not support signal logging. For a workaround, see "Log Signals of a Component" on page 88-20.

The following internal signal logging limitations apply to top-model and Model block SIL/PIL simulations.

Limitation	Applies To	
	Top-Model SIL/PIL	Model Block SIL/PIL
Signals feeding merge blocks are not supported for logging in normal simulation but are logged in SIL/PIL mode. The logged values during SIL/PIL are the same as the logged values for the output of the merge block.	Yes	No
Top-model normal simulation logs data at a periodic rate but top-model SIL/PIL simulation logs data at a constant rate under these circumstances: • **Default parameter behavior** is `Tunable`. • A constant sample time signal from a Model block is logged in the top model. • The logged signal is not directly connected to a root-level output port. To avoid this behavior and log at the constant rate in all simulation modes, set **Default parameter behavior** to `Inlined`.	Yes	No
Features not supported: • Signal logging in models referenced by the SIL/PIL component. • Signal logging in Simulink Function block. • Virtual signals. • Virtual buses. • Frame-based signals, which are logged as sample-based signals. • Continuous, asynchronous, and triggered sample times. At the top-level of export-function models, you can log signals with triggered sample times. • Logging of Stateflow states and local data.	Yes	Yes
Variable-size, function-call, and Action signals are not supported. A normal simulation produces an error. A SIL/PIL simulation produces a warning.	Yes	No
State port signals are not supported. A normal simulation produces an error. A SIL/PIL simulation does not produce a warning.	Yes	No

Custom Storage Class of Type Other

SIL/PIL simulations support the custom storage class where **Type** is set to `Other` on page 88-7. These limitations apply:

- If the code fragments returned by the TLC file associated with the custom storage class are incomplete, the SIL/PIL application might fail to compile, or produce incorrect results.
- Custom storage classes with `Imported` scope and `Pointer` access are supported, but you must provide code to initialize the pointer. For example, you can modify the response to

`DataAccess(record, "define", "", "")` to provide a definition of the storage implementation, and initialize the pointer to the address of the implementation variable.

- To determine whether a variable is `const` and therefore not tunable, the build process for the SIL/PIL application uses the memory section definition in the custom storage class. If the custom storage class defines a variable as `const` and is not associated with a `const` memory section, the target application might fail to compile, or produce an error during the simulation. In this case, associate the custom storage class with a memory section for which `Is const` is specified, for example, `MemConst`. Alternatively, if `SupportSILPIL` is an instance-specific parameter in the custom attributes class, for the associated signal or parameter, set `CoderInfo.CustomAttributes.SupportSILPIL` to `false`.

Unsupported Implementation Errors

If you use a data store, signal, or parameter implementation that SIL/PIL does not support, you can see errors like the following:

```
The following data interfaces have
implementations that are not supported by SIL or PIL.
```

`data interfaces` can be global data stores, `inports`, `outports`, or `parameters`.

The model output port has been optimized through virtual output port optimization. See “Virtualized Output Ports Optimization” on page 78-15. The error occurs because the properties (for example, data type, dimensions) of the signal or signals entering the virtual root output port have been modified by routing the signals in one of the following ways:

- Through a Mux block.
- Through a block that changes the signal data type. To check the consistency of data types in the model, display **Port Data Types**.
- Through a block that changes the signal dimensions. To check the consistency of data types in the model, display **Signal Dimensions**.

Hardware Implementation

PIL does not support multiword data types where the word order differs from the target byte order. The PIL simulation fails, displaying undefined behavior.

PIL requires that you configure the correct **Hardware Implementation** settings for the target environment, including byte ordering for targets. If you do not specify the correct byte ordering, the PIL simulation fails, displaying undefined behavior.

Bus Elements Mapped to Imported Bit-Field Definitions

If you map Simulink bus elements to bit fields through an imported header file, a SIL or PIL simulation produces a build error. For example, if your model has an Inport block connected to a bus that is a `Simulink.Bus` object with these properties:

- **Name** — `myBus`
- **Bus elements** — An array of `Simulink.BusElement` objects with these properties.

Name	DataType	Complexity	Dimensions
`bitField0`	`boolean`	`real`	`1`

Name	DataType	Complexity	Dimensions
bitField1	boolean	real	1
bitField2	boolean	real	1
bitField3	boolean	real	1
bitField4	boolean	real	1
bitField5	boolean	real	1

- **Data scope** — `Imported`
- **Header file** — `busSpecification.h`. This file contains `myBus`, which defines C bit-field data types for the bus elements.

```
typedef struct myBus
{
   unsigned int bitField0 : 1;
   unsigned int bitField1 : 1;
   unsigned int bitField2 : 1;
   unsigned int bitField3 : 1;
   unsigned int bitField4 : 1;
   unsigned int bitField5 : 1;
} myBus;
```

Size Mismatch Between Simulink and Target Hardware Data Types

When a Simulink data type and the corresponding target hardware data type differ in size, a SIL or PIL simulation produces an error. This size mismatch can occur if you map a Simulink data type to the target hardware data type through definitions in an imported header file. For example, if you create a data type alias, `T_BOOL`, which is a `Simulink.AliasType` object with these properties:

- **Base type** — `boolean`.
- **Mode** — `Built in`, `boolean`.
- **Data scope** — `Imported`.
- **Header file** — `myDefinitions.h`. This file defines `T_BOOL` as an enumerated data type:

```
typedef enum _BOOL_TYPE
{
  FALSE           = 0,
  TRUE            = 1
} BOOL_TYPE;

typedef BOOL_TYPE T_BOOL;
```

  In this case, the compiler for the target hardware determines the size of `T_BOOL`, which can differ from the size of the Simulink data type, `boolean`.

SIL Simulations with Target-Specific Custom Code

Target-specific custom code that is not portable for execution on your development computer can produce compilation or run-time failures during a SIL simulation.

For example, SIL does not support the use of custom code that explicitly casts pointers to integer-type variables that are smaller than the length of a pointer variable on your development computer. Consider using one of these alternatives:

- Run a PIL simulation.
- If you have custom code that casts pointers to a 32-bit integer type, for your development computer, set up a PIL target connectivity configuration that uses a toolchain that is configured to build a 32-bit binary application.

Sample Time Differences in Simulation Data Inspector Comparisons

If you use the Simulation Data Inspector to compare logged model outputs from normal and SIL or PIL simulations, the sample times of the outputs might differ. You can still use the Simulation Data Inspector to verify that the outputs match numerically.

Top-Model SIL/PIL Limitations

Top-Model Root-Level Logging

Top-model SIL/PIL supports signal logging for signals connected to root-level inports and outports. The C API is not required. Root-level logging has the following limitations:

- The characteristics of the logged data such as data type, sample time, and dimensions must match the characteristics of the root-level inports and outports (rather than the characteristics of the connected signal).

  In some cases, there can be differences in data type and dimensions between the signal being logged and the root inport or outport that the signal is connected to. Consider the following examples.

  - If a signal being logged has matrix dimensions `[1x5]` but the outport connected to the signal has vector dimensions (5), then the data logged during a SIL or PIL simulation has vector dimensions (5).
  - If a signal being logged has scalar dimensions but the outport connected to the signal has matrix dimensions `[1x1]`, then the data logged during a SIL or PIL simulation has matrix dimensions `[1x1]`.
- Signals connected to duplicated inports are not logged during SIL/PIL simulation. No warning is issued.

  During normal simulation, signals connected directly to duplicated inports are logged.
- The Signal Logging Selector / `DataLoggingOverride` override mechanism is not supported.
- Normal and SIL/PIL simulations log bus signals with names that are different when all of the following conditions apply:

  - The `SaveOutput` or `SignalLogging` configuration parameter is `on`.
  - The names of the elements in the bus signal are different from the corresponding names in the bus object. For example, when the `InheritFromInputs` parameter for a Bus Creator block is set to `'on'`.
- The software inserts the suffix `_wrapper` for *output logging* if the save format is `Structure` or `Structure with time` and you run the `sim` command without specifying the single-output format. The software adds `_wrapper` to the block name for signals in `yout`. If the save format is `Array`, the software does not add the suffix. For example:

  ```
  >> yout.signals

  ans =
  ```

```
          values: [11x1 double]
      dimensions: 1
           label: 'SignalLogging'
       blockName: 'sillogging_wrapper/OutputLogging'
```

To avoid this behavior, run command-line simulations with the `sim` command specifying the single-output format. See “Run Simulations Programmatically”.

Callback Support

SIL/PIL does not support the callbacks (model or block) `StartFcn` and `StopFcn`.

Note Top-model SIL/PIL supports the callback `InitFcn`.

Incremental Build

When you start a top-model SIL/PIL simulation, the software regenerates code if it detects changes to your model. The software detects changes by using a checksum for the model. The software does not detect changes that you make to:

- The `HeaderFile` property of a `Simulink.AliasType` object
- Legacy S-functions

If you make these changes, build (**Ctrl-B**) your model again before starting the next PIL simulation.

Model Block SIL/PIL Limitations

Top-Model Code Testing

The following limitations apply:

- Because model arguments do not apply to a top model, when the **Code interface** block parameter is set to `Top model`, the software does not support the **Model arguments** block parameter.
- Conditional execution does not apply to a top model. If a Model block is set up to execute conditionally and the **Code interface** block parameter is set to `'Top model'`, the software produces an error when you run a SIL or PIL simulation.
- For sample time independent models, you must set **Configuration Parameters** > **Solver** > **Periodic sample time constraint** to `Ensure sample time independent`.

Conditionally Executed Subsystem

You see an error if:

- You place your Model block, in either SIL or PIL simulation mode, in a conditionally executed subsystem and the referenced model is multirate (that is, has multiple sample times). Single-rate, referenced models (with only a single sample time) are not affected.
- Your Model block, in either SIL or PIL simulation mode, has blocks that depend on absolute time **and** is conditionally executed.

Outputs with Constant Sample Time

If the block parameter **Code interface** is `Top model`, Model block SIL/PIL supports outputs with constant sample time.

Noninlined S-Functions

Model-block SIL/PIL simulations do not support noninlined S-functions.

Referenced Models That Use Same Target Connectivity Configuration

Consider a top model with two or more Model blocks that reference models that use the same target connectivity configuration. If the Model blocks are in PIL mode simultaneously, you cannot run a simulation of the top model. An error occurs.

SIL and PIL Instances of a Referenced Model

Consider a top model that contains two instances of a Model block that reference the same model. If one instance is in SIL mode and the other instance is in PIL mode, you cannot run a simulation of the top model. An error occurs.

Subsystem Block SIL/PIL Limitations

Atomic Subsystem Workflow Limitations

The workflow described in “Unit Test Subsystem Code with SIL/PIL Manager” on page 88-31 does not support:

- These subsystem types:
  - Virtual
  - Function-call
  - Stateflow
  - For Each
  - Action
  - Initialize Function
  - Reset Function
  - Terminate Function
  - Simulink Function
- The `'Compact'` option for `ERTFilePackagingFormat` if `PreserveStaticInFcnDecl` is set to `'on'`.
- Subsystems that generate in-lined code. The subsystem block parameter `RTWSystemCode` must be `'Nonreusable function'` or `'Reusable function'`.
- Subsystems that contain:
  - Referenced models.
  - Simulink Function blocks.
  - MATLAB Function blocks.

  - Data Store Memory, Data Store Read, and Data Store Write blocks.
- The LDRA tool suite and the BullseyeCoverage tool for code coverage analysis.
- Virtual buses at the subsystem interface.

The results from model and SIL or PIL simulations might differ numerically if:

- The subsystem uses non-finite numbers, for example, `NaN` and `Inf`.
- The subsystem and the parent model have multiple sample rates and rate transitions.
- A lookup table block in the subsystem is connected to a Prelookup block across the subsystem boundary.

Variant Condition Propagation with Variant Source and Variant Sink Blocks

SIL/PIL block simulations do not support the propagation of variant conditions across component boundaries.

PIL Block Mux

The PIL block supports mux signals, except mixed data-type mux signals that expand into individual signals during a right-click subsystem build.

Code Coverage

SIL block simulations do not support the generation of code coverage results. PIL block support for code coverage depends on your target connectivity configuration and third-party product support.

Subsystem with Inherited Sample Time Blocks

When you create a SIL/PIL block from a subsystem that has blocks with inherited sample times, the generated code and SIL/PIL wrapper acquire the sample time of the original parent model. If you use the SIL/PIL block in a context that does not allow explicit sample times, for example, within a triggered subsystem, you see an error.

Try one of these workarounds:

- Before you create the SIL/PIL block, in the parent model, set **Configuration Parameters** > **Solver** > **Periodic sample time constraint** to `Ensure sample time independent`.
- Using the subsystem, create a Model block that is independent of sample time. With this block, run Model block SIL/PIL simulations.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11

Test Generated Code with SIL and PIL Simulations

Test numerical equivalence between model components and production code that you generate from the components by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

With a SIL simulation, you test source code on your development computer. With a PIL simulation, you test the compiled object code that you intend to deploy in production by running the object code on real target hardware or an instruction set simulator. To determine whether model components and generated code are numerically equivalent, compare SIL and PIL results against normal mode results.

There are three ways of running SIL and PIL simulations. You can use the top model, Model blocks, or SIL and PIL blocks that you create from a subsystem. See “Choose a SIL or PIL Approach” on page 88-11.

Target Connectivity Configuration for PIL

Before you can run PIL simulations, you must configure target connectivity. The target connectivity configuration enables the PIL simulation to:

- Build the target application.
- Download, start, and stop the application on the target.
- Support communication between Simulink and the target.

To produce a target connectivity configuration, you can use the supplied target connectivity API. For details, see “Create PIL Target Connectivity Configuration for Simulink” on page 88-38.

For supported hardware, you can use target support packages. For details, see “Embedded Coder Supported Hardware”.

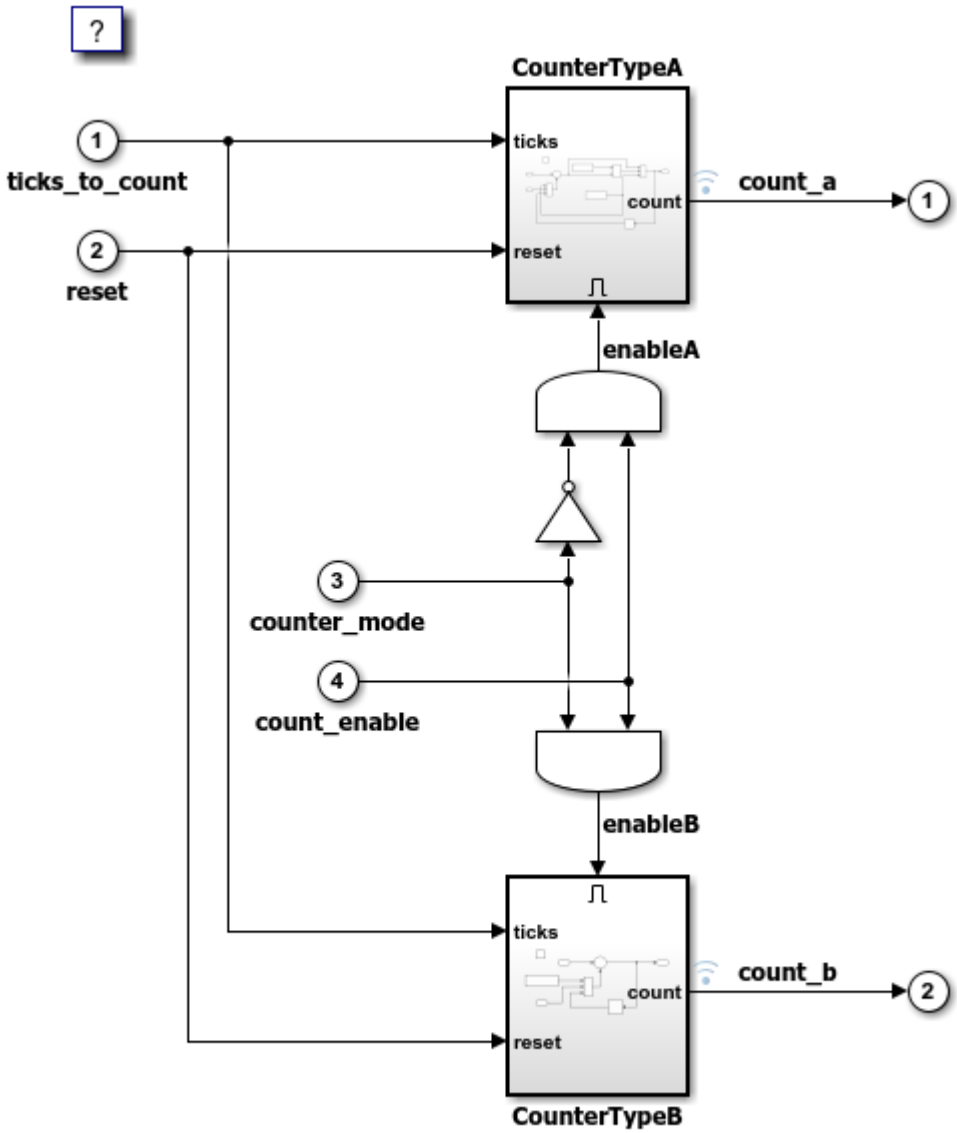
SIL or PIL Simulation with a Top Model

Test generated model code by running a top-model SIL or PIL simulation. With this approach:

- You test code generated from the top model, which uses the standalone code interface.
- You configure the model to load test vectors or stimulus inputs from the MATLAB workspace.
- You can easily switch the top model between the normal, SIL, and PIL simulation modes.

Open a simple counter top model.

```
model='SILTopModel';
close_system(model,0)
open_system(model)
```



To focus on numerical equivalence testing, turn off:

- Model coverage
- Code coverage
- Execution time profiling

```
set_param(gcs, 'RecordCoverage','off');
coverageSettings = get_param(model, 'CodeCoverageSettings');
coverageSettings.CoverageTool='None';
set_param(model, 'CodeCoverageSettings',coverageSettings);
set_param(model, 'CodeExecutionProfiling','off');
```

Configure the input stimulus data.

```
[ticks_to_count, reset, counter_mode, count_enable] = ...
    SILTopModelData(T);
```

Configure logging options in the model.

```
set_param(model, 'LoadExternalInput','on');
set_param(model, 'ExternalInput','ticks_to_count, reset, counter_mode, count_enable');
set_param(model, 'SignalLogging', 'on');
set_param(model, 'SignalLoggingName', 'logsOut');
set_param(model, 'SaveOutput','on')
```

Run a normal mode simulation.

```
set_param(model,'SimulationMode','normal')
sim_output = sim(model,10);
yout_normal = [sim_output.yout.signals(1).values sim_output.yout.signals(2).values];
```

Run a top-model SIL simulation.

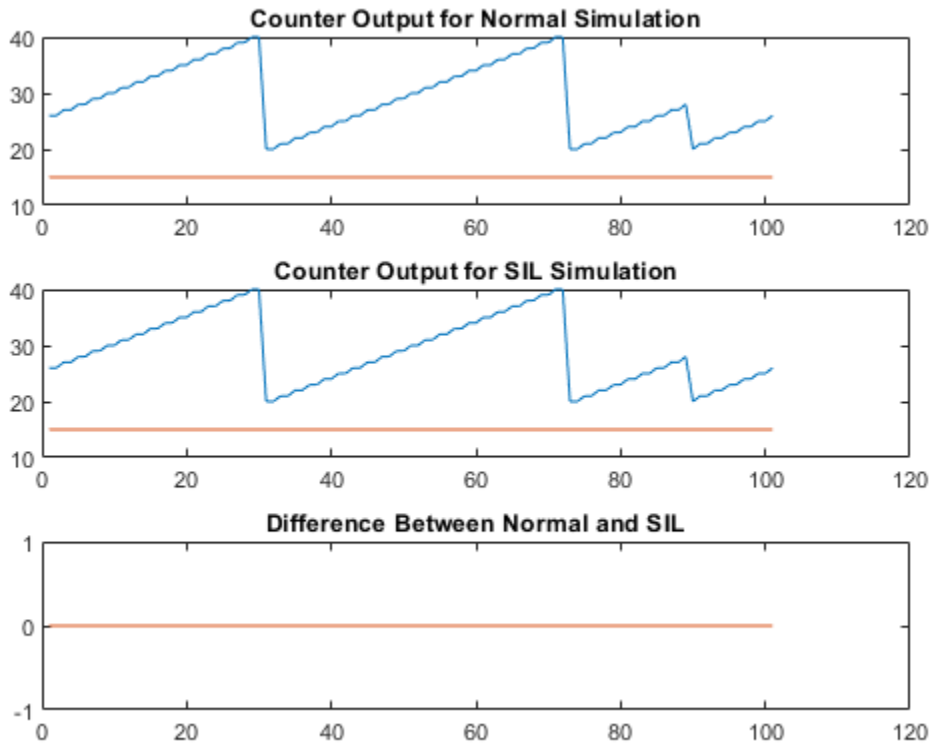
```
set_param(model,'SimulationMode','Software-in-the-Loop (SIL)')
sim_output = sim(model,10);
yout_sil = [sim_output.yout.signals(1).values sim_output.yout.signals(2).values];
```

```
### Starting build procedure for: SILTopModel
### Successful completion of build procedure for: SILTopModel

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================================
SILTopModel  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 15.935s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: SILTopModel
### Application stopped
### Stopping SIL simulation for component: SILTopModel
```

Unless up-to-date code for this model exists, new code is generated and compiled. The generated code runs as a separate process on your computer.

Plot and compare the results of the normal and SIL simulations. Observe that the results match.

```
fig1 = figure;
subplot(3,1,1), plot(yout_normal), title('Counter Output for Normal Simulation')
subplot(3,1,2), plot(yout_sil), title('Counter Output for SIL Simulation')
subplot(3,1,3), plot(yout_normal-yout_sil), ...
    title('Difference Between Normal and SIL');
```



Clean up.

```
close_system(model,0);
if ishandle(fig1), close(fig1), end
clear fig1
simResults = {'yout_sil','yout_normal','model','T',...
              'ticks_to_count','reset'};
save([model '_results'],simResults{:});
clear(simResults{:},'simResults')
```

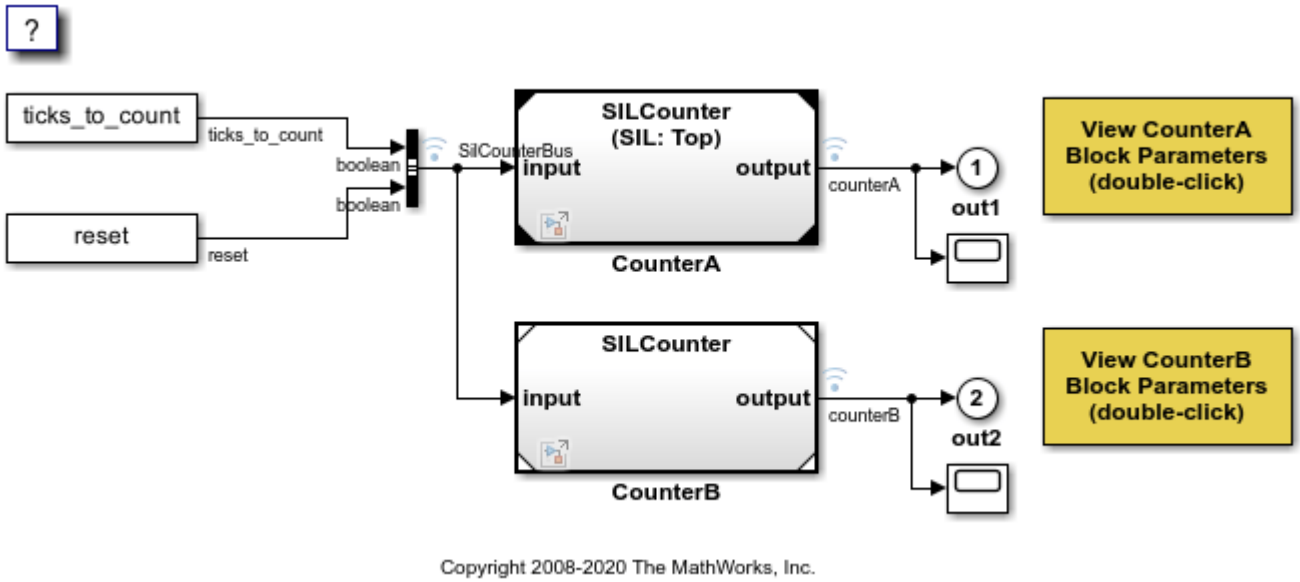
SIL or PIL Simulation with a Model Block

Test generated model code by using a test harness model that runs a Model block in SIL mode. With this approach:

- You can test code that is generated from a top model or a referenced model. The code from the top model uses the standalone code interface. The code from the referenced model uses the model reference code interface. For more information, see “Code Interfaces for SIL and PIL” on page 88-4.
- You use a test harness model or a system model to provide test vector or stimulus inputs.
- You can easily switch a Model block between the normal, SIL, and PIL simulation modes.

Open an example model that has two Model blocks which reference the same model. In a simulation, you run one Model block in SIL mode and the other Model block in normal mode.

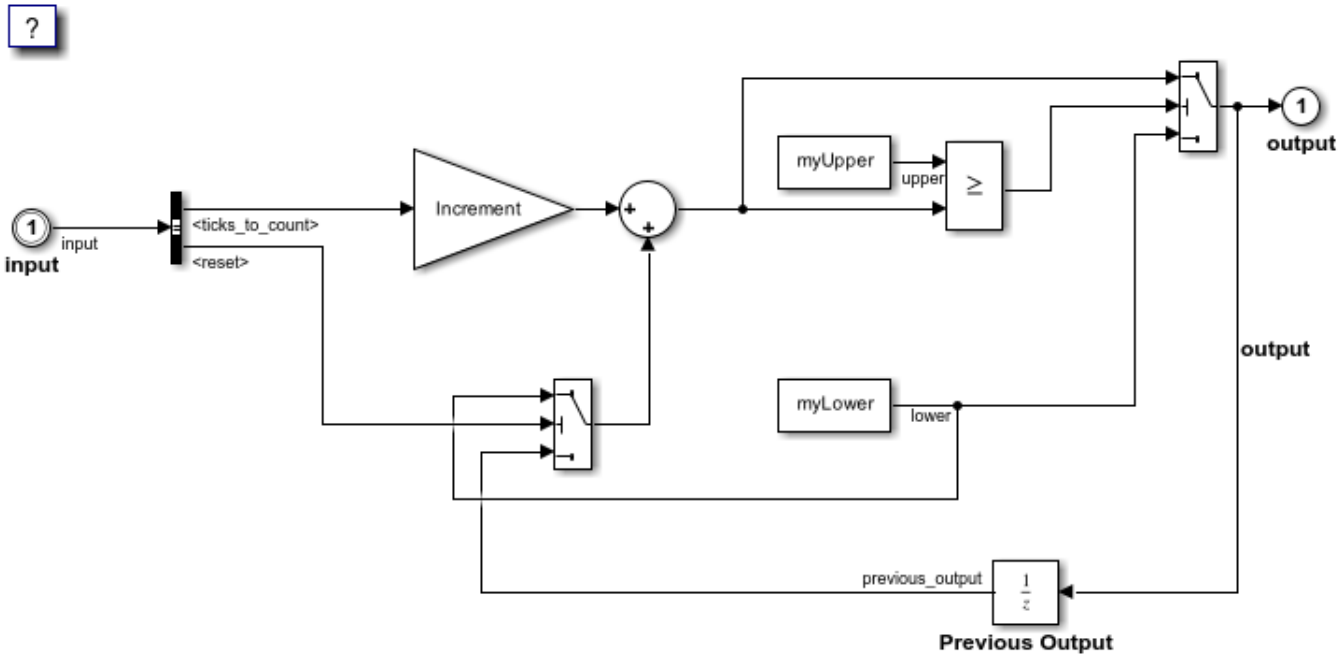
```
model='SILModelBlock';
open_system(model);
```



Turn off:

- Code coverage
- Execution time profiling

```
coverageSettings = get_param(model, 'CodeCoverageSettings');
coverageSettings.CoverageTool='None';
set_param(model, 'CodeCoverageSettings',coverageSettings);
open_system('SILModelBlock')
set_param('SILModelBlock', 'CodeExecutionProfiling','off');
open_system('SILCounter')
set_param('SILCounter', 'CodeExecutionProfiling','off');
currentFolder=pwd;
save_system('SILCounter', fullfile(currentFolder,'SILCounter.slx'))
```



Copyright 1994-2020 The MathWorks, Inc.

Configure state logging for the models.

```
set_param('SILCounter', 'SaveFormat','Dataset');
save_system('SILCounter', fullfile(currentFolder,'SILCounter.slx'))
set_param(model, 'SaveFormat','Dataset');
set_param(model, 'SaveState','on');
set_param(model, 'StateSaveName', 'xout');
```

Test Top-Model Code

For the Model block in SIL mode, specify generation of top-model code, which uses the standalone code interface.

```
set_param([model '/CounterA'], 'CodeInterface', 'Top model');
```

Run a simulation of the test harness model.

```
out = sim(model,20);
```

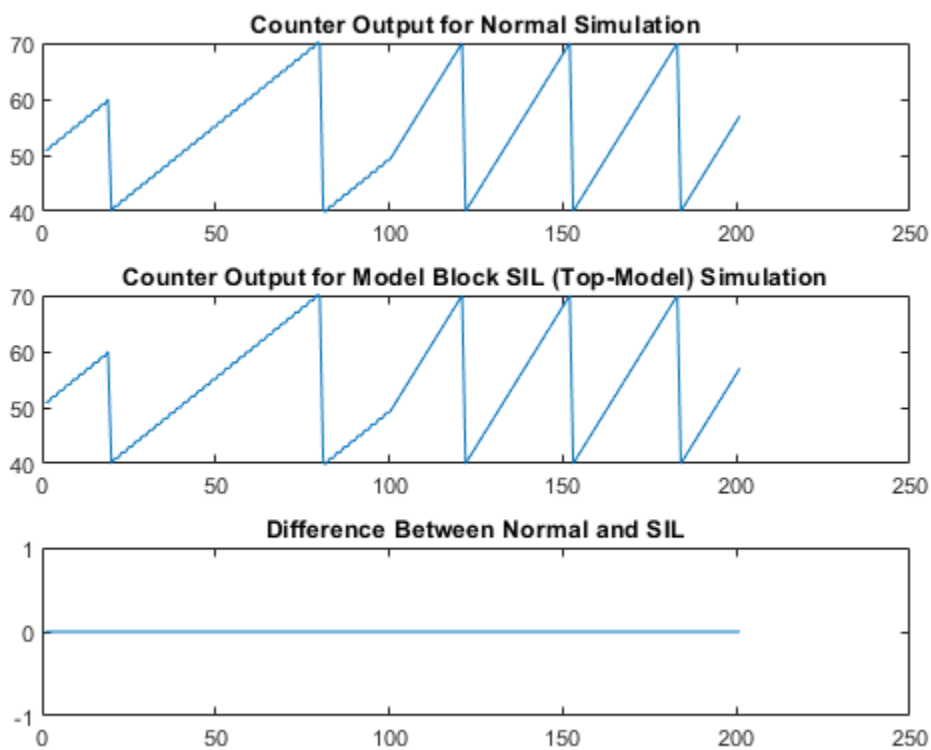
```
### Starting build procedure for: SILCounter
### Successful completion of build procedure for: SILCounter

Build Summary

Top model targets built:

Model       Action                        Rebuild Reason
========================================================================================
SILCounter  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.236s
```

```
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: SILCounter
### Application stopped
### Stopping SIL simulation for component: SILCounter
```

The model block in SIL mode runs as a separate process on your computer. In the working folder, you see that standalone code is generated for the referenced model unless generated code from a previous build exists.

Compare the behavior of Model blocks in normal and SIL modes. The results match.

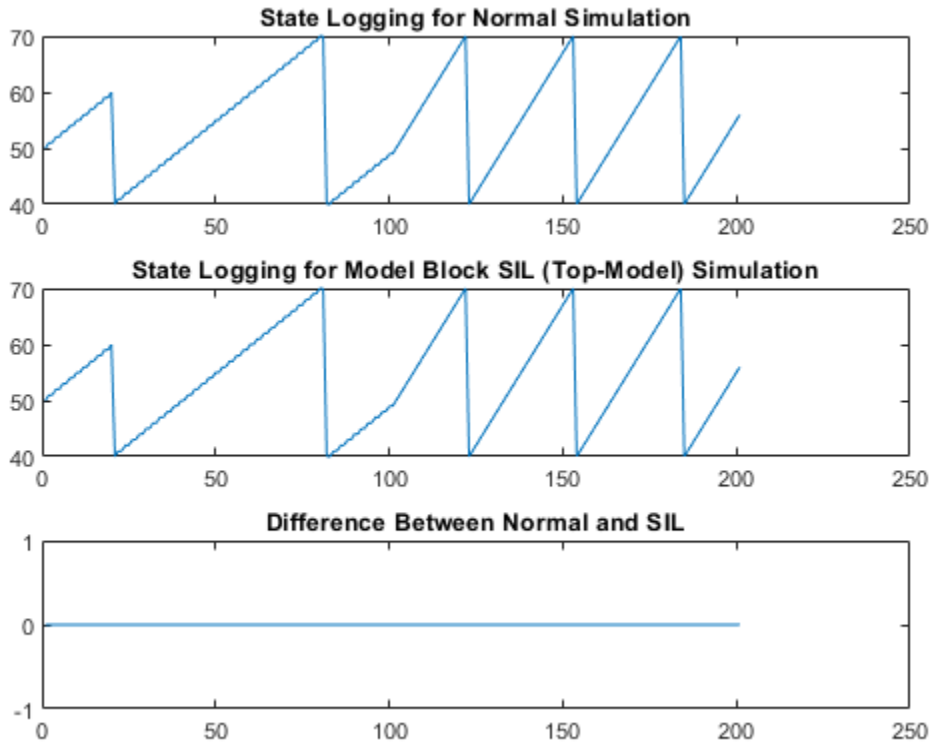
```
yout = out.logsOut;
yout_sil = yout.get('counterA').Values.Data;
yout_normal = yout.get('counterB').Values.Data;
fig1 = figure;
subplot(3,1,1), plot(yout_normal), title('Counter Output for Normal Simulation')
subplot(3,1,2), ...
    plot(yout_sil), title('Counter Output for Model Block SIL (Top-Model) Simulation')
subplot(3,1,3), plot(yout_normal-yout_sil), ...
    title('Difference Between Normal and SIL');
```



Compare the logged states of Model blocks from normal and SIL mode simulations.

```
xout = out.xout;
xout_sil = xout{1}.Values.Data;
```

```
xout_normal = xout{2}.Values.Data;
fig1 = figure;
subplot(3,1,1), plot(xout_sil), title('State Logging for Normal Simulation')
subplot(3,1,2), ...
    plot(xout_normal), title('State Logging for Model Block SIL (Top-Model) Simulation')
subplot(3,1,3), plot(xout_normal-xout_sil), ...
    title('Difference Between Normal and SIL');
```



Test Model Reference Code

For the Model block in SIL mode, specify generation of referenced model code, which uses the model reference code interface.

```
set_param([model '/CounterA'], 'CodeInterface', 'Model reference');
```

Run a simulation of the test harness model.

```
out2 = sim(model,20);
```

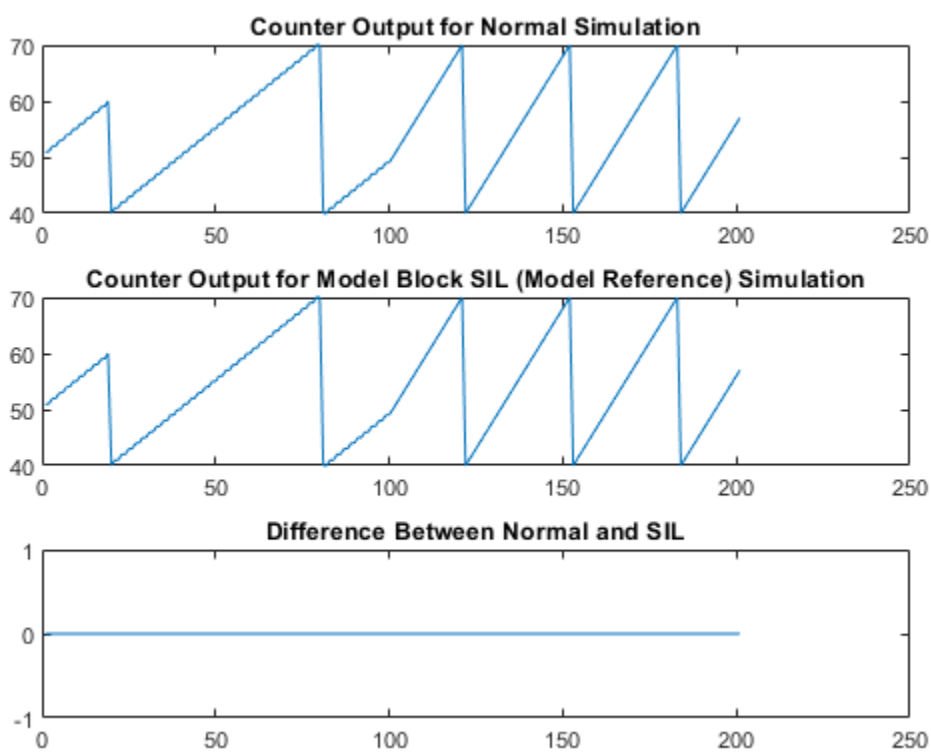
```
### Starting serial model reference code generation build.
### Starting build procedure for: SILCounter
### Successful completion of build procedure for: SILCounter

Build Summary

Code generation targets built:

Model       Action                        Rebuild Reason
```

```
==========================================================================
SILCounter  Code generated and compiled.  SILCounter.c does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 8.775s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: SILCounter
### Application stopped
### Stopping SIL simulation for component: SILCounter
```

The model block in SIL mode runs as a separate process on your computer. In the working folder, you see that model reference code is generated unless code from a previous build exists.

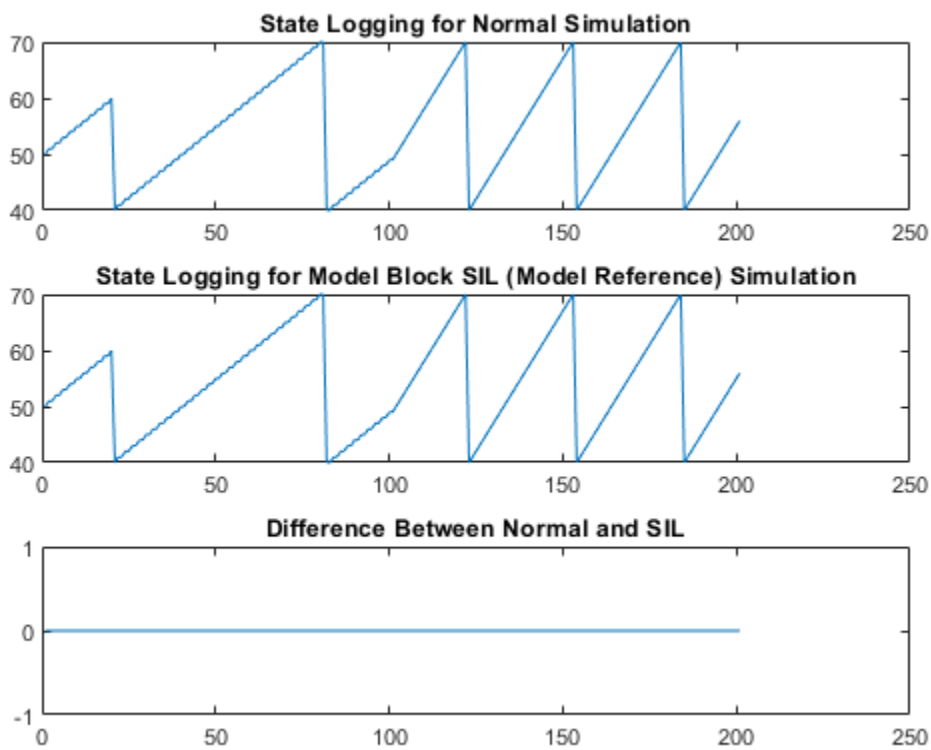
Compare the behavior of Model blocks in normal and SIL modes. The results match.

```
yout2 = out2.logsOut;
yout2_sil = yout2.get('counterA').Values.Data;
yout2_normal = yout2.get('counterB').Values.Data;
fig1 = figure;
subplot(3,1,1), plot(yout2_normal), title('Counter Output for Normal Simulation')
subplot(3,1,2), ...
    plot(yout2_sil), title('Counter Output for Model Block SIL (Model Reference) Simulation')
subplot(3,1,3), plot(yout2_normal-yout2_sil), ...
   title('Difference Between Normal and SIL');
```



Compare the logged states of Model blocks from normal and SIL mode simulations.

```
xout2 = out.xout;
xout2_sil = xout2{1}.Values.Data;
xout2_normal = xout2{2}.Values.Data;
fig1 = figure;
subplot(3,1,1), plot(xout2_sil), title('State Logging for Normal Simulation')
subplot(3,1,2), ...
    plot(xout2_normal), title('State Logging for Model Block SIL (Model Reference) Simulation')
subplot(3,1,3), plot(xout2_normal-xout2_sil), ...
    title('Difference Between Normal and SIL');
```



Clean up.

```
close_system(model,0);
if ishandle(fig1), close(fig1), end, clear fig1
simResults={'out','yout','yout_sil','yout_normal', ...
            'out2','yout2','yout2_sil','yout2_normal', ...
            'SilCounterBus','T','reset','ticks_to_count','Increment'};
save([model '_results'],simResults{:});
clear(simResults{:},'simResults')
```

SIL or PIL Block Simulation

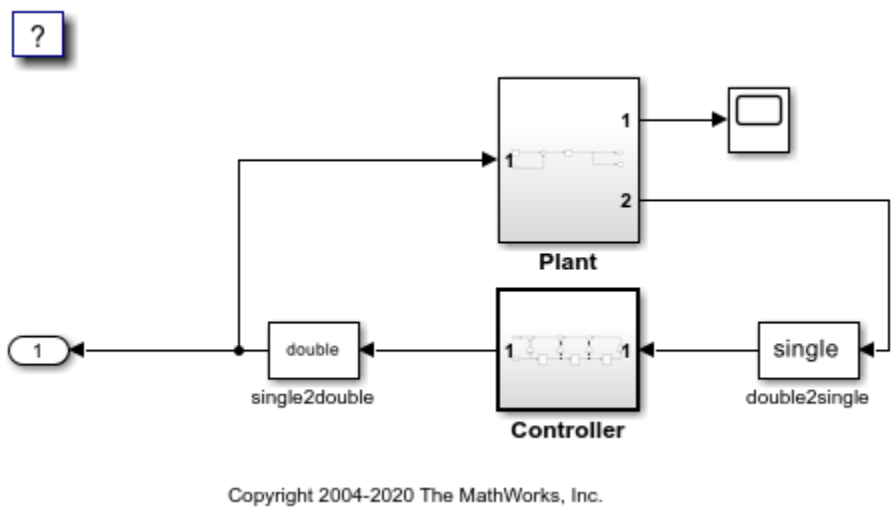
Test generated subsystem code by using a SIL or PIL block in a simulation. With this approach:

- You test code generated from subsystems, which uses the standalone code interface.

- You provide a test harness or a system model to supply test vector or stimulus inputs.
- You replace your original subsystem with the generated SIL or PIL block.

Open a simple model, which consists of a control algorithm and a plant model in a closed loop. The control algorithm regulates the output from the plant.

```
model='SILBlock';
close_system(model,0)
open_system(model)
```



Run a normal mode simulation

```
out = sim(model,10);
yout_normal = out.yout;
clear out
```

Configure the build process to create the SIL block for testing.

```
set_param(model,'CreateSILPILBlock','SIL');
```

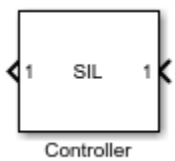
To test the behavior on production hardware, specify a PIL block.

To create the SIL block, generate code for the control algorithm subsystem. You see the SIL block at the end of the build process. Its input and output ports match those of the control algorithm subsystem.

```
close_system('untitled',0);
slbuild([model '/Controller'])
```

```
### Starting build procedure for: Controller
### Successful completion of build procedure for: Controller
### Creating SIL block ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.

Build Summary

Top model targets built:
```

```
Model       Action                        Rebuild Reason
==========================================================================================
Controller  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 23.23s
```



Alternatively, you can right-click the subsystem and select C/C++ Code > Build This Subsystem. In the dialog box that opens, click Build.

To perform a SIL simulation of the controller and plant model in a closed loop, replace the original control algorithm with the new SIL block. To avoid losing your original subsystem, do not save your model in this state.

```
controllerBlock = [model '/Controller'];
blockPosition = get_param(controllerBlock,'Position');
delete_block(controllerBlock);
add_block('untitled/Controller',[controllerBlock '(SIL)'],...
          'Position', blockPosition);
close_system('untitled',0);
clear controllerBlock blockPosition
```

Run the SIL simulation.

```
out = sim(model,10);
```

```
### Preparing to start SIL block simulation: <a href="matlab: targets_hyperlink_manager('run',2)
### Starting SIL simulation for component: SILBlock
### Application stopped
### Stopping SIL simulation for component: SILBlock
```

The control algorithm uses single-precision, floating-point arithmetic. You can expect the order of magnitude for differences between SIL and normal simulations to be close to the machine precision for single-precision data.

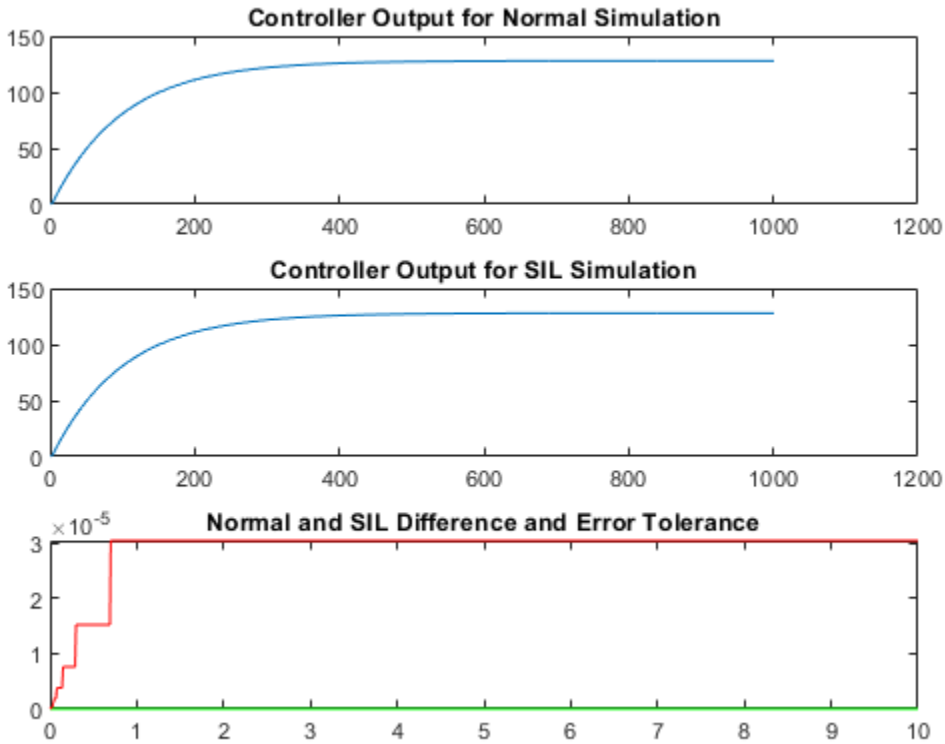
Define an error tolerance for SIL simulation results that is based on the machine precision for the single-precision, normal simulation results.

```
machine_precision = eps(single(yout_normal));
tolerance = 4 * machine_precision;
```

Compare normal and SIL simulation results. In the third plot, the differences between the simulations lie well within the defined error tolerance.

```
yout_sil = out.yout;
tout = out.tout;
fig1 = figure;
subplot(3,1,1), plot(yout_normal), title('Controller Output for Normal Simulation')
subplot(3,1,2), plot(yout_sil), title('Controller Output for SIL Simulation')
```

```
subplot(3,1,3), plot(tout,abs(yout_normal-yout_sil),'g-', tout,tolerance,'r-'), ...
    title('Normal and SIL Difference and Error Tolerance');
```



Clean up.

```
close_system(model,0);
if ishandle(fig1), close(fig1), end
clear fig1
simResults={'out','yout_sil','yout_normal','tout','machine_precision'};
save([model '_results'],simResults{:});
clear(simResults{:},'simResults')
```

Hardware Implementation Settings

When you run a SIL simulation, you must configure your hardware implementation settings (characteristics such as native word sizes) to allow compilation for your development computer. These settings can differ from the hardware implementation settings that you use when building the model for your production hardware. To avoid the need to change hardware implementation settings

between SIL and PIL simulations, enable portable word sizes. For more information, see “Configure Hardware Implementation Settings” on page 88-18.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Configure and Run SIL Simulation” on page 88-15
- “Configure Hardware Implementation Settings” on page 88-18
- “Create Execution-Time Profile for Generated Code” on page 81-2
- “Configure Code Coverage with Third-Party Tools” on page 91-9

Create a Target Communication Channel for Processor-in-the-Loop (PIL) Simulation

Implement a communication channel for processor-in-the-loop (PIL) simulation.

The communication channel enables exchange of data between different processes. The communication channel supports capabilities that require the exchange of data between the Simulink® software environment that runs on your development computer (host) and deployed code that runs on target hardware. For example, a PIL simulation.

You learn about the rtiostream interface and how it provides a generic communication channel that you can implement in the form of target connectivity drivers for different connection types. This example describes how to use the default TCP/IP implementation.

Two entities, Station A and Station B, use the rtiostream interface to set up a communication channel and exchange data. For this example, Station A and Station B are configured within the same process on your desktop computer.

The target connectivity drivers support an on-target PIL simulation. In the simulation, Station A and Station B represent the target and host computers that exchange data via the communication channel. On the host side, the target connectivity driver is implemented as a shared library that is loaded and called from within the MATLAB® product. On the target side, the driver is source code or a library that is linked to the application that runs on the target.

Additionally, you can:

- Configure your own target-side driver for TCP/IP to operate with the default host-side TCP/IP driver.
- Configure the supplied host-side driver for serial communications.
- Implement custom target connectivity drivers, for example, by using CAN or USB for host and target sides of the communication channel.

See also “Test Generated Code with SIL and PIL Simulations” on page 88-94 and “Configure Processor-In-The-Loop (PIL) for a Custom Target” on page 88-113.

View Source Code for the Default TCP/IP Implementation

The file rtiostream_tcpip.c implements client-side and server-side TCP/IP communication. A startup parameter configures the driver to operate in either client or server mode. You can use this source file as a starting point for a custom implementation. Each side of the communication channel requires only one or the other of the server or client implementations. If the client and server drivers run on different architectures, consider placing the driver code for each architecture in a separate source file.

The header file rtiostream.h contains prototypes for the functions rtIOStreamOpen/Send/Recv/Close. Include it (using #include) for custom implementations.

Extract the location of TCP/IP driver source code.

```
rtiostream_src_dir=fullfile(matlabroot,'toolbox','coder','rtiostream','src');
tcpip_dir=fullfile(rtiostream_src_dir,'rtiostreamtcpip');
```

View rtiostream_tcpip.c.

```
edit(fullfile(tcpip_dir,'rtiostream_tcpip.c'));
```

View rtiostream.h.

```
edit(fullfile(rtiostream_src_dir,'rtiostream.h'));
```

Location of Shared Library Files

To access the target connectivity drivers from the MATLAB product, you must compile them to a shared library. The shared library must be located on your system path. A shared library for the default TCP/IP drivers is located in matlabroot/bin/$ARCH ($ARCH is your system architecture, for example, win64).

The shared library filename extension and location depends on your operating system.

```
[~, ~, sharedLibExt] = coder.BuildConfig.getStdLibInfo;
```

Identify the shared library for Station A and Station B.

```
libTcpip = ['libmwrtiostreamtcpip' sharedLibExt];
disp(libTcpip)
```

```
libmwrtiostreamtcpip.dll
```

Test the Target Connectivity Drivers

If you are implementing a custom target connectivity driver, it is helpful to test it from within the MATLAB product. The following example shows how to load the default TCP/IP target connectivity drivers and use them for data exchange between Station A and Station B.

To access the drivers, you can use the MEX-file rtiostream_wrapper. With this MEX-file, you can load the shared library and access the rtiostream functions to open and close an rtiostream channel, and send and receive data.

Station A and Station B run on the host computer. Station A is configured as a TCP/IP server and Station B as a TCP/IP client. For host to target communication, you typically configure the host as a TCP/IP client and the target as a TCP/IP server.

Choose a port number for TCP.

```
if usejava('jvm')
    % Find a free port
    tempSocket = java.net.ServerSocket(0);
    port = num2str(tempSocket.getLocalPort);
    tempSocket.close;
else
    % Use a hard-coded port
    port = '14646';
end
```

Open the Station A rtiostream as a TCP/IP server.

```
stationA = rtiostream_wrapper(libTcpip,'open',...
                                  '-client', '0',...
                                  '-blocking', '0',...
                                  '-port',port);
```

If the communication channel opens, the return value is a handle to the connection. A return value of -1 indicates an error.

Check the return value.

```
assert(stationA~=(-1))
```

Open the Station B rtiostream as a TCP/IP client.

```
stationB = rtiostream_wrapper(libTcpip,'open',...
                                '-client','1',...
                                '-blocking', '0',...
                                '-port',port,...
                                '-hostname','localhost');
```

If the communication channel opens, the return value is a handle to the connection. A return value of -1 indicates an error.

Check the return value.

```
assert(stationB~=(-1))
```

Send Data from Station B to Station A

The target connectivity drivers send a stream of data in 8-bit bytes. For processors that are not byte-addressable, the data is sent in the smallest addressable word size.

Send message data from Station B to Station A.

```
msgOut = uint8('Station A, this is Station B. Are you there? OVER');

[retVal, sizeSent] = rtiostream_wrapper(libTcpip,...
                                        'send',...
                                        stationB,...
                                        msgOut,...
                                        length(msgOut));
```

A return value of zero indicates success.

```
assert(retVal==0);
```

Make sure that bytes in the message were sent.

```
assert(sizeSent==length(msgOut));
```

Allow time for data transmission to be completed.

```
pause(0.2)
```

Receive data in Station A.

```
[retVal, msgRecvd, sizeRecvd] = rtiostream_wrapper(libTcpip,...
                                                   'recv',...
                                                   stationA,...
                                                   100);
```

A return value of zero indicates success.

```
assert(retVal==0);
```

Make sure that bytes in the message were received.

```
assert(sizeRecvd==sizeSent);
```

Display the received data.

```
disp(char(msgRecvd))
```

```
Station A, this is Station B. Are you there? OVER
```

Send a Response from Station A to Station B

Send response data from Station A to Station B.

```
msgOut = uint8('Station B, this is Station A. Yes, I''m here! OVER.');
[~, sizeSent] = rtiostream_wrapper(libTcpip,... %#ok
                                   'send',...
                                   stationA,...
                                   msgOut,...
                                   length(msgOut));
```

Allow time for data transmission to be completed.

```
pause(0.2)
```

Receive data in Station B.

```
[~, msgRecvd, sizeRecvd] = rtiostream_wrapper(libTcpip,... %#ok
                                              'recv',...
                                              stationB,...
                                              100);
```

Display the received data.

```
disp(char(msgRecvd))
```

```
Station B, this is Station A. Yes, I'm here! OVER.
```

Close Connection and Unload Shared Libraries

Close rtiostream on Station B.

```
retVal = rtiostream_wrapper(libTcpip,'close',stationB);
```

A return value of zero indicates success.

```
assert(retVal==0);
```

Close rtiostream on Station A.

```
retVal = rtiostream_wrapper(libTcpip,'close',stationA);
```

A return value of zero indicates success.

```
assert(retVal==0)
```

Unload the shared library.

```
rtiostream_wrapper(libTcpip, 'unloadlibrary');
```

Host-Side Driver for Serial Communications

You can use the supplied host-side driver for serial communications as an alternative to the drivers for TCP/IP. To configure the serial driver, see `rtiostream_wrapper` in the Embedded Coder® reference documentation.

Configure Your Own Target-Side Driver

If your target has an Ethernet connection and you have a TCP/IP stack available, follow these steps:

1. Write a wrapper for your TCP/IP stack that makes it available via the rtiostream interface defined in rtiostream.h.
2. Write a test application for your target that sends and receives some data.
3. Use the rtiostream_wrapper MEX-file and host-side TCP/IP driver to test your driver software that is running on the target.
4. When you have a working target-side driver, include the driver source files in the build for your automatically generated code.

You can configure your target-side driver to operate only as a TCP/IP server because the default host-side driver for PIL is configured as a TCP/IP client.

If you need to use a communications channel that is not already supported on the host-side, write drivers for host and target. In this case, you can still use the rtiostream_wrapper MEX-file for testing your rtiostream drivers.

Configure Your Own Host-Side Driver

You can implement the target connectivity drivers by using different communication channels. For example, you can implement host-target communications via a special serial connection, which requires that you provide drivers for the host and target.

On the host side, you can test the drivers by using the rtiostream_wrapper MEX-file. If your driver includes diagnostic output from printf statements and rtiostream_wrapper loads the shared library, you must replace the printf statements with mexPrintf statements.

When you have a working host-side device driver, you must make it available within the Simulink software environment. For PIL simulation, register the shared host-side shared library via sl_customization.

See Also

Related Examples

- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Host-Target Communication for Simulink PIL Simulation” on page 88-42
- “Test Generated Code with SIL and PIL Simulations” on page 88-94

Configure Processor-In-The-Loop (PIL) for a Custom Target

Create a target connectivity configuration by using target connectivity APIs. With a target connectivity configuration, you can run processor-in-the-loop (PIL) simulations on custom embedded hardware.

You learn how to:

- Adapt the build process to support PIL simulations.
- Configure a tool for downloading and starting execution of a PIL executable on your target hardware.
- Configure a communication channel between host and target to support PIL simulation on the target hardware.

Start with a model configured for software-in-the-loop (SIL) simulation. This example guides you through the process of creating a target connectivity configuration so that you can simulate the model in PIL mode. The example runs entirely on your development computer. You can use the same steps to create a connectivity configuration for your own embedded target hardware.

See also “Test Generated Code with SIL and PIL Simulations” on page 88-94 and “Create a Target Communication Channel for Processor-in-the-Loop (PIL) Simulation” on page 88-108.

Preliminaries

Later in this example, you will add a folder to the search path. Create the folder path.

```
sl_customization_path = pwd;
```

If this folder is already on the search path, remove it.

```
if contains(path,sl_customization_path)
    rmpath(sl_customization_path)
end
```

Reset customizations.

```
sl_refresh_customizations
```

Test Generated Code with SIL Simulation

Simulate a model configured for SIL. Verify the generated code compiled for your development computer by comparing the SIL simulation behavior with the normal simulation behavior.

Make sure that the example model is newly opened.

```
close_system('SILModelBlock',0);
close_system('SILCounter',0)
open_system('SILModelBlock')
```

The CounterA Model block displays the text (SIL), which indicates that its referenced model is configured for SIL simulation.

Run a simulation of the system.

```
set_param('SILModelBlock','StopTime','10');
sim('SILModelBlock');

### Starting build procedure for: SILCounter
### Successful completion of build procedure for: SILCounter

Build Summary

Top model targets built:

Model       Action                        Rebuild Reason
========================================================================================
SILCounter  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 18.012s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: SILCounter
### Application stopped
### Stopping SIL simulation for component: SILCounter
```

Target Connectivity Configuration

Start work on a target connectivity configuration for PIL.

Make a local copy of the target connectivity configuration classes.

```
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder','+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');
```

Make the copied files writable.

```
fileattrib(fullfile('+mypil', '*'),'+w');
```

Update the package name to reflect the new location of the files.

```
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

Verify that you now have a folder +mypil in the current folder, which has the files Launcher.m, TargetApplicationFramework.m, and ConnectivityConfig.m.

```
dir './+mypil'
```

```
.                               ..                              ConnectivityConfig.m          Launche
```

Review Code to Launch the PIL Executable

The class that configures a tool for launching the PIL executable is mypil.Launcher. Open this class in the editor.

```
edit(which('mypil.Launcher'))
```

Review the content of this file. The method setArgString supplies additional command-line parameters to the executable. These parameters can include a TCP/IP port number. For an embedded processor implementation, you can choose to hard-code these settings.

Configure the Overall Target Connectivity Configuration

View the class mypil.ConnectivityConfig.

```
edit(which('mypil.ConnectivityConfig'))
```

Review the content of this file. You should be able to identify:

- The creation of an instance of rtw.connectivity.RtIOStreamHostCommunicator that configures the host side of the TCP/IP communications channel.
- A call to the setArgString method of Launcher that configures the target side of the TCP/IP communications channel.
- A call to setTimer that configures a timer for execution time measurement

To define your own target-specific timer for execution time profiling, you must use the Code Replacement Library to specify a replacement for the function code_profile_read_timer. Use a command-line API or the crtool user interface.

Review the Target-Side Communications Drivers

View the file rtiostream_tcpip.c.

```
rtiostreamtcpip_dir=fullfile(matlabroot,'toolbox','coder','rtiostream','src',...
    'rtiostreamtcpip');
edit(fullfile(rtiostreamtcpip_dir,'rtiostream_tcpip.c'))
```

Scroll down to the end of this file. See that this file contains a TCP/IP implementation of the functions rtIOStreamOpen, rtIOStreamSend, and rtIOStreamRecv. These functions are required for the target hardware to communicate with your development computer. You must provide an implementation for each of these functions that is specific to your target hardware and communication channel.

Add Target-Side Communications Drivers to the Connectivity Configuration

The class that configures additional files to include in the build is mypil.TargetApplicationFramework. Open this class in the editor.

```
edit(which('mypil.TargetApplicationFramework'))
```

Use sl_customization to Register the Target Connectivity Configuration

To use the new target connectivity configuration, you must provide an sl_customization file. The sl_customization file registers your new target connectivity configuration and specifies the required conditions for its use. The conditions specified in this file can include the name of your system target file and your hardware implementation settings.

You can view the sl_customization file. For this example, you do not have to make changes to the file.

```
edit(fullfile(sl_customization_path,'sl_customization.m'))
```

Add the sl_customization folder to the search path and refresh the customizations.

```
addpath(sl_customization_path);
sl_refresh_customizations;
```

Test Generated Code with PIL Simulation

Run the PIL simulation.

```
close_system('SILModelBlock',0)
open_system('SILModelBlock')
set_param('SILModelBlock/CounterA','SimulationMode','processor-in-the-loop (pil)');
set_param('SILModelBlock','StopTime','10');
sim('SILModelBlock');
```

```
### Starting build procedure for: SILCounter
### Generated code for 'SILCounter' is up to date because no structural, parameter or code replac
### Successful completion of build procedure for: SILCounter

Build Summary

Top model targets built:

Model       Action          Rebuild Reason
========================================================================
SILCounter  Code compiled.  Compilation artifacts were out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 3.1522s
### Connectivity configuration for component "SILCounter": My PIL Example ###
EXECUTING METHOD SETARGSTRING
SETARGSTRING called from line 65 of ConnectivityConfig.m
### Preparing to start PIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with PIL files ...
### Starting application: 'SILCounter_ert_rtw\pil\SILCounter.exe'
Starting PIL simulation
Started new process, pid = 12484
Stopping PIL simulation
Terminated process, pid = 12484
```

Review the preceding messages. Confirm that the simulation ran without errors. You have now implemented a target connectivity configuration for PIL. You can use the same APIs to implement a connectivity configuration for your own combination of embedded processor, download tool, and communications channel.

Cleanup

Remove the search path for this example.

```
rmpath(sl_customization_path)
```

Reset customizations.

```
sl_refresh_customizations
```

Close the models.

```
close_system('SILModelBlock',0)
close_system('SILCounter',0)
```

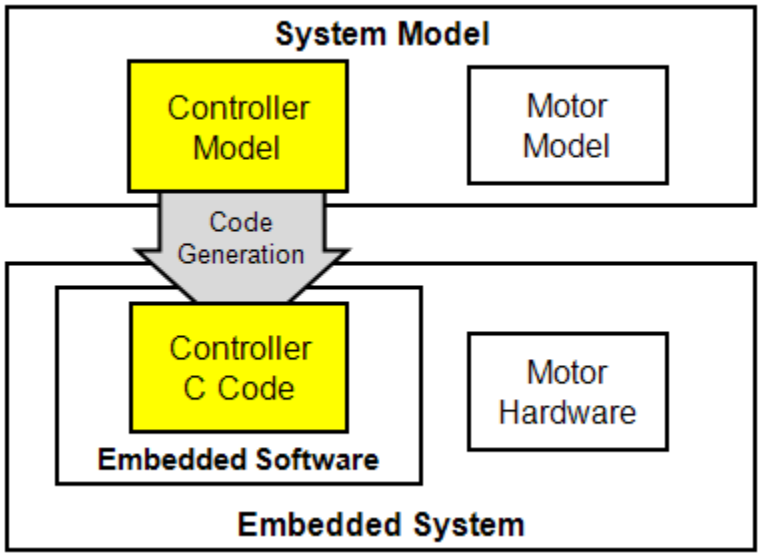
See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Configure and Run PIL Simulation” on page 88-24

Field-Oriented Control of Permanent Magnet Synchronous Machine

This example shows the basic workflow and key APIs for generating C code from a motor control algorithm, and for verifying its compiled behavior and execution time. You will use processor-in-the-loop (PIL) simulation to gain confidence that the C code will perform as expected when you integrate it with embedded software that interfaces with the motor hardware. Although this workflow uses a motor control application for a specific processor, you can apply this workflow to another application or processor.

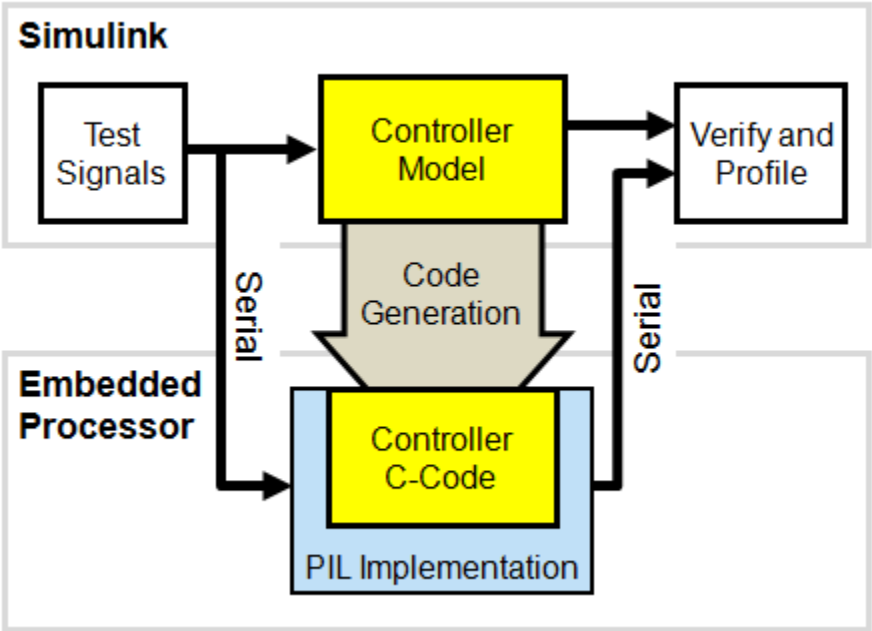


We use a Field-Oriented Control algorithm for a Permanent Magnet Synchronous Machine to show this workflow. This control technique is common in motor drive systems for hybrid electric vehicles, manufacturing machinery, and industrial automation.

Overview

In this example, generate and verify C code from a control algorithm model, which you can integrate with additional embedded software required to interface to motor hardware.

You will use a simulation environment to model and verify the behavior of a closed-loop motor control system. Once the control system behavior is within specification, you will generate C code from the controller model. After inspecting the code, you will assess its functional behavior and execution time using processor-in-the-loop (PIL) testing.



To facilitate PIL testing, you will select test signals to exercise the controller model and establish reference outputs. You will review an example PIL implementation for a Texas Instruments™ F28335 processor that communicates with Simulink® on the host computer over a serial connection. You can use this example as a starting point to create a PIL implementation for your own processor. You will run the controller model in PIL mode to measure execution time and verify execution behavior of the code running on the embedded processor against the simulation reference outputs.

In the final implementation on the embedded processor, you would integrate the generated controller C code with additional embedded software, such as peripherals and interrupts, required to interface with the motor hardware.

Notes

- Simscape™ Electrical™ is required for system simulation in the section "Verify Behavior through System Simulation." It is not required for other tasks.
- The Texas Instruments F28335 PIL implementation is a reference approach that you can apply to virtually any processor. However, if you wish to use this implementation directly, you will need additional support files, compilers, and tools from Texas Instruments. You can find more information about those in "Create PIL Implementation and Register with Simulink" section of this example. This reference PIL implementation does not require the Texas Instruments C2000™ Embedded Target feature of Embedded Coder®, but C2000 users are encouraged to install Texas Instruments C2000 Support Package using Add-On Explorer.

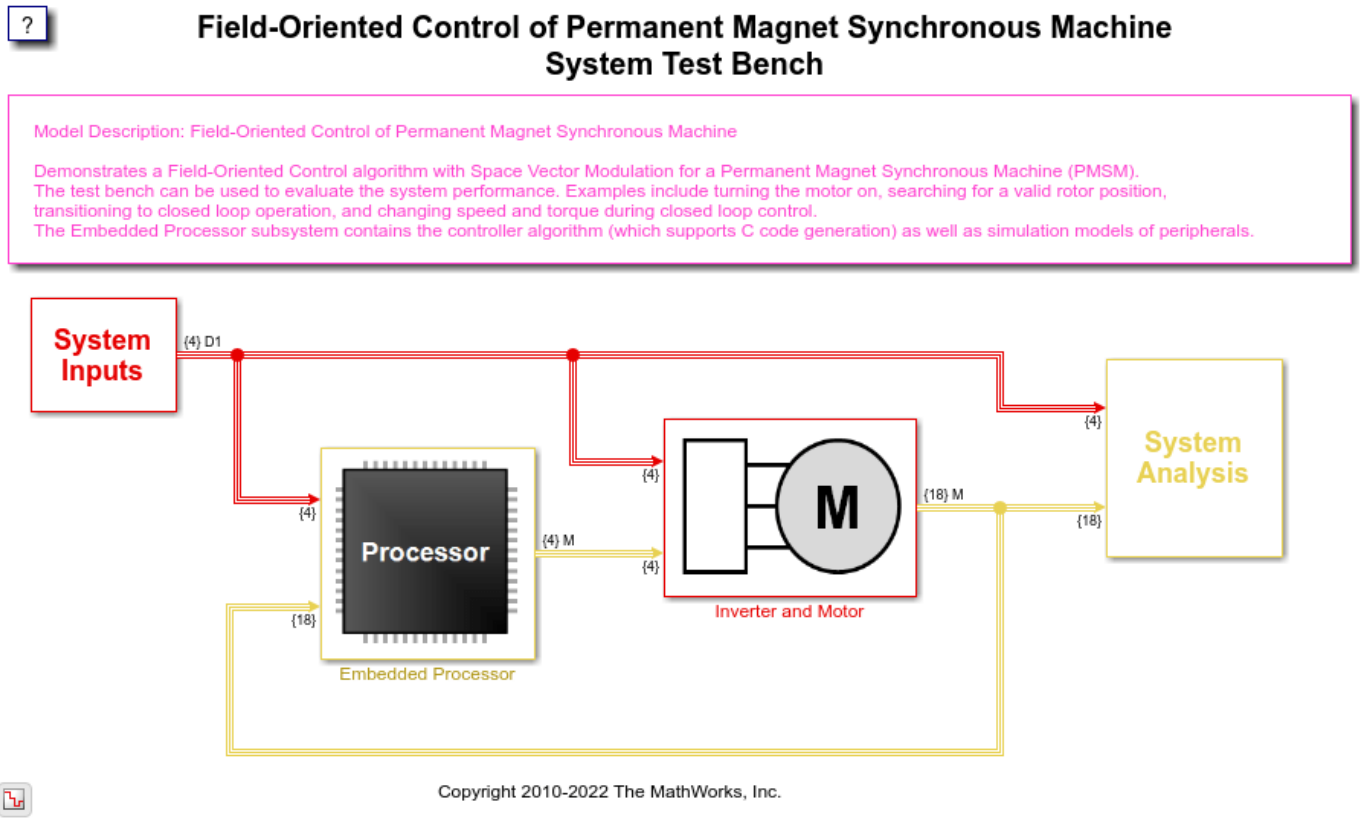
Verify Behavior Through System Simulation

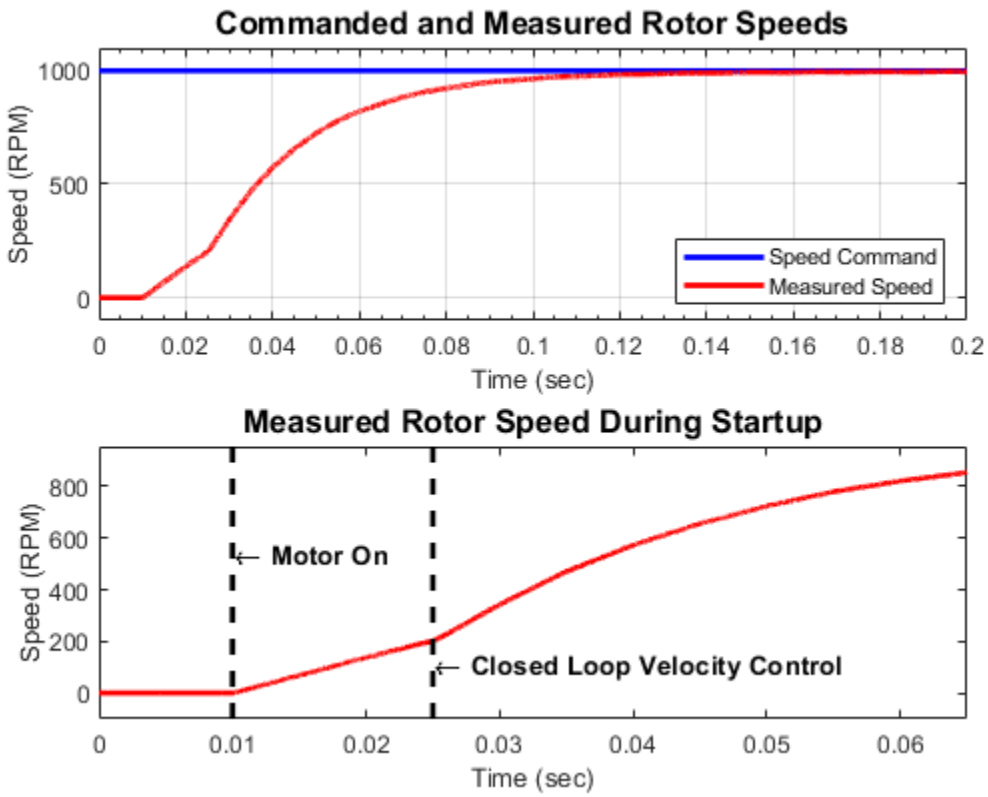
In this section, you will verify the controller in a closed loop system simulation.

The system model test bench consists of test inputs, the embedded processor, the power electronics and motor hardware, and visualizations. You can use the system model to exercise the controller and explore its expected behavior. You can use the following commands to execute the model and plot logged signals.

```
open_system('rtwdemo_pmsmfoc_system')
out_system = sim('rtwdemo_pmsmfoc_system')
rtwdemo_pmsmfoc_plotsignals(out_system.logsout)
```

```
out_system =

  Simulink.SimulationOutput:
                 logsout: [1x1 Simulink.SimulationData.Dataset]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```





The plots show that the motor is stationary until the `motor_on` signal is true. The motor then spins in an open loop until a known position is detected, which is indicated by the encoder index pulse. The controller then transitions to closed loop operation and the motor reaches a steady state speed.

Explore Model Architecture

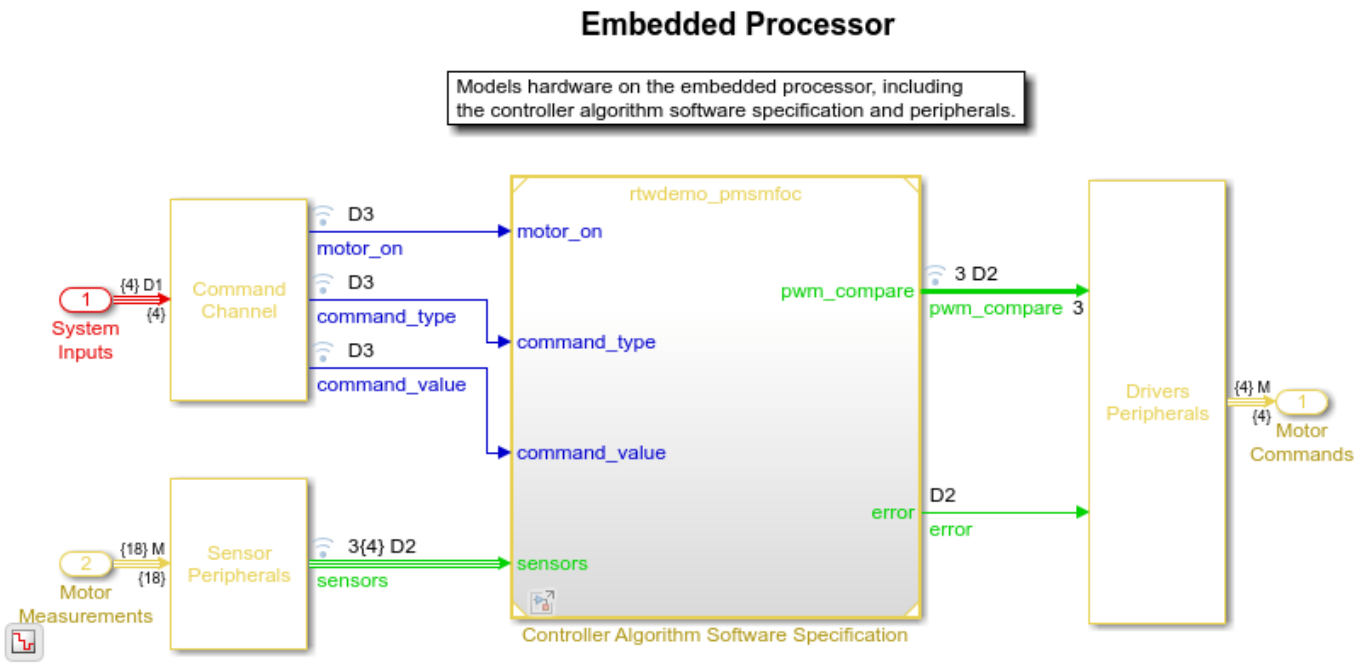
In this section, you will explore the model architecture including how data is specified, how the controller is partitioned from the test bench, and how controller is scheduled. This architecture facilitates system simulation, algorithmic code generation, and PIL testing.

A data definition file creates the MATLAB® data required for simulation and code generation. This data file is automatically run within the `PreLoadFcn` callback of the system test bench model.

```
edit('rtwdemo_pmsmfoc_data.m')
```

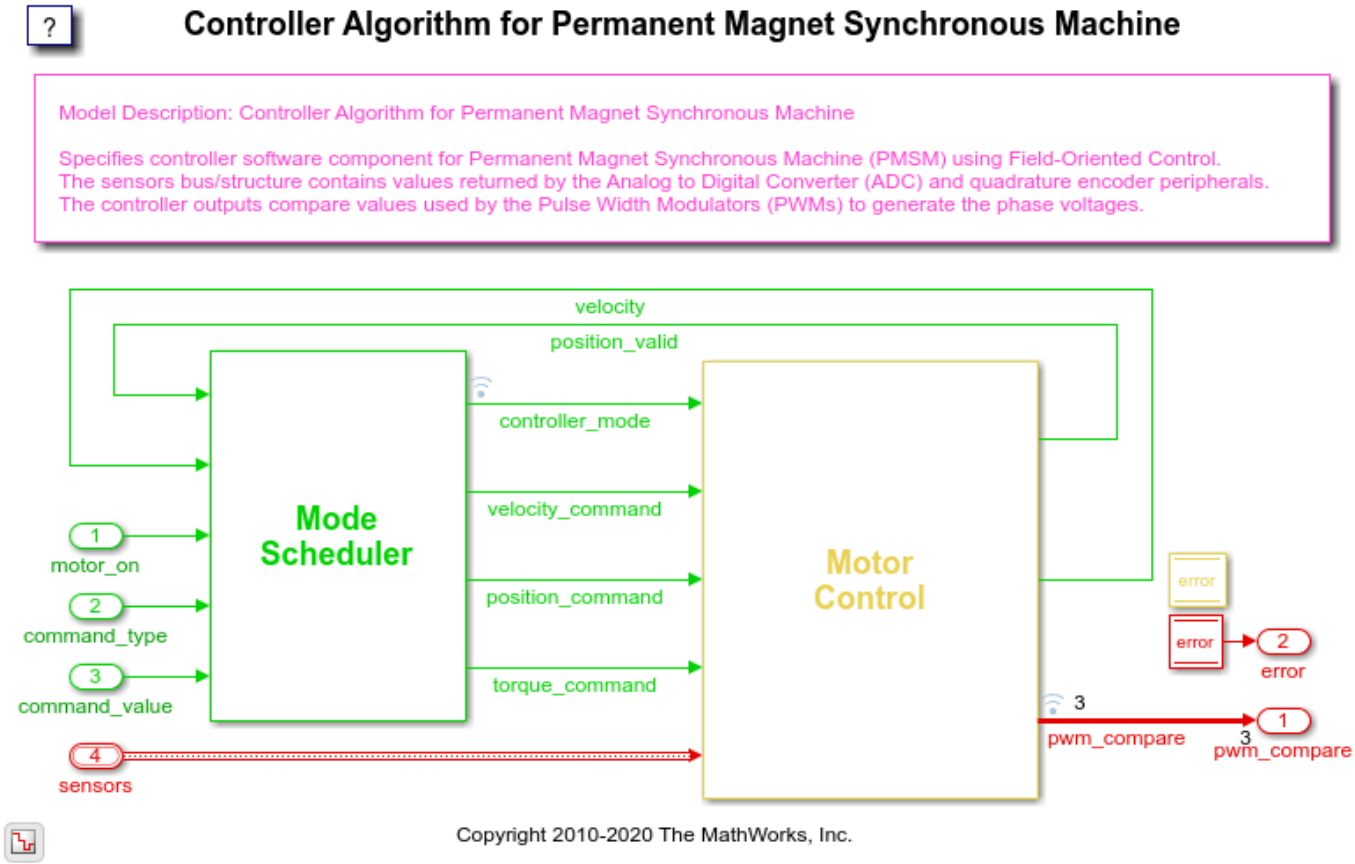
Within the system test bench model, the embedded processor is modeled as a combination of the peripherals and the controller software.

```
open_system('rtwdemo_pmsmfoc_system/Embedded Processor')
```



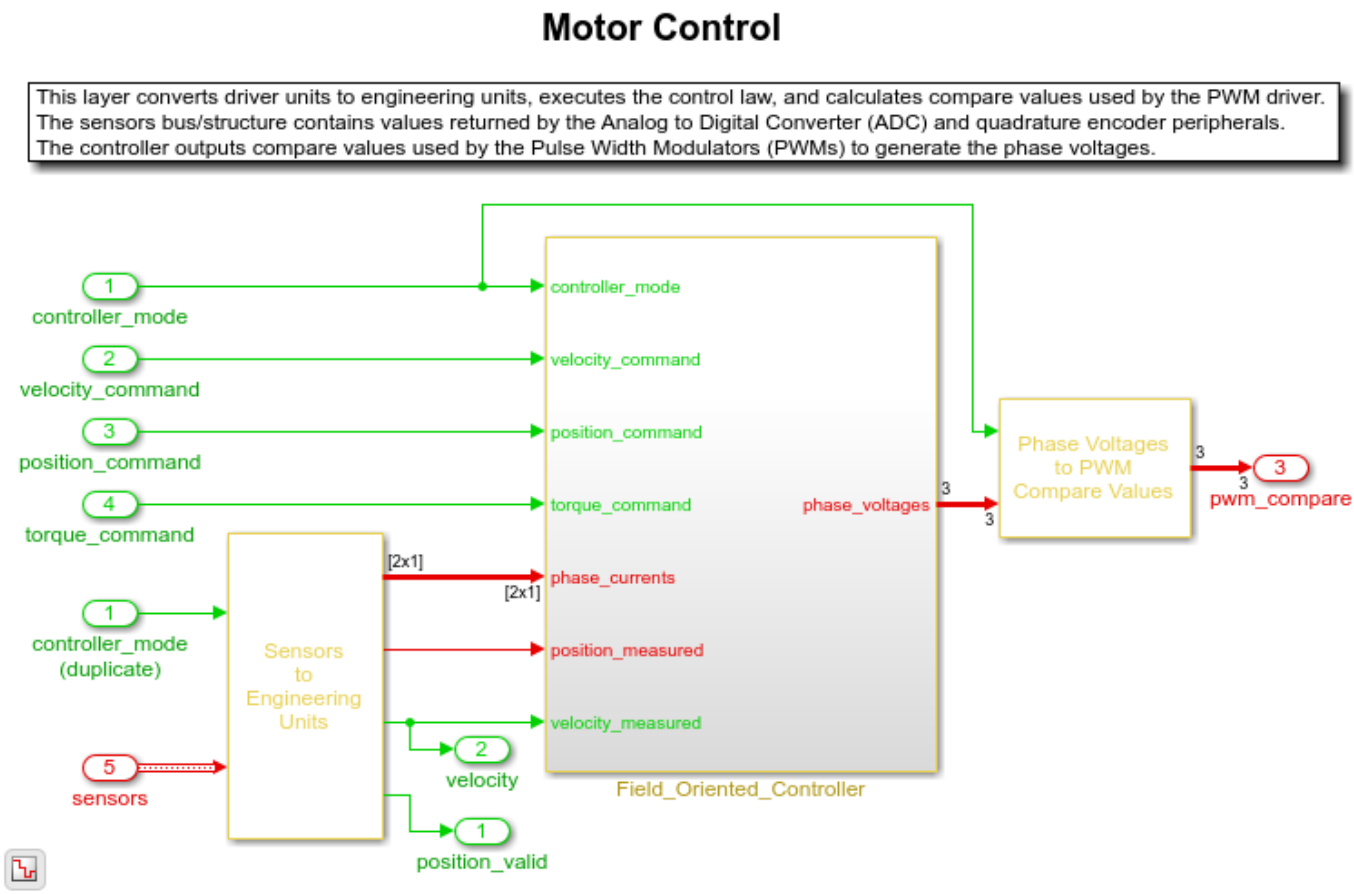
The controller software is specified in a separate model. Within this model, the Mode_Scheduler subsystem uses Stateflow® to schedule different modes of operation of the Motor_Control algorithm.

```
open_system('rtwdemo_pmsmfoc')
```



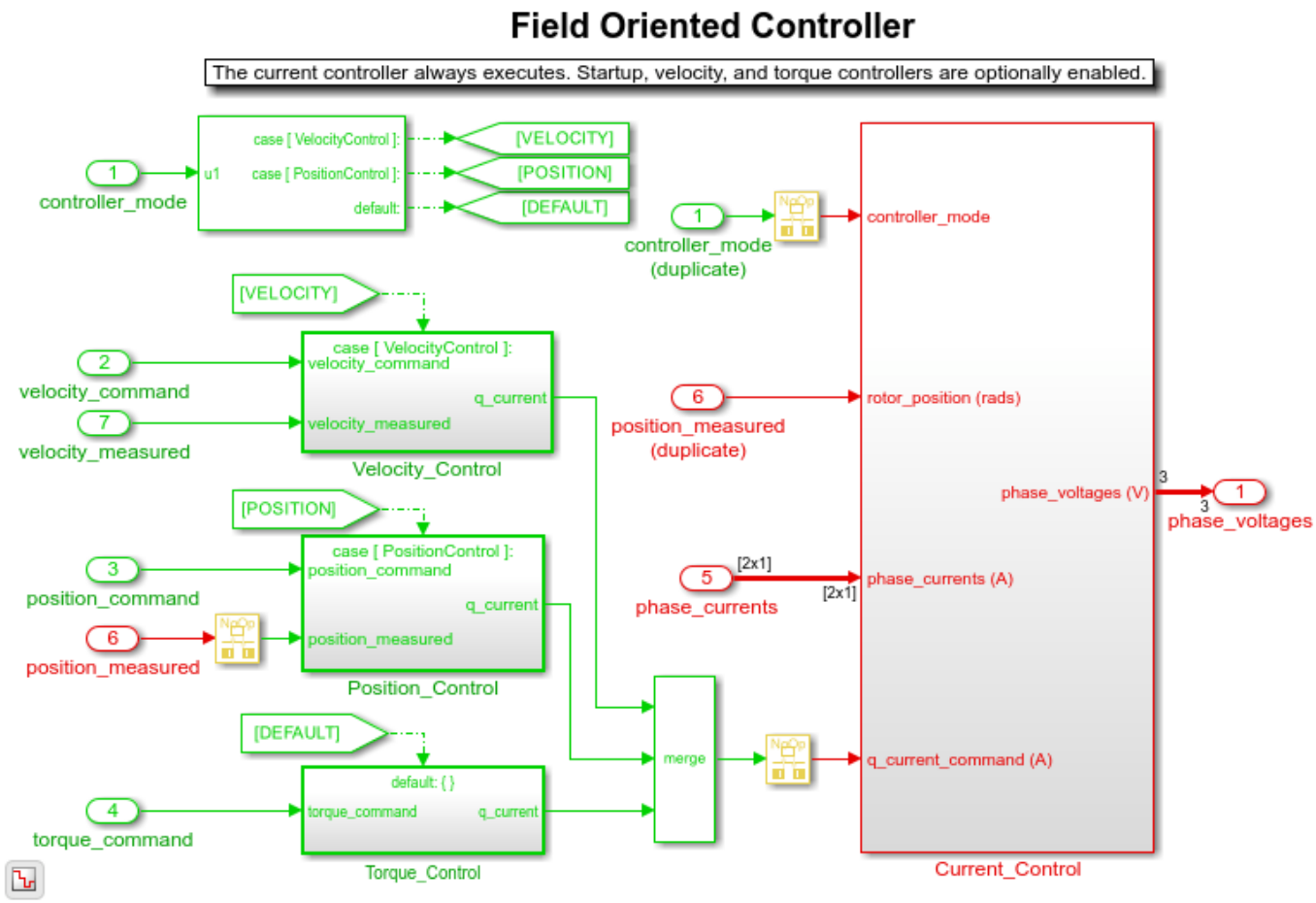
Within the Motor_Control subsystem, sensor signals are converted to engineering units and passed to the core controller algorithm. The controller algorithm calculates voltages. The voltages are then converted to a driver signal.

```
open_system('rtwdemo_pmsmfoc/Motor_Control')
```



The primary control law is a field-oriented controller. The controller has a low rate outer loop that controls the speed, and a higher rate inner loop that controls the current.

```
open_system('rtwdemo_pmsmfoc/Motor_Control/Field_Oriented_Controller')
```



The Velocity Controller outer loop is executed as a multiple of the Current Control loop time. You can view the MATLAB variables, which specify these sample times:

```
fprintf('High rate sample time = %f seconds\n', ctrlConst.TsHi)
fprintf('Low rate sample time  = %f seconds\n', ctrlConst.TsLo)
```

```
High rate sample time = 0.000040 seconds
Low rate sample time  = 0.005000 seconds
```

Notice that the highest rate in the controller algorithm is 25 kHz.

```
fprintf('High rate frequency = %5.0f Hz\n', 1/ctrlConst.TsHi)
```

```
High rate frequency = 25000 Hz
```

Generate Controller C Code for Integration into Embedded Application

In this section, you will generate and visually inspect the C code function for the controller.

To ease integration, the controller model is configured in single-tasking mode so that the generated code can be invoked using one function call. This function handles the lower and higher rates. The generated controller function must be executed at the high rate sample time.

The function prototype is specified in the model configuration parameters and the input and output ports are passed as arguments. You can view the function specification for the controller algorithm.

```
mdlFcn = RTW.getFunctionSpecification('rtwdemo_pmsmfoc');
disp(mdlFcn.getPreview('init'))
disp(mdlFcn.getPreview('step'))
```

```
Controller_Init ( )
error = Controller ( motor_on, command_type, current_request, * sensors, * pwm_compare )
```

Through use of a global structure in the generated code, you can access the field-oriented controller proportional and integral gains. This global structure is specified in the data definition file.

```
disp(ctrlParams.Value)
disp(ctrlParams.CoderInfo)
```

```
                            Current_P: 10
                            Current_I: 10000
                           Velocity_P: 0.0050
                           Velocity_I: 0.0150
                           Position_P: 0.1000
                           Position_I: 0.6000
                   StartupAcceleration: 1
                        StartupCurrent: 0.2000
                     RampToStopVelocity: 20
             AdcZeroOffsetDriverUnits: 2.2522e+03
                 AdcDriverUnitsToAmps: 0.0049
    EncoderToMechanicalZeroOffsetRads: 0
                       PmsmPolePairs: 4

Simulink.CoderInfo
    StorageClass: 'ExportedGlobal'
      Identifier: ''
       Alignment: -1
```

You generate C code from the model as follows.

```
slbuild('rtwdemo_pmsmfoc')
```

```
### Starting build procedure for: rtwdemo_pmsmfoc
### Successful completion of build procedure for: rtwdemo_pmsmfoc

Build Summary

Top model targets built:

Model            Action                        Rebuild Reason
========================================================================================
rtwdemo_pmsmfoc  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 12.342s
```

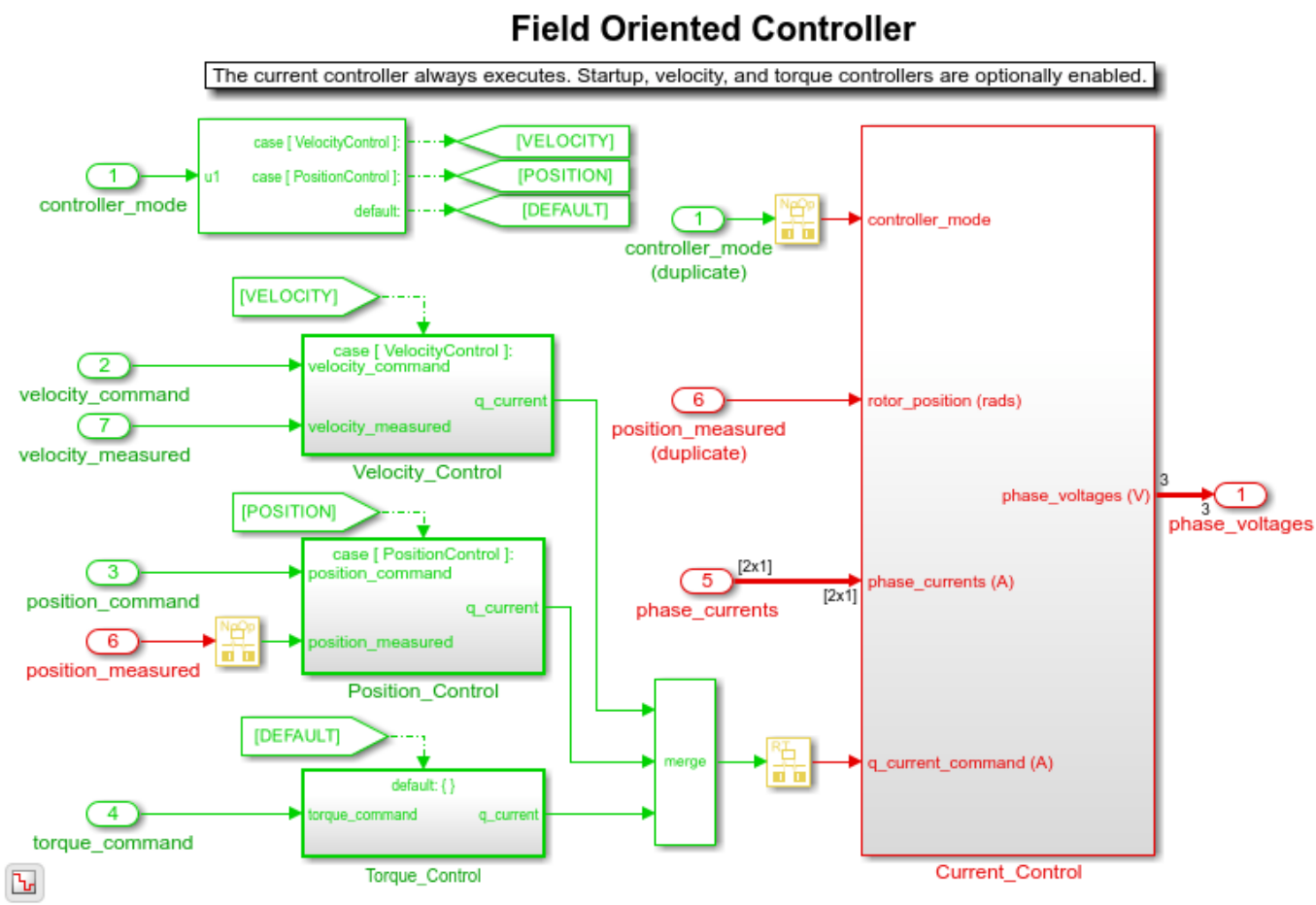
Use the generated report to inspect the generated C code file and verify that the correct step and initialization functions were generated. Also verify that the parameter structure was created as a global variable.

Establish Reference Behavior for Controller Model

In this section, you will establish test inputs and a reference output to help verify behavior and profile execution time during PIL testing. You will make a local copy of the controller model then load a set of test input signals that exercise the different modes within the controller. You will then configure the controller model to attach these logged signals to input ports, execute the controller model, and log the output port signal to the workspace.

Configuration parameters of the controller model used to establish the reference behavior and test environment will change as described below. Blocks and parameters used to specify the controller model's design and generate production code will not change. However, to avoid modifying any part of the installed controller model, save the model and change its name to `rtwdemo_pmsmfoc_local.slx`.

```
save_system('rtwdemo_pmsmfoc','rtwdemo_pmsmfoc_local.slx')
close_system('rtwdemo_pmsmfoc_system',0);
close_system('rtwdemo_pmsmfoc',0);
```



To profile execution times, select a set of test inputs that will execute the paths of interest within the controller. One way to acquire these test inputs and reference output is to log them from the system simulation model.

```
in.motor_on      = out_system.logsout.getElement('motor_on').Values;
in.command_type  = out_system.logsout.getElement('command_type').Values;
in.command_value = out_system.logsout.getElement('command_value').Values;
in.sensors       = out_system.logsout.getElement('sensors').Values;
display(in)
```

```
in =

  struct with fields:

        motor_on: [1×1 timeseries]
    command_type: [1×1 timeseries]
   command_value: [1×1 timeseries]
         sensors: [1×1 struct]
```

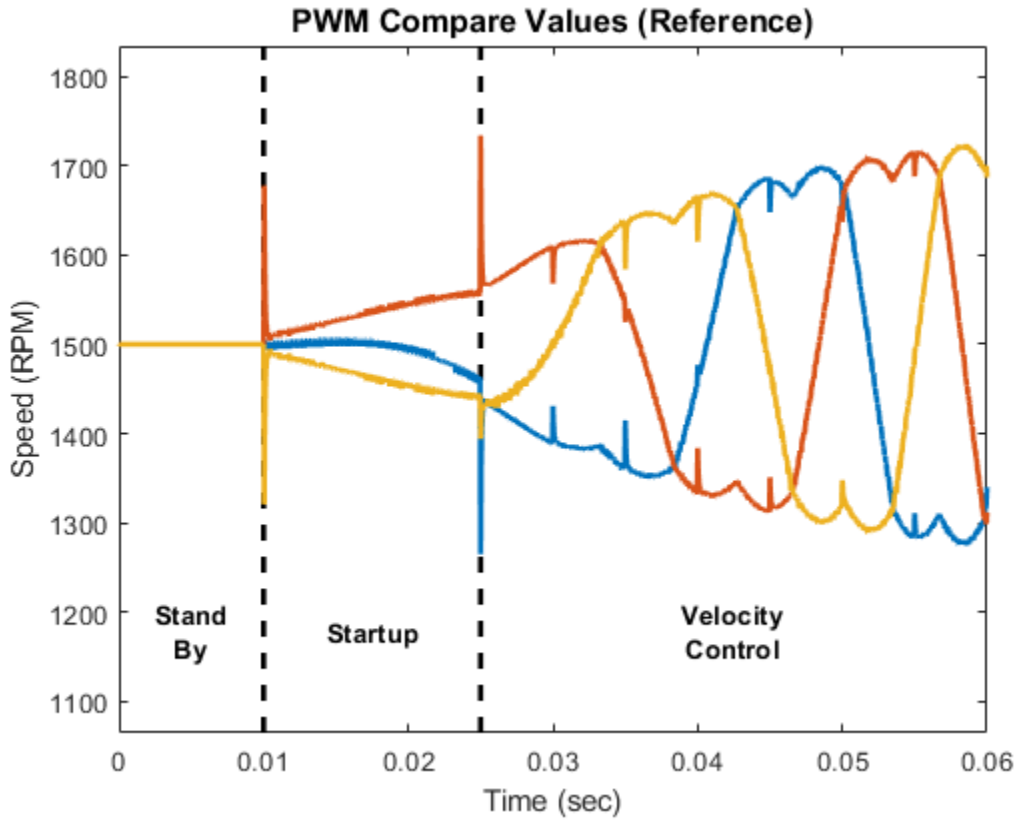
These signals can now be attached to the input ports and imported into the controller model such that it can execute directly and independently from the system model. An advantage of this approach is that the controller model can be tested and verified as a standalone component, facilitating reuse and integration with other system models or closed-loop test benches. To elaborate, or prepare, the controller model for testing, change its Configuration Parameters to attach input signals and log signals in the MATLAB workspace. These changes can be made graphically in the model's Configuration Parameters dialog, or programmatically as shown below.

```
set_param('rtwdemo_pmsmfoc_local',...
    'LoadExternalInput', 'on',...
    'ExternalInput', 'in.motor_on, in.command_type, in.command_value, in.sensors',...
    'StopTime','0.06',...
    'ZeroInternalMemoryAtStartup','on',...
    'SimulationMode', 'normal')
save_system('rtwdemo_pmsmfoc_local.slx')
```

You can now execute the controller model and plot the signal associated with the PWM Compare output port.

```
out = sim('rtwdemo_pmsmfoc_local')
controller_mode = out.logsout.getElement('controller_mode').Values;
pwm_compare_ref = out.logsout.getElement('pwm_compare').Values;
rtwdemo_pmsmfoc_plotpwmcompare(controller_mode, pwm_compare_ref)
```

```
out =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```



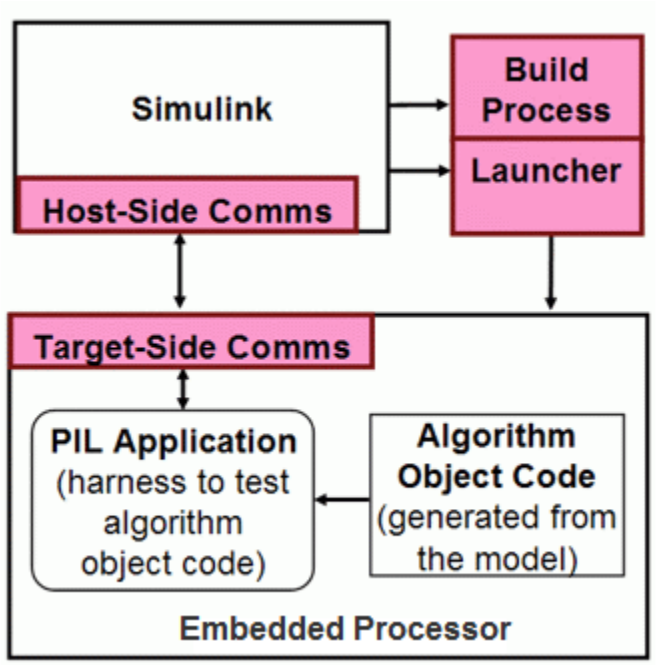
The logged output will be used as the reference behavior for PIL testing.

Notice that the plot is annotated with information about the mode of the controller at each time step. This mode information will be useful when interpreting execution profiling information.

Create PIL Implementation

In this section, you will study and use an example PIL implementation. You will begin by reviewing prerequisite help documentation from Embedded Coder. You will then copy the example PIL implementation into your local directory and register it with Simulink. You will review the approach used to develop the PIL implementation and can explore the associated files to gain additional insight. If you are using a Spectrum Digital Inc. eZdsp F28335 board with Code Composer v4 and a serial connection, you will be able to configure this PIL implementation to directly work with the controller model. If you are using a different processor, you can use this PIL implementation as a starting point to create your own implementation.

The fundamentals of creating a custom PIL implementation are described in “Create PIL Target Connectivity Configuration for Simulink” on page 88-38 . You should familiarize yourself with the basic concept of using the rtiostream API to facilitate communication between Simulink (host-side) and the embedded processor (target-side) during PIL testing. Note that Embedded Coder provides host-side drivers for the default TCP/IP implementation (for all platforms supported by Simulink) as well as a Windows® only version for serial communication. Building the generated code is accomplished using a makefile as described in “Customize Template Makefiles” . To create a PIL implementation you will need to perform several tasks in the embedded environment including writing a target-side communications driver, writing a makefile to build the generated code, and automating download and execution of the built executable.



Using the above approach, a PIL implementation was created for a Spectrum Digital Inc. eZdsp F28335 board. Below is a summary of the target connectivity API components used in this implementation.

- **Host-Side Communication** - The host-side connectivity driver is configured to use serial communication.
- **Target-Side Communication** - The target-side communication is achieved using a handwritten serial implementation of the rtiostream functions as well as timer access functions.
- **Build Process** - A makefile based approach is used to build the executable application.
- **Launcher** - Downloading and running the executable is accomplished using the Debug Server Scripting (DSS) utility of Code Composer Studio™ v4 (CCSv4).

The PIL implementation was iteratively developed in three stages. Below is a description of these stages and the tasks performed in these stages. You may find it helpful to follow a similar approach when developing your own PIL implementation

Stage 1: Create serial communication application with CCSv4

- Install CCSv4 and verify that it can connect with the F28335 eZdsp board.
- Write an embedded application that sends and receives serial data.
- Test the serial communication between the host computer and the embedded application.
- Identify the commands and options for the compiler, linker, and archiver to build the application using a makefile.
- Download and run the application from a Windows Command Prompt using the DSS utility.

Stage 2: Implement and test embedded serial rtiostream and launch automation with MATLAB

- Extend the serial application to implement the rtiostream API functions for echoing data. Write the rtIOStreamOpen to perform general board initialization, including configuring the serial port.

- Verify sending and receiving serial data with the embedded processor from MATLAB using the rtiostream_wrapper function.
- Download and run the application from MATLAB using the system command to call the DSS utility.

Stage 3: Implement and test connectivity configuration with Simulink

- Create a connectivity configuration class to configure the host-side serial communication, specify which target-side code files from the rtiostream application should be included in the build process, specify how to access a timer that will be used to collect profiling data, and integrate calling the DSS utility to launch the embedded application.
- Create a tool specification makefile (`target_tools.mk`) which specifies the commands and options for the compiler, linker, and archiver. This makefile is included by the template makefile (`target_tools.mk`).
- Create a template makefile (`ec_target.tmf`) that includes `target_tools.mk`.
- Identify parameters which may be installation dependent and store them as MATLAB preferences.
- Create a Simulink customization file which specifies when the PIL implementation is valid.

The files associated with this PIL implementation are included with Embedded Coder, but are not on the MATLAB path. To explore these files, you can copy them into a local directory. You can register this PIL implementation by adding this directory to the MATLAB path and refreshing the Simulink customizations.

```
%copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','examplePilF28335'),'examplePilF28335',
addpath(genpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','examplePilF28335')))
sl_refresh_customizations
```

MATLAB preferences are used to specify path information and the host serial COM port number. If you are using this PIL implementation directly, you must specify these preferences as appropriate for your configuration.

Note that the `TI_F28xxx_SysSWDir` preference points to a directory provided by Texas Instruments in their **C2000 Experimenter Kit Application Software** (`sprc675.zip`). These files are not included with Embedded Coder.

```
setpref('examplePilF28335','examplePilF28335Dir', fullfile(matlabroot,'toolbox','rtw','rtwdemos'
setpref('examplePilF28335','CCSRootDir',          'C:\Program Files\Texas Instruments\ccsv4');
setpref('examplePilF28335','TI_F28xxx_SysSWDir',  'C:\Program Files\Texas Instruments\TI_F28xxx_S
setpref('examplePilF28335','targetConfigFile',    fullfile(matlabroot,'toolbox','rtw','rtwdemos'
setpref('examplePilF28335','baudRate',            115200);
setpref('examplePilF28335','cpuClockRateMHz',     150);
setpref('examplePilF28335','boardConfigPLL',      10);
setpref('examplePilF28335','COMPort',             'COM4');
```

The PIL implementation is now ready to be used.

Prepare Controller Model for PIL Testing

In this section you will configure the controller model to use the PIL implementation. You will review the customization file used to register the PIL implementation, set the configuration parameters of the model to use the PIL implementation, and enable logging controller outputs and execution profiling data.

When you begin a simulation in PIL mode, Simulink will check if any of the registered PIL implementations are valid. A customization file specifies which configuration parameters correspond

to a valid PIL implementation. You can explore the customization file for this implementation by invoking the following command.

```
edit(fullfile(matlabroot,'toolbox','rtw','rtwdemos','examplePilF28335','sl_customization.m'));
```

Notice that this file specifies setting for the hardware device and the template makefile which are required to use this PIL implementation. You can modify the configuration parameters in the controller model to match these settings. These changes can be made graphically in the model's Configuration Parameters dialog, or programmatically as shown below.

```
set_param('rtwdemo_pmsmfoc_local',...
       'ProdHWDeviceType', 'Texas Instruments->C2000',...
       'TemplateMakefile', 'ec_target.tmf',...
       'GenCodeOnly', 'off',...
       'SimulationMode', 'processor-in-the-loop (pil)')
```

You can specify collecting execution profiling information during PIL testing by logging the simulation output values as the variable `pilOut` and logging the execution profiling information as the variable `executionProfile`. These changes can be made graphically in the model's Configuration Parameters dialog, or programmatically as shown below.

```
set_param('rtwdemo_pmsmfoc_local',...
    'CodeExecutionProfiling', 'on',...
    'CodeExecutionProfileVariable','executionProfile',...
    'CodeProfilingSaveOptions','AllData');
save_system('rtwdemo_pmsmfoc_local.slx')
```

The controller model is now ready to be run in PIL mode.

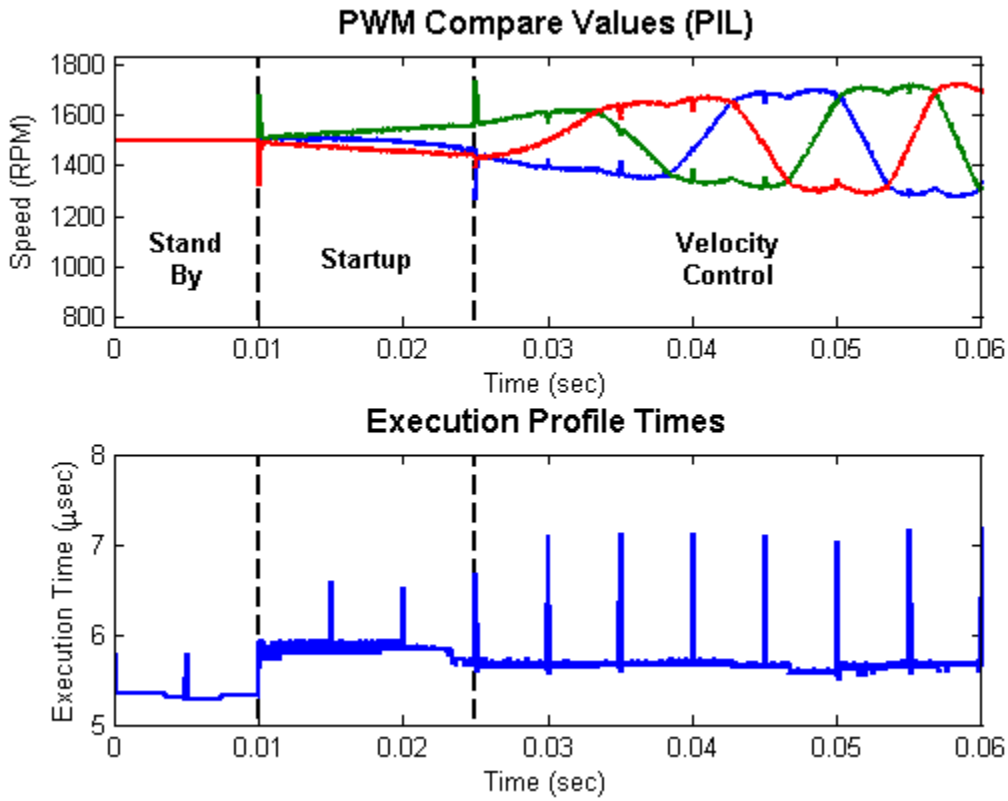
Test Behavior and Execution Time of Generated Code

In this section you will run the controller model in PIL mode and explore the behavioral and execution profiling results. You will verify that the behavior of the compiled controller code matches the reference simulation behavior, and then verify that the execution of the code meets timing requirements.

You can run the model and plot the PIL simulation results. When you start the model for the first time, Embedded Coder will generate code for the algorithm, link the algorithm code with the serial communication interface code, build the embedded application, download the application to the board, and begin the on-target simulation. Note that during subsequent PIL simulations, code is only regenerated if the model changes. Due to the overhead associated with the serial communication interface, the PIL simulation may run slower than the model in normal mode.

The MATLAB commands below are intentionally commented out as they require connection to hardware and use of embedded development tools described previously. If you have the hardware connected and embedded development tools installed, uncomment and execute these lines to run the model, plot the results, and verify the behavior is numerically equivalent to the simulation running in normal mode. Otherwise, continue reviewing this section to learn about PIL execution analysis options.

```
% UNCOMMENT THE BELOW LINES TO RUN THE SIMULATION AND PLOT THE RESULTS
% if exist('slprj','dir'), rmdir('slprj','s'); end
% out = sim('rtwdemo_pmsmfoc_local')
% pwm_compare_pil = out.logsout.getElement('pwm_compare').Values;
% rtwdemo_pmsmfoc_plotpwmcompare_pil(controller_mode, pwm_compare_pil, executionProfile)
```



The upper plot is the output of the controller, PWM Compare. Note that the outputs in PIL mode look the same as the outputs of the simulation in normal mode shown in the section "Establish Reference Behavior for Controller Model". You can subtract the outputs of normal mode simulation from the PIL mode simulation output to verify that they are numerically equivalent:

```
% UNCOMMENT THE BELOW LINE TO VERIFY NUMERICAL EQUIVALENCE OF THE OUTPUTS
% pilErrorWithRespectToReference = sum(abs(pwm_compare_pil.Data - pwm_compare_pil.Data))
```

```
pilErrorWithRespectToReference =

   0     0     0
```

The lower plot is the amount of time spent executing the controller model at each simulation time step. The "Stand By" state requires the least time. Small periodic spikes in execution time occur because the controller is multi-rate and single tasking. The periodic spikes correspond to the time required to run both the base rate and 5 millisecond rate code in the same task.

Since the controller must be executed at 25 kHz on the embedded processor, the algorithm must complete its execution within 40 microseconds (minus additional headroom requirements by other code, which may also be executing on the final application.) The profiling results indicate that the algorithm will execute within the time allotted for this configuration of the embedded environment.

The generated code is now verified to provide numerically equivalent results and meets the execution timing requirements for this test case.

```
close_system('rtwdemo_pmsmfoc_local',0);
close_system('power_utile',0);
```

The MATLAB preferences used in this PIL implementation are persistent between MATLAB sessions. If you want to remove these preferences, run the following commands.

```
rmpref('examplePilF28335');
rmexamplePilF28335hooks();
```

Conclusion

This example showed system level simulation and algorithmic code generation using a Field-Oriented Control algorithm for a Permanent Magnet Synchronous Machine to explore functional behavior of the controller algorithm. It also showed a general approach for target integration, functional testing, and execution profiling for any embedded processor. Once the algorithm was behaviorally correct, code was generated from the controller model, tested on the target processor and profiled. The algorithm code is now verified and can be integrated with embedded software that interfaces with the motor hardware for further testing.

See Also

Related Examples

- “SIL and PIL Simulations” on page 88-2
- “Choose a SIL or PIL Approach” on page 88-11
- “Create PIL Target Connectivity Configuration for Simulink” on page 88-38
- “Configure and Run SIL Simulation” on page 88-15
- “Create Execution-Time Profile for Generated Code” on page 81-2
- “MATLAB Project for FOC of PMSM with Quadrature Encoder” (Motor Control Blockset)
- “Algorithm-Export Workflows for Custom Hardware” (Motor Control Blockset)

Check Configuration

Use the `cgv.Config` class to check model settings for a SIL or PIL simulation. You can review your model configuration and determine the settings that you must change. By default, `cgv.Config` changes configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can:

- Change configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but not change the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Note

- Execution in the target environment can require additional modifications to configuration parameter values or the model.
- Do not use referenced configuration sets in models that you are changing using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `getRefConfigSet` method of the `Simulink.ConfigSetRef` class.
- If you use `cgv.Config` on a model that executes a callback function, the callback function can change configuration parameter values each time the model loads. The callback function can revert changes that `cgv.Config`. For more information, see “Customize Model Behavior with Callbacks”.

To verify that your model is configured for SIL or PIL:

1. Construct a `cgv.Config` object that changes the configuration parameter values without saving the model.
2. Determine and change the configuration parameter values that the object recommends using the `configModel` method.
3. Display a report of the changes that `configModel` makes.
4. Review the changes.
5. To apply the changes to your model, save the model.

See Also

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “Verify Numerical Equivalence with CGV” on page 88-136
- “Verify Numerical Equivalence Between Two Modes of Execution of a Model” on page 88-137

Verify Numerical Equivalence with CGV

Before verifying numerical equivalence:

- Configure your model for SIL or PIL simulation.
- Use the `cgv.Config` class of the CGV API to verify the model configuration for SIL or PIL simulation.
- Configure your model for code generation. For more information, see "Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 49-2.
- Save your model. If you modify a model without saving it, CGV can issue an error.

To verify numerical equivalence:

- Set up the tests for the first execution environment. For example, simulation.
- Use `run` to run the tests for the first execution environment.
- Set up the tests for the second execution environment. For example, top-model PIL.
- Use `cgv.CGV.run` to run the tests for the second execution environment.
- Use `getOutputData` to get the output data for each execution environment.
- Use `getSavedSignals` to display the signal names in the output data. (optional)
- Build a list of signal names for input to other `cgv.CGV` methods. (optional)
- Use `createToleranceFile` to create a file correlating tolerance information with output signal names. (optional)
- Use `compare` to compare the output signals of the first and second execution environments for numerical equivalence.

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

See Also

Related Examples

- "Configure and Run SIL Simulation" on page 88-15
- "Check Configuration" on page 88-135
- "Verify Numerical Equivalence Between Two Modes of Execution of a Model" on page 88-137

Verify Numerical Equivalence Between Two Modes of Execution of a Model

In this section...

"Configure the Model" on page 88-137
"Execute the Model" on page 88-137
"Compare All Output Signals" on page 88-138
"Compare Individual Output Signals" on page 88-139
"Plot Output Signals" on page 88-140

The following example describes configuring, executing, and comparing the results of the rtwdemo_cgv model in normal and software-in-the-loop (SIL) simulation modes.

Configure the Model

The first task for verifying numerical equivalence is to check the configuration of your model.

1. Open the `rtwdemo_cgv` model.

```
cgvModel = 'rtwdemo_cgv';
load_system(cgvModel);
```

2. Save the model to a working directory.

```
save_system(cgvModel, fullfile(pwd, cgvModel));
close_system(cgvModel); % avoid original model shadowing saved model
```

3. Use the `cgv.Config` to create a `cgv.Config` object. Specify parameters that check and modify configuration parameter values and save the model for top-model SIL mode of execution.

```
cgvCfg = cgv.Config('rtwdemo_cgv', 'connectivity', 'sil', 'SaveModel', 'on');
```

4. Use the `configModel` method to review your model configuration and to change the settings to configure your model for SIL. When `'connectivity'` is set to `'sil'`, the system target file is automatically set to `'ert.tlc'`. If you specified the parameter/value pair, (`'SaveModel', 'on'`) when you created the `cgvCfg` object, the `cgv.Config.configModel` method saves the model.

   Note CGV runs on models that are open. If you modify a model without saving it, CGV can issue an error.

```
cgvCfg.configModel(); % Evaluate, change, and save your model for SIL
```

5. Display a report of the changes that `cgv.Config.configModel` makes to the model.

```
cgvCfg.displayReport(); % In this example, this reports no changes
```

Execute the Model

Use the CGV API to execute the model in two modes. The two modes in this example are normal mode simulation and SIL mode. In each execution of the model, the CGV object for each mode captures the output data and writes the data to a file.

1. If you have not already done so, follow the steps described in “Configure the Model” on page 88-137.
2. Create a `cgv.CGV` object that specifies the `rtwdemo_cgv` model in normal mode simulation.

   ```
   cgvSim = cgv.CGV(cgvModel, 'connectivity', 'sim');
   ```

   Note When the top model is set to normal simulation mode, the CGV API sets referenced models in PIL mode to accelerator mode.

3. Provide the input file to the `cgvSim` object.

   ```
   cgvSim.addInputData(1, [cgvModel '_data']);
   ```

4. Before execution of the model, specify the MATLAB files to execute or MAT-files to load. This step is optional.

   ```
   cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
   ```

5. Specify a location where the object writes all output data and metadata files for execution. This step is optional.

   ```
   cgvSim.setOutputDir('cgv_output');
   ```

6. Execute the model.

   ```
   result1 = cgvSim.run();
   ```

7. Get the output data associated with the input data.

   ```
   outputDataSim = cgvSim.getOutputData(1);
   ```

8. For the next mode of execution, SIL, repeat steps 2–7.

   ```
   cgvSil = cgv.CGV( cgvModel, 'Connectivity', 'sil');
   cgvSil.addInputData(1, [cgvModel '_data']);
   cgvSil.addPostLoadFiles({[cgvModel '_init.m']});
   cgvSil.setOutputDir('cgv_output');
   result2 = cgvSil.run();
   ```

Compare All Output Signals

After setting up and running the test, compare the outputs by doing the following:

1. If you have not already done so, configure and test the model, as described in “Configure the Model” on page 88-137 and “Execute the Model” on page 88-137.
2. Test that the execution result of the model:

   ```
   if ~result1 || ~result2
       error('Execution of model failed.');
   end
   ```

3. Use the `getOutputData` method to get the output data from the cgv.CGV objects.

   ```
   simData = cgvSim.getOutputData(1);
   silData = cgvSil.getOutputData(1);
   ```

4. Display a list of signals by name using the `getSavedSignals` method.

   ```
   cgvSim.getSavedSignals(simData);
   ```

5. Using the list of signals, build a list of signals in a cell array of character vectors. The signal list can contain a number of signals.

   ```
   signalList = {'simData.getElement(4).Values.Data'};
   ```

6. Use the `createToleranceFile` method to create a file, in this example, `'localtol'`, correlating tolerance information with output signal names.

```
toleranceList = {{'absolute', 0.5}};
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
```

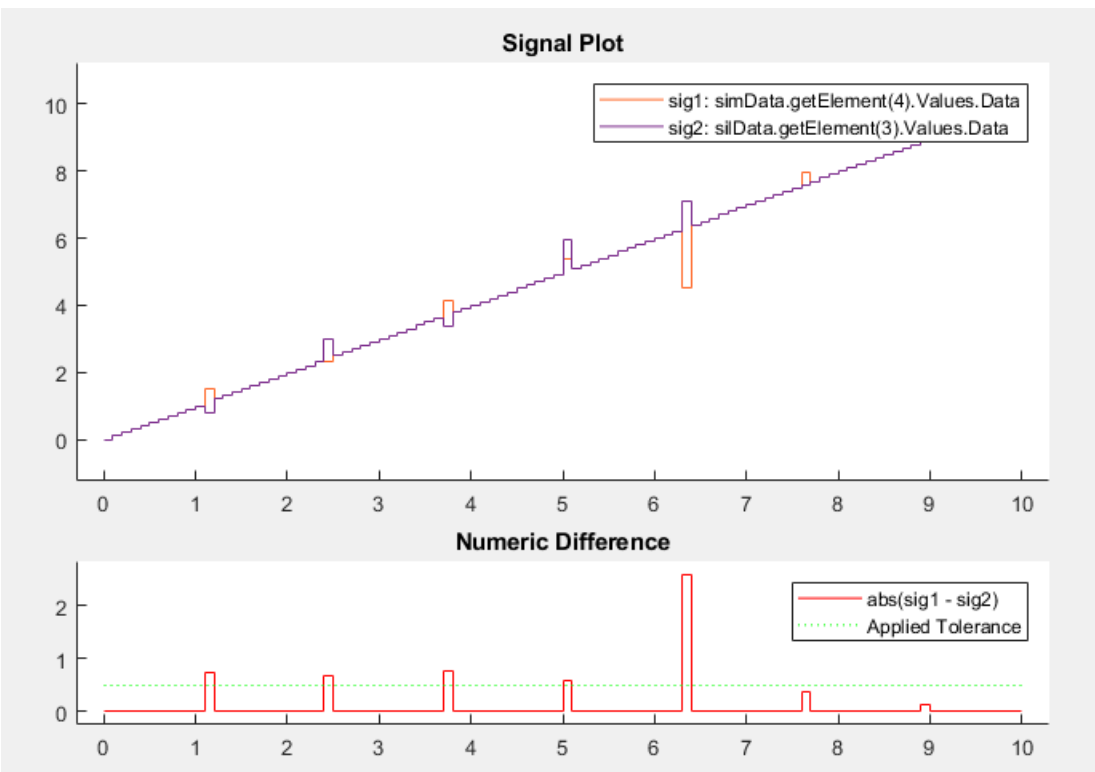
7 Compare the output data signals. By default, the `compare` method looks at all signals which have a common name between both executions. If a tolerance file is present, `cgv.CGV.compare` uses the associated tolerance for a specific signal during comparison; otherwise the tolerance is zero. In this example, the `'Plot'` parameter is set to `'mismatch'`. Therefore, only mismatched signals produce a plot.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'Tolerancefile', 'localtol');
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
disp('Mismatched Signal Names:');
disp(mismatchNames);
```

At the MATLAB command line, you see:

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.getElement(4).Values.Data'
```

A plot results from the signal mismatch.



The lower plot displays the numeric difference between the results.

Compare Individual Output Signals

After setting up and running the test, compare the outputs of individual signals by doing the following:

1. If you have not already done so, configure and test the model, as described in “Configure the Model” on page 88-137 and “Execute the Model” on page 88-137.
2. Use the `getOutputData` method to get the output data from the `cgv.CGV` objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

3. Use the `getSavedSignals` method to display the output data signal names. Build a list of specific signal names in a cell array of character vectors. The signal list can contain a number of signals.

```
cgv.CGV.getSavedSignals(simData);

signalList = {'simData.getElement(3).Values.hi1.mid0.lo1.Data', ...
'simData.getElement(3).Values.hi1.mid0.lo2.Data', ...
'simData.getElement(2).Values.Data(:,3)'};
```

4. Use the specified signals as input to the `compare` method to compare the signals from separate runs.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'signals', signalList);
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames);
end
```

At the MATLAB command line, the result is:

```
3 Signals match, 0 Signals mismatch
```

Plot Output Signals

After setting up and running the test, use the `plot` method to plot output signals.

1. If you have not already done so, configure and test the model, as described in “Configure the Model” on page 88-137 and “Execute the Model” on page 88-137.
2. Use the `getOutputData` method to get the output data from the `cgv.CGV` objects.

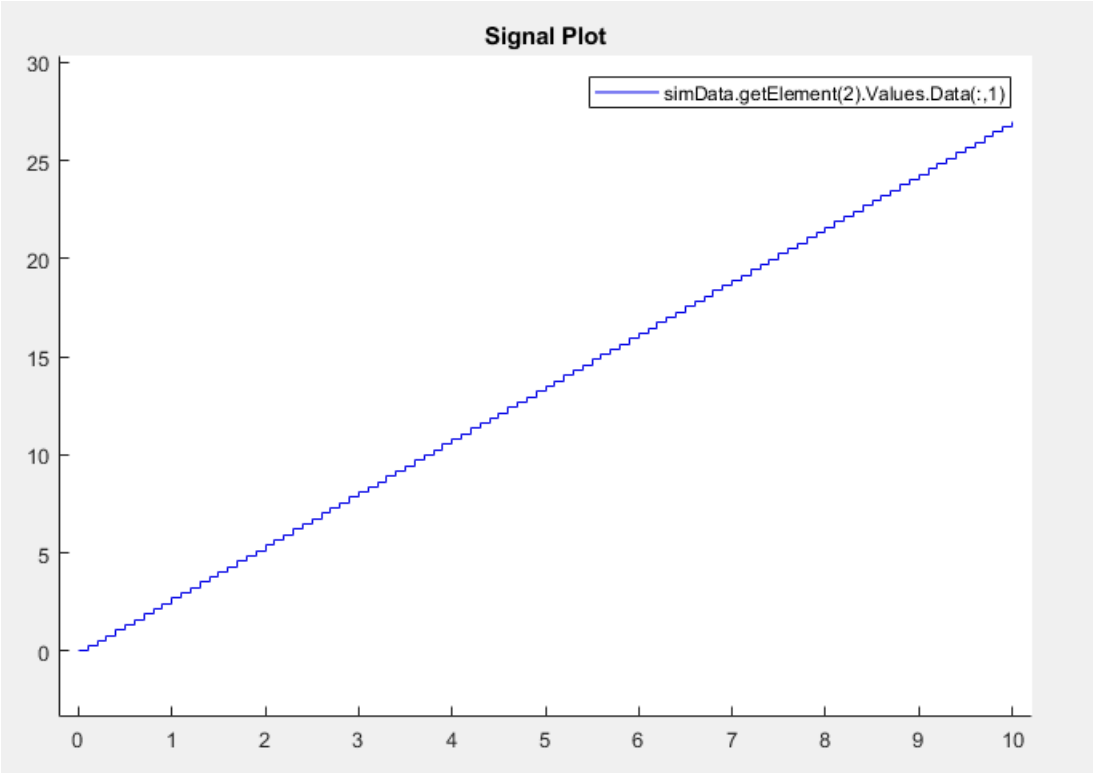
```
simData = cgvSim.getOutputData(1);
```

3. Use the `getSavedSignals` method to display the output data signal names. Build a list of specific signal names in a cell array of character vectors. The signal list can contain a number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.getElement(2).Values.Data(:,1)'};
```

4. Use the specified signal list as input to the `plot` method to compare the signals from separate runs.

```
[signalNames, signalFigures] = cgv.CGV.plot(simData, ...
     'Signals', signalList);
```



See Also

Related Examples

- “Verify Numerical Equivalence with CGV” on page 88-136
- “Check Configuration” on page 88-135

Numerical Consistency between Model and Generated Code

Numerical Consistency of Model and Generated Code Simulation Results

In this section...

"Numerical Consistency" on page 89-2
"Numerical Consistency in Complex Systems" on page 89-2
"Reasons for Block-Level Numerical Differences" on page 89-4

Numerical Consistency

In the Model-Based Design workflow, you use MathWorks products to generate code for numerical applications that employ fixed-point and floating-point arithmetic.

- To develop models, you use MATLAB, Simulink, and Stateflow.
- To generate source code, you use Simulink Coder and Embedded Coder.
- To test numerical equivalence between your model and generated code, you compare model and generated code simulation results. For example, normal mode simulation results compared with software-in-the-loop (SIL) simulation results.

The results from the model and generated code simulations are numerically consistent if:

- In fixed-point applications, the results agree in a bit-wise comparison.
- In floating-point applications, the results agree with an error tolerance that you specify.

Use the Simulation Data Inspector to compare results. To determine whether discrepancies exist or are significant, you can specify absolute and relative tolerance values:

- For fixed-point applications, you can specify an absolute tolerance of zero.
- For floating-point applications, you can specify tolerance with respect to a reference value or signal. The choice of reference depends on your application. Consider these examples:
  - An algorithm that solves a linear algebraic equation by iterative, feed-forward error calculations. You can specify tolerance with respect to `eps`.
  - A Proportional-Integral-Derivative (PID) controller for a closed-loop system. For transient behavior, you can specify tolerance with criteria from a standard. For steady-state behavior, you can specify tolerance with reference to the PID controller characteristics.

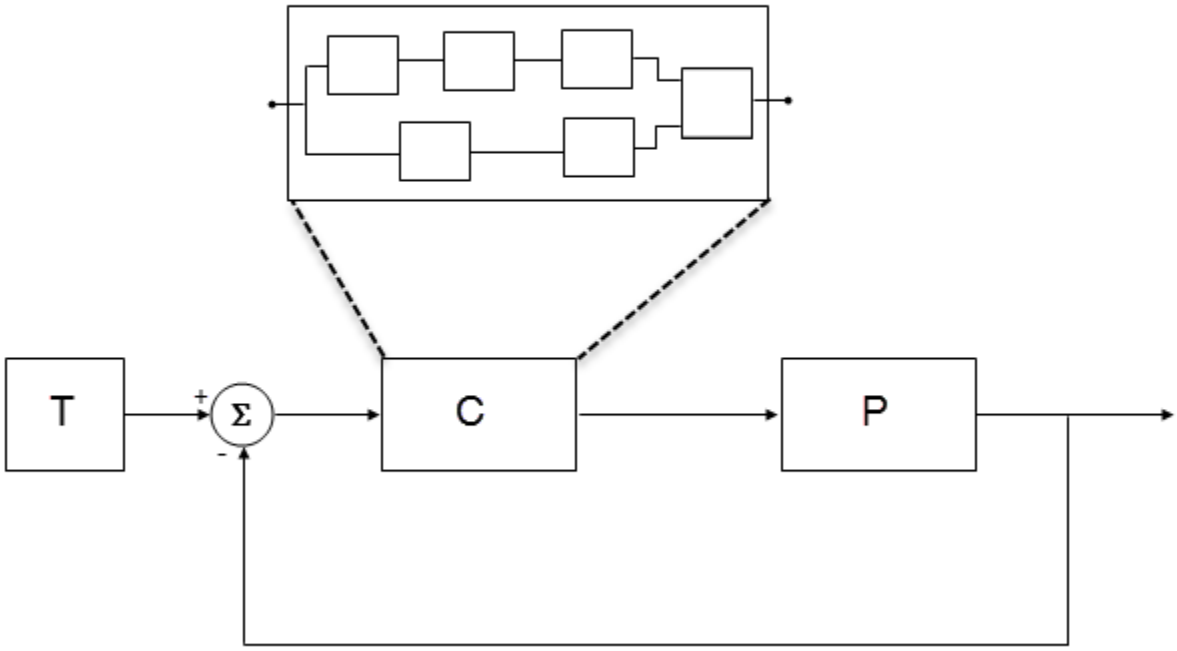
Programmatically, you can specify absolute and relative tolerance values through the `absTol` and `relTol` properties of the `Simulink.sdi.Signal` object.

Numerical Consistency in Complex Systems

For complex systems, numerical differences between model and generated code simulations can be a result of block-level differences propagating through the system. If you observe numerical differences at the system level:

1. Identify blocks for which block-level numerical differences exceed the error tolerance.
2. Investigate each identified block.

Consider the following plant-controller model.



- T produces reference or test signals.
- C is the controller component. The controller output is the plant input. C can be a Model block that comprises multiple Model blocks.
- P is the plant component. The plant output is subtracted from the reference signal to produce the controller input.

To test numerical equivalence between the model controller and the generated code version:

1. Run the model in normal mode, and, using the Simulation Data Inspector, record the output of C.
2. Specify SIL mode for C. Rerun the simulation, recording the output of C.
3. Using the Simulation Data Inspector, compare normal and SIL mode outputs with reference to your specified error tolerance.

If the Simulation Data Inspector comparison indicates a match, the model and generated code results are numerically consistent.

If the normal and SIL mode outputs do not match:

1. Within C, enable signal logging for block outputs.
2. Run the model in normal mode.
3. Rerun the simulation with C in SIL mode.

4. Using the Simulation Data Inspector, compare the logged output signals with reference to your specified error tolerance. See “Compare Simulation Data”.
5. Identify blocks for which normal and SIL mode output differences exceed the error tolerance.
6. Analyze each identified block and look for the cause. For example, the generated code might use a different math library than MATLAB.

Note If the comparison of a large number of signals is required, you can automate the workflow with Simulink Test. See “SIL Verification for a Subsystem” (Simulink Test).

Reasons for Block-Level Numerical Differences

In fixed-point and floating-point application development, there are factors that can affect numerical agreement between block-level results from model and generated code simulations.

Some factors can affect both fixed-point and floating-point applications. For example, the use of:

- Code generation optimization.
- Custom code.
- Code replacement library entries whose results differ from MATLAB results.
- Code replacement libraries that implement different algorithms.

Other factors affect only floating-point applications. For example:

- Numerical soundness of algorithm.
- Algorithm sensitivity to input.
- Closed loop and open loop behavior.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also

`eps`

Related Examples

- “Analyze Simulation Results”
- “SIL Verification for a Subsystem” (Simulink Test)
- “Differences Between Generated Code and MATLAB Code”
- “Types of In-the-Loop Testing in the V-Model”
- MATLAB Function
- “SIL and PIL Limitations” on page 88-83

Software-in-the-Loop Execution for MATLAB Coder

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2
- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Software-in-the-Loop Execution From Command Line” on page 90-6
- “Debug Generated Code During SIL or PIL Execution” on page 90-9
- “Create PIL Target Connectivity Configuration for MATLAB” on page 90-12
- “Host-Target Communication for MATLAB PIL Execution” on page 90-16
- “Specify Hardware Timer for MATLAB” on page 90-19
- “Custom Toolchain Directives Required for Code Execution Profiling” on page 90-22
- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23
- “Processor-in-the-Loop Execution From Command Line” on page 90-25
- “Verification of Code Generation Assumptions” on page 90-29
- “SIL/PIL Execution Support and Limitations” on page 90-30
- “Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization” on page 90-32
- “SIL and PIL Verification for Deployment on Raspberry Pi” on page 90-33
- “Software-in-the-loop Execution For MATLAB Function with Multiple Signatures” on page 90-39

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

MATLAB Coder supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution, which enables you to verify production-ready source code and compiled object code. With these execution modes, you can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of library code.

In SIL execution, through a MATLAB SIL interface, the software compiles and runs library code on your development computer. In PIL execution, through a MATLAB PIL interface, the software cross-compiles and runs production object code on a target processor or an equivalent instruction set simulator. Before you run a PIL execution, you must set up a PIL connectivity configuration for your target.

The workflow for generating and verifying code is:

1. Set up MATLAB Coder.
2. Fix errors detected at design time.
3. Generate MEX function.
4. Test MEX function.
5. Generate C/C++ library code.
6. Verify generated C/C++ code through SIL or PIL execution — requires Embedded Coder license.

In step 4, you verify code that is generated for execution within MATLAB. However, this code is different from the standalone code generated for libraries. In step 6, with an Embedded Coder license, you use SIL or PIL execution to verify the standalone code.

For more information, use the following table.

Feature	See
SIL execution	• “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4 • “Software-in-the-Loop Execution From Command Line” on page 90-6
PIL target connectivity configuration	• “Create PIL Target Connectivity Configuration for MATLAB” on page 90-12 • “Host-Target Communication for MATLAB PIL Execution” on page 90-16 • “Processor-in-the-Loop Execution From Command Line” on page 90-25
PIL execution	• “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23 • “Processor-in-the-Loop Execution From Command Line” on page 90-25

Feature	See
Code generation, MEX functions, and libraries	• “MATLAB Code Analysis” • “Generating Code” • “Deployment”

Software-in-the-Loop Execution with the MATLAB Coder App

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click , and then click `Open existing project`. Select the project. For example, `kalman_filter01.prj`.
3. On the **Generate Code** page, click the **Generate** arrow .
4. In the **Generate** dialog box:
   a. Set **Build type** to `Static Library` or `Dynamic Library`.
   b. In the **Output file name** field, use the default value. For example, `kalman01`.
   c. Specify **Language**.
   d. Clear the **Generate code only** check box.
   e. In the **Hardware Board** field, use the default value (`MATLAB Host Computer`).

   You do not have to specify the **Toolchain** setting. By default, the MATLAB Coder app locates an installed toolchain.
5. To generate the C or C++ code, click **Generate**.
6. Click **Verify Code**.
7. In the command field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.
8. If required, select the **Enable source-level debugging for SIL or PIL** check box.
9. To start the SIL execution, click **Run Generated Code**.

   The MATLAB Coder app:

   - Generates a standalone library, for example, `codegen\lib\kalman01`.
   - Generates SIL interface code, for example, `codegen\lib\kalman01\sil`.
   - Runs the test file, replacing calls to the MATLAB function with calls to the generated code in the library.
   - Displays messages from the SIL execution in the **Test Output** tab.
10. Verify that the results from the SIL execution match the results from the original MATLAB functions.
11. To terminate the SIL execution process, click **Stop SIL Verification**. Alternatively, on the **Test Output** tab, click the link that follows `To terminate execution`.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

See Also

Related Examples

- “Generate C Code by Using the MATLAB Coder App”
- “Software-in-the-Loop Execution From Command Line” on page 90-6
- “Debug Generated Code During SIL or PIL Execution” on page 90-9
- “Generate Execution Time Profile” on page 82-3
- “Generate Stack Usage Profile” on page 82-13

More About

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2

Software-in-the-Loop Execution From Command Line

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

To set up and start a SIL execution from the command line:

1. Create a `coder.EmbeddedCodeConfig` object.
2. Configure the object for SIL.
3. Use the `codegen` function to generate library code for your MATLAB function and the SIL interface.
4. Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the SIL execution, use the `clear` *`function`*`_sil` or `clear mex` command.

The following example shows how you can set up and run a SIL execution from the command line.

Tutorial Files: Kalman Filter

Open this example to obtain the files for this tutorial:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

SIL Execution of Code Generated for a Kalman Estimator

1. **Configure SIL execution**

   a. From your working folder, create a `coder.EmbeddedCodeConfig` object.

   ```
   config = coder.config('lib');
   config.GenerateReport = true; % Optional, documents code in HTML report
   ```

   b. Configure the object for SIL.

   ```
   config.VerificationMode = 'SIL';

   % Check that production hardware setting is the default
   % i.e. 'Generic->MATLAB Host Computer'
   disp(config.HardwareImplementation.ProdHWDeviceType);
   ```

   c. If required, enable the Microsoft Visual Studio debugger for SIL execution:

   ```
   config.SILPILDebugging = true;
   ```

2. **Generate code and run SIL execution**

   Generate library code for the `kalman01` MATLAB function and the SIL interface, and run the MATLAB test file, `test01_ui`. The test file uses `kalman01_sil`, the generated SIL interface for `kalman01`.

```
codegen -config config -args {zeros(2,1)} kalman01 -test test01_ui
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\sil` — SIL interface code for `kalman01`.

Verify that the output of this run matches the output from the original `kalman01.m` function.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

3 **Debug code during SIL execution**

If you enable the Microsoft Visual Studio debugger, then running the test file opens the Microsoft Visual Studio IDE with debugger breakpoints at the start of the `kalman01_initialize` and `kalman01` functions.

You can use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

a Remove all breakpoints.

b Click the **Continue** button (**F5**).

The SIL execution runs to completion.

4 **Terminate SIL execution**

Terminate the SIL execution process.

```
clear kalman01_sil;
```

You can also use the command `clear mex`, which clears MEX functions from memory.

Note Suppose that the MATLAB current working directory (the folder that `pwd` displays) is a UNC path (for example, `\\server\a\b\c`) on a Windows platform. If you launch a SIL executable from this path, the location where the SIL executable is launched can be unpredictable and different from the MATLAB current working directory (for example, `C:\Windows`).

To fix this issue, use a mapped network drive for the UNC path as your MATLAB current working directory.

See Also

Related Examples

- “Generate C Code by Using the MATLAB Coder App”
- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Debug Generated Code During SIL or PIL Execution” on page 90-9

- “Generate Execution Time Profile” on page 82-3
- “Generate Stack Usage Profile” on page 82-13

More About

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2

Debug Generated Code During SIL or PIL Execution

In this section...
“SIL Debugging” on page 90-9 “PIL Debugging” on page 90-11

If a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution fails or you notice differences between the outputs of your original functions and the generated code, you can rerun the execution with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which might help you to understand the cause of the problem.

For a SIL or PIL execution failure, you can also view information from the standard output and standard error streams in the MATLAB Command Window. For example:

- Output from `printf` statements in your code.
- If you enable run-time error detection, messages sent to `stderr`.
- Some low-level system messages.

Note During a SIL execution, the target application redirects the `stdout` and `stderr` streams. When the application terminates, the MATLAB Command Window displays the information from the redirected streams. The target application also provides a basic signal handler, which captures the POSIX signals `SIGFPE`, `SIGILL`, `SIGABRT`, and `SIGSEV`. For this signal handler, the target application includes the file `signal.h`.

SIL Debugging

This table gives information about debugger support.

Operating System	Supported Debuggers
Windows	Microsoft Visual Studio IDE.
	Visual Studio Code with MinGW GDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.
macOS	LLDB.
	Visual Studio Code with LLDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.
Linux	GNU Data Display Debugger (DDD).

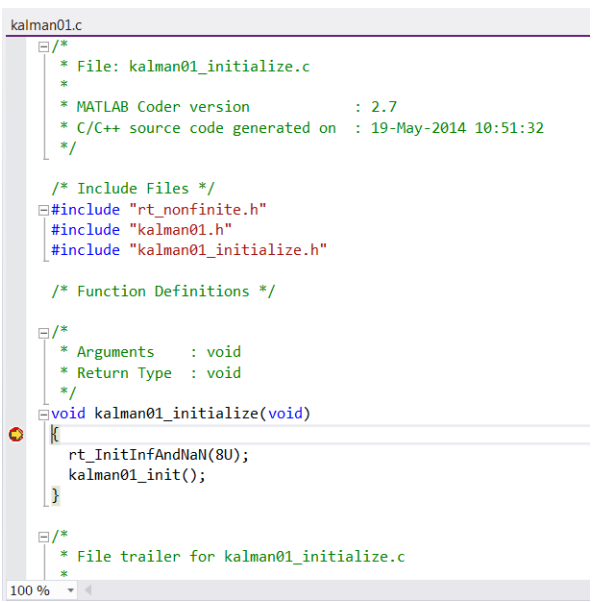
Operating System	Supported Debuggers
	Visual Studio Code with GDB. Requires installation of MATLAB Coder Interface for Visual Studio Code Debugging support package. See https://www.mathworks.com/matlabcentral/fileexchange/103555.

Note You can perform SIL debugging only if the MATLAB product family supports your Microsoft Visual C++, Xcode, or GNU GCC compiler. For more information, see supported compilers.

To run a SIL execution with debugging enabled:

1. On the **Generate Code** page, click **Verify Code**.
2. Select the **Enable source-level debugging for SIL or PIL** check box.
3. Click **Run Generated Code**.

On a Windows computer, your *`user_fn`*`.c` or *`user_fn`*`.cpp` file opens, for example, in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the *`user_fn`*`_initialize` and *`user_fn`* functions.



You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

1. Remove all breakpoints.
2. Click the **Continue** button (**F5**).

   The SIL execution runs to completion.
3. To terminate the SIL execution process, on the **Test Output** tab, click the link that follows `To terminate execution`, for example, `clear kalman01_sil`.

The Microsoft Visual Studio IDE closes automatically.

Note If you select **Debug** > **Stop Debugging**, the SIL execution times out with the following error message:

```
Communications error: failed to send data to the target. There might be
multiple reasons for this failure.

...
...
```

PIL Debugging

When you set up PIL connectivity between Simulink and the target hardware, specify a debugger by using `target.ExecutionService` and `target.DebugExecutionTool` objects — see “Support PIL Debugging” on page 88-51 and “DebugExecutionTool Template” on page 88-52.

To run a PIL execution with debugging enabled:

1. On the **Generate Code** page, click **Verify Code**.
2. Select the **Enable source-level debugging for SIL or PIL** check box.
3. Click **Run Generated Code**.

See Also

Related Examples

- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Software-in-the-Loop Execution From Command Line” on page 90-6
- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23
- “Processor-in-the-Loop Execution From Command Line” on page 90-25

Create PIL Target Connectivity Configuration for MATLAB

In this section...

"Target Connectivity Configurations for PIL" on page 90-12
"Create a Target Connectivity API Implementation" on page 90-12
"Register Target Connectivity Configuration" on page 90-14
"Verify Target Connectivity Configuration" on page 90-14

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) execution for your target environments.

Through a target connectivity configuration, you specify:

- A target connectivity configuration name for a target connectivity API implementation.
- Settings that define compatible MATLAB code. For example, the code that is generated for a particular hardware implementation.

A PIL execution requires a target connectivity PIL API implementation that integrates third-party tools for:

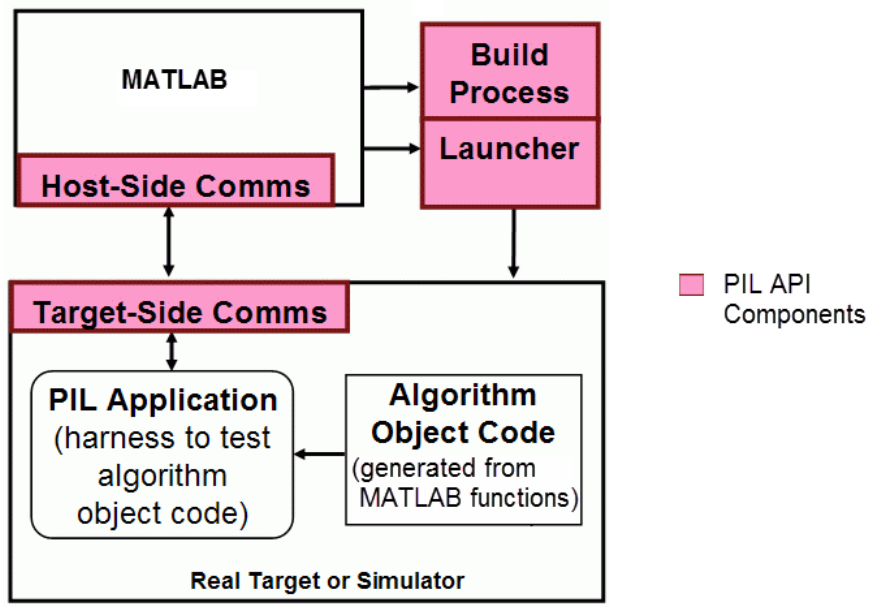
- Building the PIL application that runs on the target hardware
- Downloading, starting, and stopping the application on the target
- Communicating between MATLAB and the target

You can have many different connectivity configurations for PIL execution. Register a connectivity configuration with MATLAB by creating an `rtwTargetInfo.m` file and placing it on the MATLAB search path.

In a PIL execution, the software determines which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the MATLAB code under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

Create a Target Connectivity API Implementation

This diagram shows the components of the PIL target connectivity API.



You must provide implementations of the three API components:

- Build API — Specify a toolchain approach for building generated code.
- Launcher API — Control how MATLAB starts and stops the PIL executable.
- Communications API — Customize connectivity between MATLAB and the PIL target. Embedded Coder provides host-side support for TCP/IP and serial communications, which you can adapt for other protocols.

These steps outline how you create a target connectivity API implementation. The example code shown in the steps is taken from the `ConnectivityConfig.m` file used in “Processor-in-the-Loop Execution From Command Line” on page 90-25.

1. Create a subclass of `rtw.connectivity.Config`.

   ```
   ConnectivityConfig < rtw.connectivity.Config
   ```

2. In the subclass:

   - Instantiate `rtw.connectivity.MakefileBuilder`, which configures the build process.

     ```
     builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
                    targetApplicationFramework, ...
                    exeExtension);
     ```

   - Create a subclass of `rtw.connectivity.Launcher`, which downloads and executes the application using a third-party tool.

     ```
     launcher = mypil.Launcher(componentArgs, builder);
     ```

3. Configure your `rtiostream` API implementation of the host-target communications on page 90-16 channel.

   - For the target side, you must provide the driver code for communications, for example, code for TCP/IP or serial communications. To integrate this code into the build process, create a subclass of `rtw.pil.RtIOStreamApplicationFramework`.

- For the host side, you can use a supplied library for TCP/IP or serial communications. Instantiate `rtw.connectivity.RtIOStreamHostCommunicator`, which loads and initializes the library that you specify.

```
hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                       componentArgs, ...
                       launcher, ...
                       rtiostreamLib);
```

4. If you require execution-time profiling for generated code, create a timer object that provides details of the hardware-specific timer and associated source files. See "Specify Hardware Timer for MATLAB" on page 90-19.
5. If you require stack usage profiling for generated code, specify a driver implementation that obtains stack usage data from the target hardware. The driver must return the value of the stack register. If you do not specify a driver, the PIL simulation tries to use a default generic driver. See "Implement Driver to Obtain Stack Usage Data During PIL Execution" on page 82-17.

Register Target Connectivity Configuration

To register a target connectivity API implementation as a target connectivity configuration in MATLAB:

1. Create or update an `rtwTargetInfo.m` file. In this file:
   - Create a target connectivity configuration object that specifies, for example, the configuration name for a target connectivity API implementation and compatible MATLAB code.
   - Invoke `registerTargetInfo`.
2. Add the folder containing `rtwTargetInfo.m` to the search path and refresh the MATLAB Coder library registration information.

For more information, see `rtw.connectivity.ConfigRegistry`.

Verify Target Connectivity Configuration

To verify your target connectivity configuration early on and independently of your algorithm development and code generation, use the `piltest` function. With the function, you can run a suite of tests. The function:

- Runs the MATLAB function and performs PIL executions.
- Compares results and produces errors if it detects differences.

For an example, see "PIL Execution of Code Generated for a Kalman Estimator" on page 90-25.

See Also

`piltest` | `rtw.connectivity.Config` | `rtw.connectivity.ConfigRegistry` | `rtw.connectivity.Launcher` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.pil.RtIOStreamApplicationFramework`

Related Examples

- "Processor-in-the-Loop Execution From Command Line" on page 90-25

- “Design Subclass Constructors”
- “Host-Target Communication for MATLAB PIL Execution” on page 90-16
- “Specify Hardware Timer for MATLAB” on page 90-19

Host-Target Communication for MATLAB PIL Execution

In this section...

"Communications rtiostream API" on page 90-16
"Synchronize Host and Target" on page 90-17
"Test an rtiostream Driver" on page 90-17
"Troubleshooting Host-Target Communication" on page 90-17

This section provides information about using the `rtiostream` API for communication between your development computer and target hardware during PIL execution.

Communications rtiostream API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL verification requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation as well as a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see "Create a Target Connectivity API Implementation" on page 90-12.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see "Test an rtiostream Driver" on page 90-17 and `rtiostreamtest`.

Synchronize Host and Target

If you use the `rtiostream` API to implement the communications channel, the host and target must be synchronized, which prevents MATLAB from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP rtiostream implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator`. This approach works well for connection-oriented TCP/IP `rtiostream` implementations because MATLAB automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the MATLAB side of the `rtiostream` connection opens without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher` / `rtiostream` implementation that confirms completion of target initialization.

Test an rtiostream Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite can help you to:

- Reduce the time required for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduce the time required for testing custom `rtiostream` drivers.
- Analyze the performance of custom `rtiostream` drivers.

The test suite consists of two parts. One part is an application that runs on the target hardware. The other part runs in MATLAB. For details, see `rtiostreamtest`.

Troubleshooting Host-Target Communication

To display communication I/O information during PIL execution, in the MATLAB Coder app, on the **All Settings** tab, set **SIL/PIL Verbosity** to `On`. Or, in a `coder.EmbeddedCodeConfig` object, set the `SILPILVerbosity` property to `true`.

By default, **SIL/PIL Verbosity** is set to `Off`.

For SIL execution with **SIL/PIL Verbosity** set to `On`, the app displays I/O information for communication that occurs between MATLAB and the target application that runs as a separate process on your development computer.

See Also

`rtIOStreamClose` | `rtIOStreamOpen` | `rtIOStreamRecv` | `rtIOStreamSend` | `rtiostream_wrapper` | `rtiostreamtest` | `rtw.connectivity.RtIOStreamHostCommunicator`

Related Examples

- “Create PIL Target Connectivity Configuration for MATLAB” on page 90-12

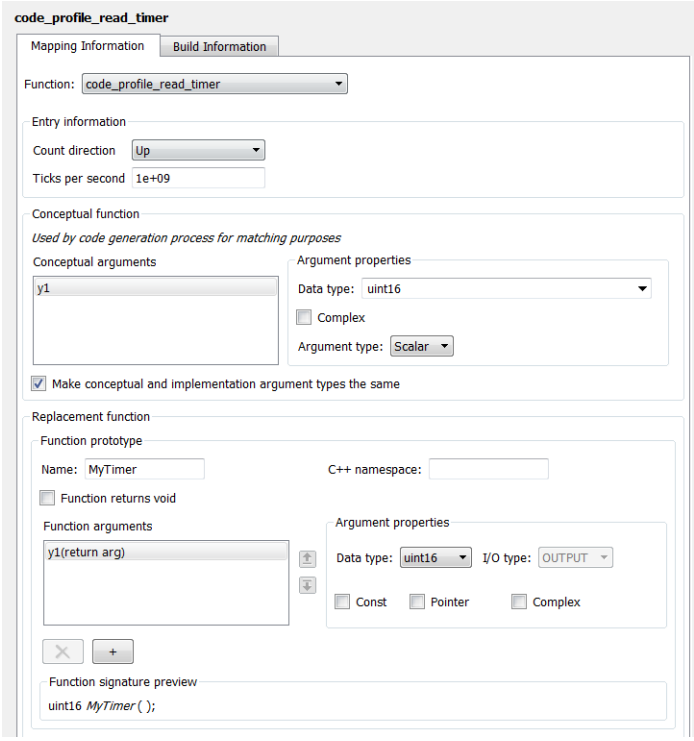
Specify Hardware Timer for MATLAB

For processor-in-the-loop (PIL) code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the **Code Replacement Tool** or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

1. Open the Code Replacement Tool. In the Command Window, enter `crtool`.
2. Create a new code replacement table. Select **File** > **New table**.
3. Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry** > **Function**.
4. In the middle view, select the new unnamed function.
5. On the **Mapping Information** pane:
   a. From the **Function** drop-down list, select `code_profile_read_timer`.
   b. Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select `Up`.
   c. In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

      The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.
   d. In the **Conceptual function** section, from the **Data type** list, select `uint8`, `uint16`, `uint32`, or `uint64`.
   e. In the **Function prototype** section, in the **Name** field, specify a replacement function name, for example, `MyTimer`.
   f. Click **Apply**.



g To validate the function entry, click **Validate entry**.

6 On the **Build Information** pane, specify the required build information. See Specify build information in “Define Code Replacement Library Optimizations” on page 72-36.

7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

8 Assuming you save the table as *`MyCrlTable`*`.m`, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCrlTable)
```

See Also

Related Examples

- “Create a Target Connectivity API Implementation” on page 90-12
- “Generate Execution Time Profile” on page 82-3
- “Define Code Replacement Library Optimizations” on page 72-36

More About

- “What Is Code Replacement?” on page 57-2

- “What Is Code Replacement Customization?” on page 72-3

Custom Toolchain Directives Required for Code Execution Profiling

If you use a custom toolchain for building a processor-in-the-loop (PIL) target application that supports code execution profiling, in the toolchain definition file, you must provide these compiler directives:

- `CompileFlag` — Specify the flag that C or C++ compiler uses to compile source files without linking. For example, if the compiler in the toolchain is GNU C or C++, set the directive to -c.
- `PreprocessFile` — Specify the flag that C or C++ compiler uses to preprocess the source files without running other compiler stages. For example, if the compiler in the toolchain is GNU C or C ++, set the directive to -E.

This section from a toolchain definition file example shows how you can use the `setDirective` method to provide the directives.

```
tc = coder.make.ToolchainInfo('BuildArtifact','nmake makefile');
tc.Name ='My Toolchain Name';
tc.Platform ='win64';
tc.SupportedVersion ='14';
...
% ------------------------------
% C Compiler
% ------------------------------
tool = tc.getBuildTool('C Compiler');
tool.setName('My C Compiler');
...
tool.setDirective('CompileFlag','-c');
tool.setDirective('PreprocessFile','-E');
...
```

See Also

More About

- “Custom Toolchain Registration”
- “Add Custom Toolchains to MATLAB® Coder™ Build Process”

Processor-in-the-Loop Execution with the MATLAB Coder App

Use processor-in-the-loop (PIL) execution to verify the numerical behavior of cross-compiled object code with reference to your original MATLAB functions.

Before you run a PIL execution, you must define a target connectivity configuration. In “Processor-in-the-Loop Execution From Command Line” on page 90-25, steps 1 and 2 of the example “PIL Execution of Code Generated for a Kalman Estimator” on page 90-25 show how you can set up and register a connectivity configuration for PIL execution on your development computer.

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click , and then click `Open existing project`. Select the project. For example, `kalman_filter.prj`.
3. On the **Generate Code** page, click the **Generate** arrow .
4. In the **Generate** dialog box:
   a. Set **Build type** to `Static Library` or `Dynamic Library`.
   b. In the **Output file name** field, use the default value. For example, `kalman01`.
   c. Clear the **Generate code only** check box.
   d. From the **Hardware Board** drop-down list, select `None - Select device below`.
   e. In the **Device** fields, specify vendor and type. These settings must match the target hardware settings in the `rtwTargetInfo.m` file of your target connectivity configuration. For host-based PIL, select settings that match your host computer. For example:
      - For a Windows 64-bit system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)`. In addition, set **Enable long long** to `Yes`.
      - For a Linux 64-bit system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Linux 64)`.
      - For a Mac OS X system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Mac OS X)`.

   You do not have to specify the **Toolchain** setting. By default, the MATLAB Coder app locates an installed toolchain.
5. To generate the C or C++ code, click **Generate**.
6. Click **Verify Code**.
7. In the command field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.
8. To start the PIL execution, click **Run Generated Code**.

   The MATLAB Coder app:
   - Generates a standalone library, for example, `codegen\lib\kalman01`.
   - Generates PIL interface code, for example, `codegen\lib\kalman01\pil`.
   - Runs the test file, replacing calls to the MATLAB function with calls to the generated code in the library.

- Displays messages from the PIL execution in the **Test Output** tab.

9 Verify that the results from the PIL execution match the results from the original MATLAB functions.

10 To terminate the PIL execution process, click **Stop PIL Verification**. Alternatively, on the **Test Output** tab, click the link that follows `To terminate execution`.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

See Also

Related Examples

- “Generate C Code by Using the MATLAB Coder App”
- “Processor-in-the-Loop Execution From Command Line” on page 90-25
- “Generate Execution Time Profile” on page 82-3
- “Generate Stack Usage Profile” on page 82-13

More About

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2

Processor-in-the-Loop Execution From Command Line

Use processor-in-the-loop (PIL) execution to verify code that you intend to deploy in production.

To set up and start a PIL execution from the command line:

1. Create, register, and verify your target connectivity configuration.
2. Create a `coder.EmbeddedCodeConfig` object.
3. Configure the object for PIL.
4. Use the `codegen` function to generate library code for your MATLAB function and the PIL interface.
5. Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the PIL execution, use the `clear` *`function`*`_pil` or `clear mex` command.

The following example shows how you can use line commands to set up and run a PIL execution on your development computer.

Tutorial Files: Kalman Filter

Open this example to obtain the files for this tutorial:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

PIL Execution of Code Generated for a Kalman Estimator

1 Create a target connectivity API implementation

a In your current working folder, make a local copy of the connectivity classes.

```
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder','+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');
```

b Make the copied files writable.

```
fileattrib(fullfile('+mypil', '*'),'+w');
```

c Update the package name to reflect the new location of the files.

```
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

d Check that you now have a folder `+mypil` in the current folder, which includes three files, `Launcher.m`, `TargetApplicationFramework.m`, and `ConnectivityConfig.m`.

```
dir './+mypil'
```

e Review the code that starts the PIL application. The `mypil.Launcher` class configures a tool for starting the PIL executable. Open this class in the editor.

```
edit(which('mypil.Launcher'))
```

Review the content of this file. For example, consider the `setArgString` method. This method allows additional command line parameters to be supplied to the application. These parameters can include a TCP/IP port number. For an embedded processor implementation, you might have to hard code these settings.

f The class `mypil.ConnectivityConfig` configures target connectivity.

```
edit(which('mypil.ConnectivityConfig'))
```

Review the content of this file. For example:

- The creation of an instance of `rtw.connectivity.RtIOStreamHostCommunicator` that configures the host side of the TCP/IP communications channel.
- A call to the `setArgString` method of `Launcher` that configures the target side of the TCP/IP communications channel.
- A call to `setTimer` that configures a timer for execution time measurement. To define your own target-specific timer for execution time profiling, you must use the Code Replacement Library to specify a replacement for the function `code_profile_read_timer`.

g Review the target-side communication drivers.

```
rtiostreamtcpip_dir=fullfile(matlabroot,'toolbox','coder','rtiostream','src',...
    'rtiostreamtcpip');
edit(fullfile(rtiostreamtcpip_dir,'rtiostream_tcpip.c'))
```

Scroll down to the end of this file. The file contains a TCP/IP implementation of the functions `rtIOStreamOpen`, `rtIOStreamSend`, and`rtIOStreamRecv`. These functions are required for target communication with the host. For each of these functions, you must provide an implementation that is specific to your target hardware and communication channel.

The `mypil.TargetApplicationFramework` class adds target-side communication drivers to the connectivity configuration.

```
edit(which('mypil.TargetApplicationFramework'))
```

The file specifies additional files to include in the build.

2 Register a target connectivity configuration

Use an `rtwTargetInfo.m` file to:

- Create a target connectivity configuration object.
- Invoke `registerTargetInfo`, which registers the target connectivity configuration.

The target connectivity configuration object specifies, for example:

- The configuration name and associated API implementation. See `rtw.connectivity.ConfigRegistry`.
- A toolchain for your target hardware. This example assumes that the target hardware is your host computer, and uses the toolchain supplied for host-based PIL verification. For information about toolchains, see “Custom Toolchain Registration”.

a Insert the following code into your `rtwTargetInfo.m` file, and save the file in the current working folder or in a folder that is on the MATLAB search path:

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain =  rtw.connectivity.Utils.getHostToolchainNames;

% Through the HardwareBoard and TargetHWDeviceType properties,
% define compatible code for the target connectivity configuration
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

b Refresh the MATLAB Coder library registration information.

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

3 Verify target connectivity configuration

Use the supplied `piltest` function to verify your target connectivity configuration.

a Create a `coder.EmbeddedCodeConfig` object for verifying the target connectivity configuration.

```
configVerify = coder.config('lib');
```

b Specify the manufacturer and test hardware type. For example, PIL execution on a 64-bit Windows development computer requires:

```
configVerify.HardwareImplementation.TargetHWDeviceType =...
   'Intel->x86-64 (Windows64)';
configVerify.HardwareImplementation.ProdLongLongMode = true;
```

c Run `piltest`.

```
piltest(configVerify, 'ConfigParam', {'ProdLongLongMode'} )
```

4 Configure the PIL execution

a Create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
```

b Configure the object for PIL.

```
config.VerificationMode = 'PIL';
```

c Specify production hardware, which must match one of the test hardware settings in `rtwTargetInfo.m`. For PIL execution on your development computer, specify settings that match the computer. For example, if your computer is a Windows 64-bit system, specify:

```
config.HardwareImplementation.ProdHWDeviceType =...
   'Intel->x86-64 (Windows64)';
config.HardwareImplementation.ProdLongLongMode = true;
```

For a Linux 64-bit system, set `ProdHWDeviceType` to `'Intel->x86-64 (Linux 64)'`.

For a Mac OS X system, set `ProdHWDeviceType` to `'Intel->x86-64 (Mac OS X)'`.

5 Generate code and run PIL execution

Generate library code for the `kalman01` MATLAB function and the PIL interface, and run the MATLAB test file, `test01_ui`. The test file uses `kalman01_pil`, the generated PIL interface for `kalman01`.

```
codegen -config config -args {zeros(2,1)} kalman01 -test test01_ui
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\pil` — PIL interface code for `kalman01`.

Verify that the output of this run matches the output from the original `kalman01.m` function.

Note On a Windows operating system, the Windows Firewall can potentially block a SIL or PIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

6 Terminate PIL execution

Terminate the PIL execution process.

```
clear kalman01_pil;
```

See Also

Related Examples

- “Generate C Code by Using the MATLAB Coder App”
- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23
- “Generate Execution Time Profile” on page 82-3
- “Generate Stack Usage Profile” on page 82-13

More About

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2

Verification of Code Generation Assumptions

The settings on the **More Settings** > **Hardware** tab specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

At the start of a processor-in-the-loop (PIL) execution, the software verifies the **Hardware** tab settings with reference to the target hardware. The software checks:

- The correctness of settings. For example, the integer bit length in the **Sizes** > **int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

If required, the software generates warnings and errors.

See Also

Related Examples

- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 90-23

SIL/PIL Execution Support and Limitations

Feature		Supported
Output types	Static library	Yes
	Dynamic library	Yes
	Executable	No
Languages	C	Yes
	C++	Yes
Interface types	Inputs	Yes
	Outputs	Yes
	Constant inputs	Yes
	Global data	Yes. SIL and PIL execution supports four types of storage classes on page 99-2 for MATLAB Coder global variables. The synchronization of global data between MATLAB and the SIL or PIL application depends on the type of storage class that you specify: • `ExportedGlobal` (default) — The synchronization of global data between MATLAB and a SIL or PIL application is identical to the synchronization between MATLAB and a MEX function. • `ExportedDefine` — There is no synchronization of global data between MATLAB and the SIL or PIL application. The application uses the values of the global variables in MATLAB at the time of code generation. • `ImportedExtern` and `ImportedExternPointer` — There is no synchronization of global data between MATLAB and the SIL or PIL application. The application uses initial values of global variables, which you specify in the external code. If the global variables are not initialized in the external code, the SIL or PIL execution results are undefined.
	Constant global data	Yes
	Reentrant code	Yes
	Multiple entry points	Yes
Data types	Basic types	Yes
	Enumerated types	Yes
	Structures	Yes

Feature		Supported
	Complex data	Yes
	Fixed-point data	Yes
	Multiword fixed-point data	SIL only
	`char` arrays	Yes
	Empty values	Yes
	Cell arrays	Yes
Size	Scalars	Yes
	Fixed-size arrays	Yes
	Static variable-size arrays	Yes
	Dynamic variable-size size arrays	Yes

See Also

Related Examples

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 90-2

Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization

By default, a SIL or PIL execution performs constant input checking and global data synchronization. Constant input checking compares the value that a test file provides for a constant input argument with the value specified at code generation time. If the values do not match, an error occurs. Global data synchronization makes the values of global variables in the SIL or PIL execution environment consistent with the values in the MATLAB workspace. If a global variable is constant and its value in the SIL or PIL execution environment differs from its value in the MATLAB workspace, an error occurs.

It is possible to speed up a SIL or PIL execution by disabling constant input checking or global data synchronization. However, if you disable these features, the SIL or PIL execution results might differ from the results in MATLAB.

Disable Constant Input Checking or Global Data Synchronization at the Command Line

In a `coder.EmbeddedCodeConfig` object that you configured for SIL or PIL execution:

- To disable constant input checking, set the `SILPILCheckConstantInputs` property to `false`. For example, for an object `cfg`, use this code:

  ```
  cfg.SILPILCheckConstantInputs = false;
  ```

- To disable global data synchronization, set the `SILPILSyncGlobalData` property to `false`. For example, for an object `cfg`, use this code:

  ```
  cfg.SILPILSyncGlobalData = false;
  ```

Disable Constant Input Checking or Global Data Synchronization in the MATLAB Coder App

In the settings for a project that you set up for SIL or PIL execution, on the **Debugging** tab:

- To disable constant input checking, set **Check constant inputs in SIL/PIL** to `No`.
- To disable global data synchronization, set **Synchronize global data in SIL/PIL** to `No`.

See Also

More About

- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 90-4
- “Software-in-the-Loop Execution From Command Line” on page 90-6
- “Constant Input Checking in MEX Functions”
- “Generate Code for Global Data”

SIL and PIL Verification for Deployment on Raspberry Pi

This example shows how to verify the generated code by using the software-in-the-loop (SIL) and processor-in-the-loop (PIL) executions. In this example, the PIL execution uses a Raspberry Pi® hardware board.

Run MATLAB Code

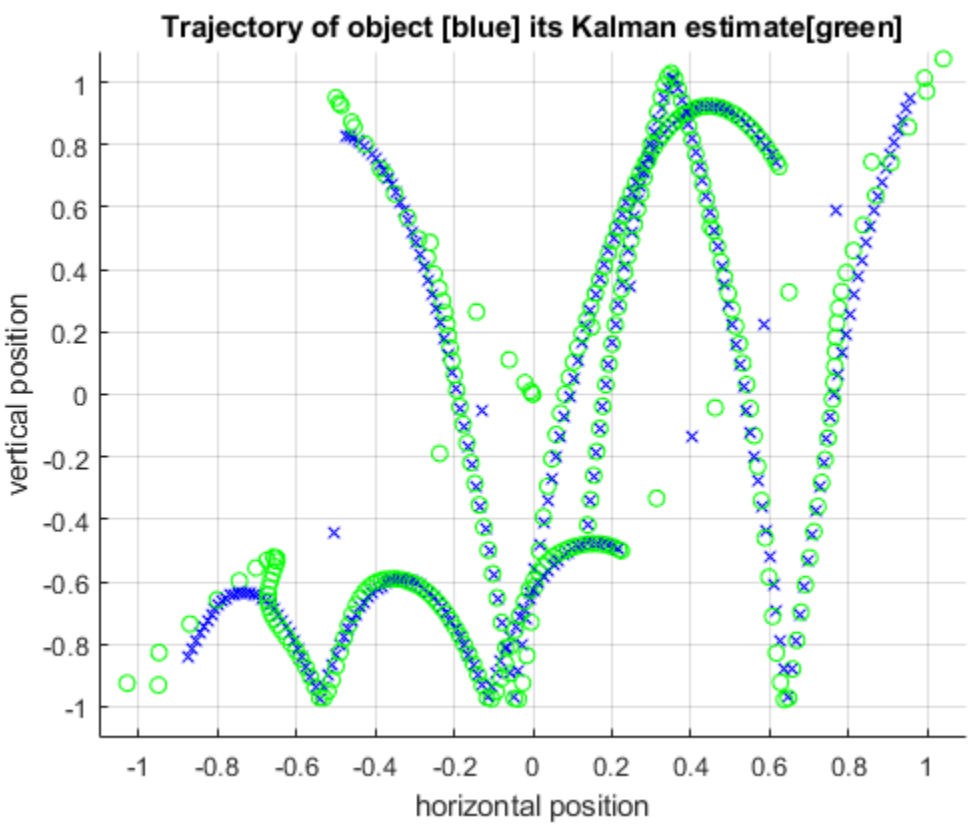
This example uses these supporting files:

- `kalman01.m` contains the MATLAB® function `kalman01` for the Kalman estimator.
- `position.mat` contains the input data about the trajectory of an object.
- `test01_ui.m` is a MATLAB script that loads the input data from `position.mat` into the workspace, calls `kalman01` to estimate the object location by using the Kalman filtering algorithm, and finally calls the `plot_trajectory` function that plots both the actual trajectory of the object and the Kalman estimate.
- `plot_trajectory.m` contains the `plot_trajectory` function described above.

To test the MATLAB code, run the script `test01_ui`.

```
test01_ui
```

```
Current plot held
```




```
Current plot released
```

In the plot that is displayed, you see that the Kalman estimate closely follows the actual trajectory of the object. The test data includes two sudden shifts or discontinuities in position. These discontinuities check that the Kalman filter can quickly readjust its estimate and get back to tracking the object accurately.

For more information, see “C Code Generation for a MATLAB Kalman Filtering Algorithm”.

Software-in-the-Loop (SIL) Execution

In SIL execution, through a MATLAB SIL interface, the code generator compiles and runs library code on your development computer.

Configure SIL Execution

Create a `coder.EmbeddedCodeConfig` object to generate a static C library. Set the `VerificationMode` property to `'SIL'`. Optionally, enable the Microsoft® Visual Studio® debugger for SIL execution.

```
config = coder.config('lib');
config.VerificationMode = 'SIL';
% config.SILDebugging = true;
```

Check that the production hardware setting is set to the default value `'Generic->MATLAB Host Computer'`.

```
disp(config.HardwareImplementation.ProdHWDeviceType);
```

```
Generic->MATLAB Host Computer
```

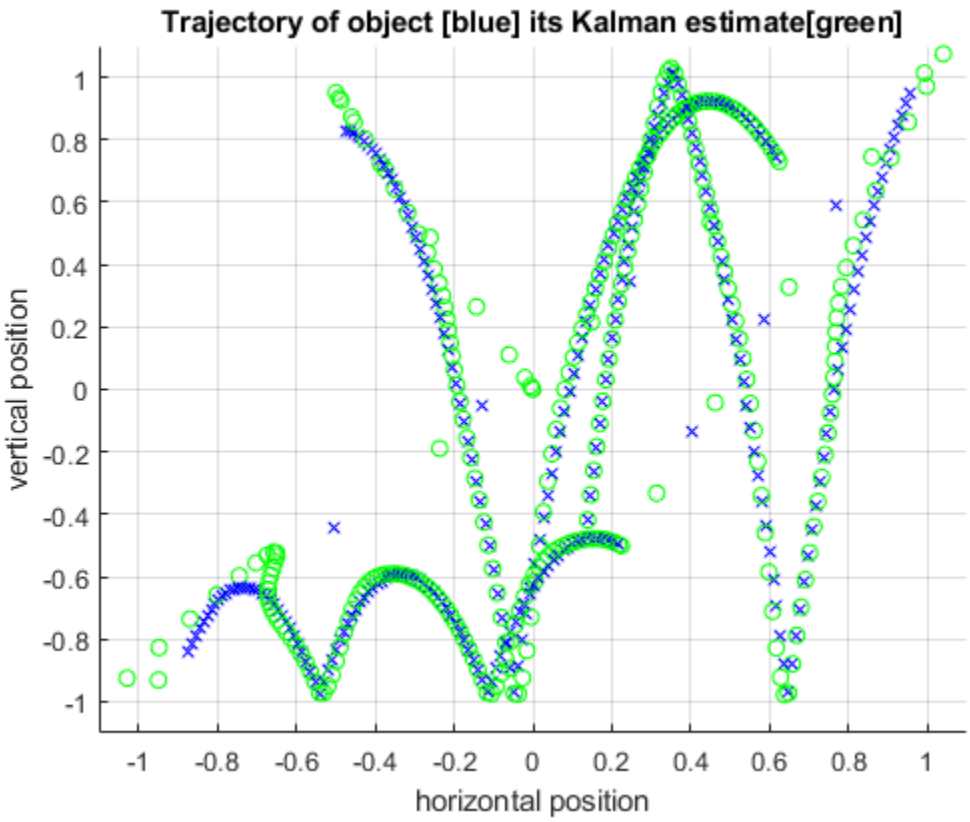
Generate Static Library and Run SIL Execution

Use the `codegen` command with the configuration object `config` to generate a static C library and the SIL interface for the `kalman01` function. To perform SIL execution, you can simply call the generated SIL interface `kalman01_sil` at the MATLAB command line.

In the `codegen` command, the `-test` option enables you to run the test script `test01_ui` with calls to the `kalman01` function replaced by calls to the generated SIL interface `kalman01_sil`.

```
codegen -config config -args {zeros(2,1)} kalman01 -report -test test01_ui
```

```
Code generation successful: View report

Running test file: 'test01_ui' with MEX function 'kalman01_sil.mexw64'.
Current plot held
### Starting SIL execution for 'kalman01'
    To terminate execution: clear kalman01_sil
```




```
Current plot released
```

The code generator creates these output folders:

- `codegen\lib\kalman01` — Static library for `kalman01`.
- `codegen\lib\kalman01\sil` — SIL interface code for `kalman01`.

Observe that the output of this run matches the output from the `kalman01` MATLAB function.

Debug Code During SIL Execution

If you enable the Microsoft Visual Studio debugger, then running the test file opens the Microsoft Visual Studio IDE with debugger breakpoints at the start of the `kalman01_initialize` and `kalman01` functions.

You can use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

- Remove all breakpoints.
- Click the Continue button (**F5**).

The SIL execution runs to completion.

Terminate SIL Execution

Terminate the SIL execution process.

```
clear kalman01_sil;
```

```
### Stopping SIL execution for 'kalman01'
```

You can also use the command `clear mex`, which clears MEX functions from memory.

Usage Notes for Windows® platform

Usage notes for SIL execution on a Windows platform:

- The Windows Firewall can potentially block a SIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.
- Suppose that the MATLAB current working directory (the folder that `pwd` displays) is a UNC path (for example, `\\server\a\b\c`) on a Windows platform. If you launch a SIL executable from this path, the location where the SIL executable is launched can be unpredictable and different from the MATLAB current working directory (for example, `C:\Windows`). To fix this issue, use a mapped network drive for the UNC path as your MATLAB current working directory.

Processor-in-the-Loop (PIL) Execution

In PIL execution, through a MATLAB PIL interface, the code generator cross-compiles and runs production object code on a target processor or an equivalent instruction set simulator. Before you run a PIL execution, you must set up a PIL connectivity configuration for your target. In this example, the PIL execution uses a Raspberry Pi® hardware board.

Configure PIL Execution

Create a `coder.EmbeddedCodeConfig` object to generate a static C library. Set the `VerificationMode` property to `'PIL'`. Set the `Hardware` property to a hardware board configuration object for Raspberry Pi.

```
config = coder.config('lib');
config.VerificationMode = 'PIL';
config.Hardware = coder.hardware('Raspberry Pi');
```

If you are to your Raspherry Pi hardware board for the first time, you must also specify the `Username`, `DeviceAddress`, and `Password` properties of the `config.Hardware` parameter.

Generate Static Library and Run PIL Execution

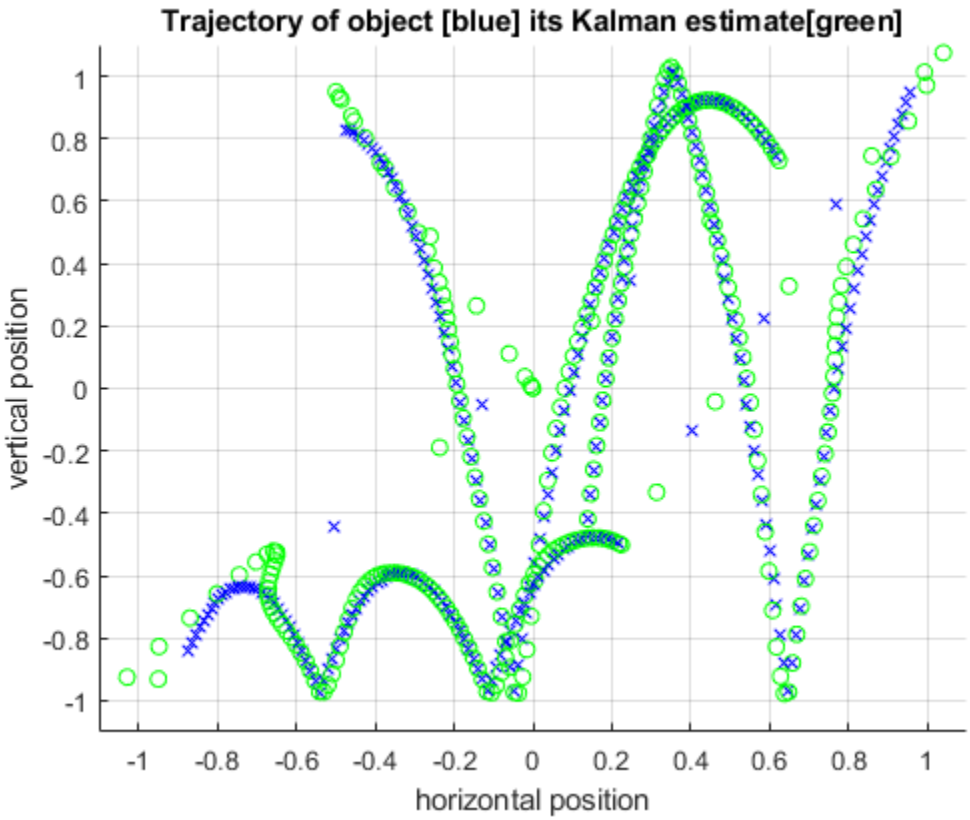
Use the `codegen` command with the configuration object `config` to generate a static C library and the PIL interface for the `kalman01` function. To perform PIL execution, you can simply call the generated PIL interface `kalman01_pil` at the MATLAB command line.

In the `codegen` command, the `-test` option enables you to run the test script `test01_ui` with calls to the `kalman01` function replaced by calls to the generated PIL interface `kalman01_pil`.

```
codegen -config config -args {zeros(2,1)} kalman01 -report -test test01_ui
```

```
### Connectivity configuration for function 'kalman01': 'Raspberry Pi'
Location of the generated elf : /home/pi/MATLAB_ws/R2021b/C/Users/aghosh/Documents/Examples/ecode
Code generation successful: View report
```

```
Running test file: 'test01_ui' with MEX function 'kalman01_pil.mexw64'.
Current plot held
### Connectivity configuration for function 'kalman01': 'Raspberry Pi'
### Starting application: 'codegen\lib\kalman01\pil\kalman01.elf'
    To terminate execution: clear kalman01_pil
### Launching application kalman01.elf...
```




```
Current plot released
```

The code generator creates these output folders:

- `codegen\lib\kalman01` — Static library for `kalman01`.
- `codegen\lib\kalman01\pil` — PIL interface code for `kalman01`.

Observe that the output of this run matches the output from the `kalman01` MATLAB function.

The Windows Firewall can potentially block a PIL execution. To allow the execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

Terminate PIL Execution

Terminate the PIL execution process.

```
clear kalman01_pil;
```

```
### Host application produced the following standard output (stdout) and standard error (stderr)
```

You can also use the command `clear mex`, which clears MEX functions from memory.

See Also

`codegen` | `coder.config` | `coder.EmbeddedCodeConfig`

More About

- "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" on page 90-2

Software-in-the-loop Execution For MATLAB Function with Multiple Signatures

In this section...

"Generate Multisignature SIL MEX File for an Entry-Point Function" on page 90-39
"Generate Multisignature SIL MEX File for Multiple Entry-Point Functions" on page 90-40

For MATLAB function with multiple signatures, software-in-the-loop (SIL) enables you to verify the generated C/C++ code by using a SIL MEX file that supports multiple signatures. This feature enables you to generate one SIL MEX file for verification instead of generating separate SIL MEX files for each signature.

Generate Multisignature SIL MEX File for an Entry-Point Function

1. **MATLAB code for `myAdd` function**

   Copy the MATLAB code of one entry-point function `myAdd`, and save it in your working directory.

   The entry-point function, `myAdd`, returns the sum of two values.

   ```
   function y = myAdd(a,b)
   %#codegen
   y = a+b;
   end
   ```

2. **Configure SIL execution**

   Create a `coder.EmbeddedCodeConfig` object and configure the object for SIL.

   ```
   cfg = coder.config('lib');
   cfg.VerificationMode = "SIL";
   ```

3. **Generate code using `codegen` command**

   The `codegen` command generates a SIL MEX file `myAdd_sil` that works with three different data types: `double`, `int8`, and `vector of doubles`. You specify the entry-point function followed by `-args` for each signature of the entry-point function.

   ```
   codegen -config cfg myAdd -args {1,2} -args {int8(2),int8(3)} -args {1:10,1:10} -report
   ```

4. **Run generated SIL MEX file `myAdd_sil`**

   At the command prompt, call the generated SIL MEX file `myAdd_sil`. Make sure that the values you pass to `myAdd_sil` match the input properties that you specified in the `codegen` command.

   ```
   myAdd_sil(3,4)

   ans =

        7

   myAdd_sil(int8(5),int8(6))

   ans =

     int8
   ```

```
   11
```

```
myAdd_sil(1:10,2:11)
```

```
ans =

     3     5     7     9    11    13    15    17    19    21
```

Running the MATLAB function `myAdd` with these input values produces the same output. These test cases verify that `myAdd` and `myAdd_sil` have the same behavior.

5 **Terminate SIL execution**

Terminate the SIL execution process.

```
clear myAdd_sil;
```

You can also use the command `clear mex`, which clears MEX files from memory.

6 **Entry Point Summary Report**

In the generated code, different signatures are mapped to versioned entry-point functions `myAdd1`, `myAdd2`, and `myAdd3` as shown in the figure.

Generate Multisignature SIL MEX File for Multiple Entry-Point Functions

1 **MATLAB code for `myAdd` and `myMul` functions**

Copy the MATLAB codes of two entry-point functions, `myAdd` and `myMul` and save it in your working directory.

The first entry-point function, `myAdd`, returns the sum of two values:

```
function y = myAdd(a,b)
%#codegen
```

```
y = a+b;
end
```

The second entry-point function, `myMul`, returns the multiplication of two values:

```
function y = myMul(a,b)
%#codegen
y = a*b;
end
```

2 **Configure SIL execution**

Create a `coder.EmbeddedCodeConfig` object and configure the object for SIL.

```
cfg = coder.config('lib');
cfg.VerificationMode = "SIL";
```

3 **Generate code using `codegen` command**

The `codegen` command generates a SIL MEX file `myMath_sil` for the signatures that you specified in the `codegen` command. The SIL MEX file `myMath_sil` works for `myAdd` and `myMul` with two different data types: `double`, and `int8`. You specify the entry-point function followed by `-args` for each signature of the entry-point function.

```
codegen -config cfg myAdd.m -args {1,2} -args {int8(1),int8(2)} myMul.m -args {1,2} -args {in
```

4 **Run generated SIL MEX file `myMath_sil`**

At the command prompt, call the generated SIL MEX file `myMath_sil`. Make sure that the values you pass to `myMath_sil` match the input properties that you specified in the `codegen` command.

```
myMath_sil("myAdd",3,4)

ans =

     7

myMath_sil("myAdd",int8(5),int8(6))

ans =

   int8

   11

myMath_sil("myMul",3,4)

ans =

     12

myMath_sil("myMul",int8(5),int8(6))

ans =

    int8

    30
```

Running the MATLAB function `myAdd` and `myMul` with these input values produces the same output. These test cases verify that `myAdd`, `myMul` and the generated SIL MEX file `myMath_sil` have the same behavior.

5 **Terminate SIL execution**

Terminate the SIL execution process.

```
clear myMath_sil;
```

You can also use the command `clear mex`, which clears MEX files from memory.

6 **Entry Point Summary Report**

In the generated code, different signatures are mapped to versioned entry-point functions `myAdd1`, `myAdd2`, `myMul1` and `myMul2` as shown in the figure.

Example main file provides information about how to use entry-points in your application. “Incorporate Generated Code Using an Example Main Function”, helps to generate example main.c file.

Note You can implement the multisignature PIL MEX for an entry-point function and multiple entry-point functions in the same way that the multisignature SIL MEX is implemented.

Limitations

The multisignature SIL MEX and PIL MEX generation does not have graphical user interface (GUI) support.

See Also

`codegen` | `coder.config`

More About

- “Generate Code for Multiple Entry-Point Functions”
- “Generate One MEX Function for Multiple Signatures”

Code Coverage in Embedded Coder

- “Simulink Code Coverage Metrics” on page 91-2
- “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 91-6
- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 91-11
- “Configure Code Coverage Programmatically” on page 91-13
- “Code Coverage Summary and Annotations” on page 91-15
- “Code Coverage Tool Support” on page 91-20
- “Tips and Limitations” on page 91-21
- “Collect Code Coverage Metrics with a Third-Party Tool” on page 91-25

Simulink Code Coverage Metrics

In this section...

"Statement Coverage" on page 91-2
"Condition Coverage" on page 91-2
"Decision Coverage" on page 91-3
"Modified Condition/Decision Coverage (MCDC)" on page 91-4
"Cyclomatic Complexity" on page 91-4
"Relational Boundary Coverage" on page 91-5

If you have a Simulink Coverage license, you can run a SIL or PIL simulation that produces code coverage metrics for generated model code. The simulation performs several types of code coverage analysis.

Statement Coverage

Statement coverage measures the number of source code statements that execute when the code runs. Use this type of coverage to determine whether every statement in the program has been invoked at least once.

The percentage of statement coverage is represented by the following equation:

Statement coverage = (Number of executed statements / Total number of statements) *100

Statement Coverage Example

This code snippet contains five statements. To achieve 100% statement coverage, you need at least one test with positive *x* values, one test with negative *x* values, and one test with *x* values of zero.

```
if (x > 0)
  printf( "x is positive" );
else if (x < 0)
  printf( "x is negative" );
else
  printf( "x is 0" );
```

Condition Coverage

Condition coverage analyzes statements that include conditions in source code. Conditions are C/C++ Boolean expressions that contain relation operators (<, >, <=, or >=), equation operators (!= or ==), or logical negation operators (!), but that do not contain logical operators (&& or ||). This type of coverage determines whether every condition has been evaluated to all possible outcomes at least once.

The percentage of condition coverage is represented by the following equation:

Condition coverage = (Number of executed condition outcomes / Total number of condition outcomes) *100

Condition Coverage Example

In this expression:

```
y = x<=5 || x!=7;
```

there are two conditions:

```
x<=5
x!=7
```

To achieve 100% condition coverage, your test cases need to demonstrate a true and false outcome for both conditions. For example, a test case where *x* is equal to 4 demonstrates a true case for both conditions, and a case where *x* is equal to 7 demonstrates a false case for both conditions.

Boolean Assignment Statements

Code coverage analyzes assignment statements that contain Boolean values and reports them as a condition. Model coverage only analyzes logical expressions in assignment statements, meaning only expressions that contain a logical operator such as the logical `AND` (&&) or the logical `OR` (`||`). This difference can result in a discrepancy between model and code coverage results and can sometimes result in unsatisfiable condition outcomes in the code coverage analysis.

For example, consider the following statement:

```
bool A = true;
```

During code coverage analysis, Simulink Coverage analyzes this statement for condition coverage. The statement is true at every time step, so the result is that you get 50% condition coverage on this statement because the false case cannot occur. Model coverage does not analyze this statement, so it also creates a discrepancy between model and code coverage results.

Decision Coverage

Decision coverage analyzes statements that represent decisions in source code. Decisions are Boolean expressions composed of conditions and one or more of the logical C/C++ operators && or `||`. Conditions within branching constructs (if/else, while, and do-while) are decisions. Decision coverage determines the percentage of the total number of decision outcomes the code exercises during execution. Use this type of coverage to determine whether all decisions, including branches, in your code are tested.

Note The decision coverage definition for DO-178C compliance differs from the Simulink Coverage definition. For decision coverage compliance with DO-178C, in the Configuration Parameters, set the **Structural Coverage Level** to `Condition Decision` for Boolean expressions not containing && or || operators.

The percentage of decision coverage is represented by the following equation:

Decision coverage = (Number of executed decision outcomes / Total number of decision outcomes) *100

Decision Coverage Example

This code snippet contains three decisions:

```
y = x<=5 && x!=7;                // decision #1

if( x > 0 )                    // decision #2
  printf( "decision #2 is true" );
else if( x < 0 && y )                // decision #3
  printf( "decision #3 is true" );
else
  printf( "decisions #2 and #3 are false" );
```

To achieve 100% decision coverage, your test cases must demonstrate a true and false outcome for each decision.

Modified Condition/Decision Coverage (MCDC)

Modified condition/decision coverage (MCDC) analyzes whether the conditions within decisions independently affect the decision outcome during execution. To achieve 100% MCDC, your test cases must demonstrate:

- All conditions within decisions have been evaluated to all possible outcomes at least once.
- Every condition within a decision independently affects the outcome of the decision.

The percentage of MCDC is represented by the following equation:

MCDC coverage = (Number of conditions evaluated to all possible outcomes affecting the outcome of the decision / Total number of conditions within the decisions) *100

Modified Condition/Decision Coverage Example

For this decision:

```
X || ( Y && Z )
```

the following set of test cases delivers 100% MCDC coverage.

	X	Y	Z
Test case #1	0	0	1
Test case #2	0	1	0
Test case #3	0	1	1
Test case #4	1	0	1

In order to demonstrate that the conditions Y and Z can independently affect the decision outcome, the condition X must be false for those test cases. If the condition X is true, then the decision is already known to be true. Therefore, the conditions Y and Z would not affect the decision outcome.

Cyclomatic Complexity

Cyclomatic complexity measures the structural complexity of code by using the McCabe complexity measure. To compute the cyclomatic complexity of code, code coverage uses this formula:

$$c = \sum_{1}^{N} (o_n - 1)$$

N is the number of decisions in the code. o_n is the number of outcomes for the nth decision point. Code coverage adds 1 to the complexity number for each C/C++ function.

Coverage Example

For this code snippet, the cyclomatic complexity is 3:

```
void evalNum(int x)
{
  if (x > 0)                          // decision #1
    printf( "x is positive" );
  else if (x < 0)                     // decision #2
    printf( "x is negative" );
  else
    printf( "x is 0" );
}
```

The code contains one function that has two decision points. Each decision point has two outcomes. Using the preceding formula, N is 2, o_1 is 2, and o_2 is 2. Code coverage uses the formula with these decisions and outcomes and adds 1 for the function. The cyclomatic complexity for this code snippet is:

$$c = (o_1 - 1) + (o_2 - 1) + 1 = (2 - 1) + (2 - 1) + 1 = 3$$

Relational Boundary Coverage

Relational boundary code coverage examines code that has relational operations. Relational boundary code coverage metrics align with those for model coverage, as described in “Relational Boundary Coverage” (Simulink Coverage). Fixed-point values in your model are integers during code coverage.

See Also

Related Examples

- “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 91-6

Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode

If you have Embedded Coder and Simulink Coverage, you can analyze coverage for generated code during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation.

In this section...

"Enable SIL or PIL Code Coverage for a Model" on page 91-6
"Review the Coverage Results for Models in SIL or PIL Mode" on page 91-6
"Limitations" on page 91-8

Enable SIL or PIL Code Coverage for a Model

To record SIL or PIL code coverage for a model:

1. Ensure you have a supported compiler installed. For a list of supported compilers, see Supported and Compatible Compilers.
2. In the Configuration Parameters dialog box, on the left pane, click **Code Generation**. From the list, select **Verification**.
3. Under **Code profiling**, set **Measure function execution times** to `Off`.
4. Under **Code coverage for SIL or PIL**, set **Third-party tool** to `None (use Simulink Coverage)`.
5. Enable coverage for a model in SIL or PIL mode or a reference model in SIL or PIL mode. For more information about enabling coverage, see "Specify Coverage Options" (Simulink Coverage).
6. Run a SIL or PIL simulation.

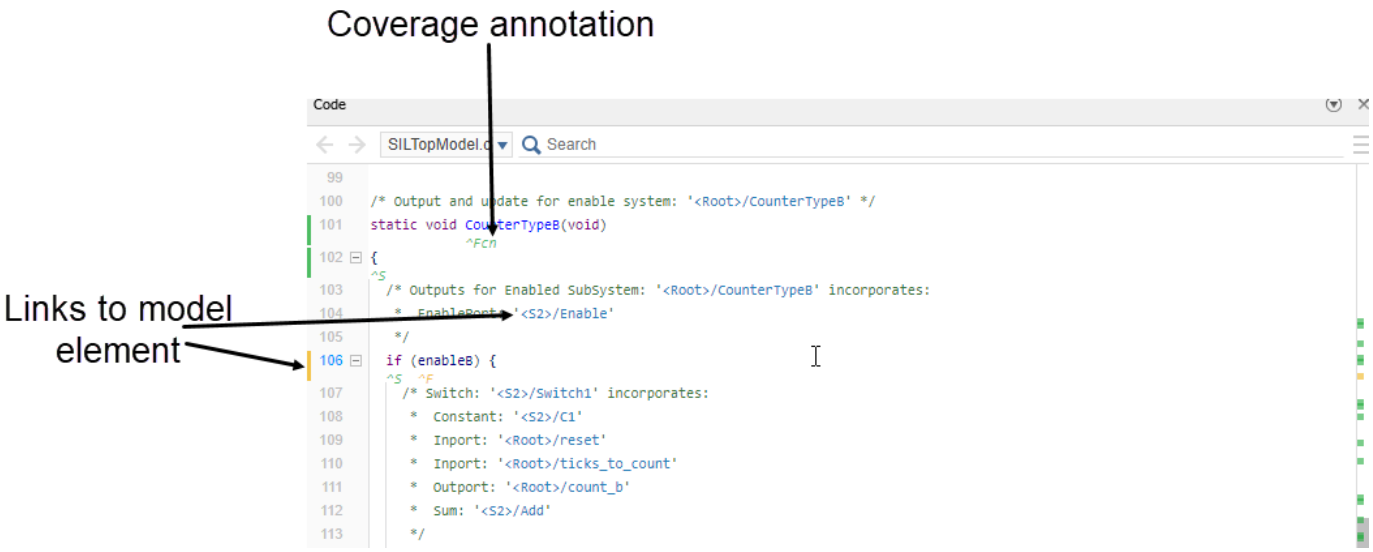
   Note The **Coverage (Run)** button in the Coverage toolstrip forces a Normal mode simulation for the top-level model and does not yield SIL or PIL code coverage.

   To run a SIL or PIL simulation:

   a. In the Simulink window, click **Apps** and then click **SIL/PIL Manager**.
   b. In the **SIL/PIL** tab, click **Automated Verification > SIL/PIL Simulation Only**.
   c. Click **Run SIL/PIL**.

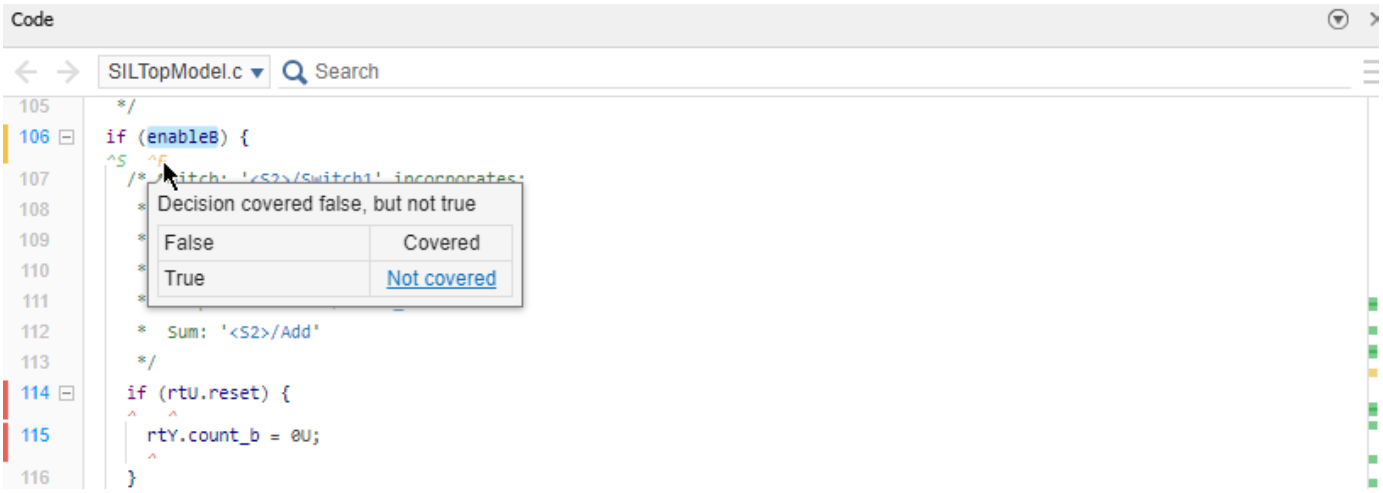
Review the Coverage Results for Models in SIL or PIL Mode

To view the code coverage information in the **Code** pane, click the menu icon to the right of the search box and select **Show code coverage**. If the option is disabled, then on the **Coverage** tab, click **Coverage Highlighting**. The code displays highlighting and annotations that show code coverage information. You can navigate from the code to the associated model blocks by using the links in the line numbers, code elements, and comments.

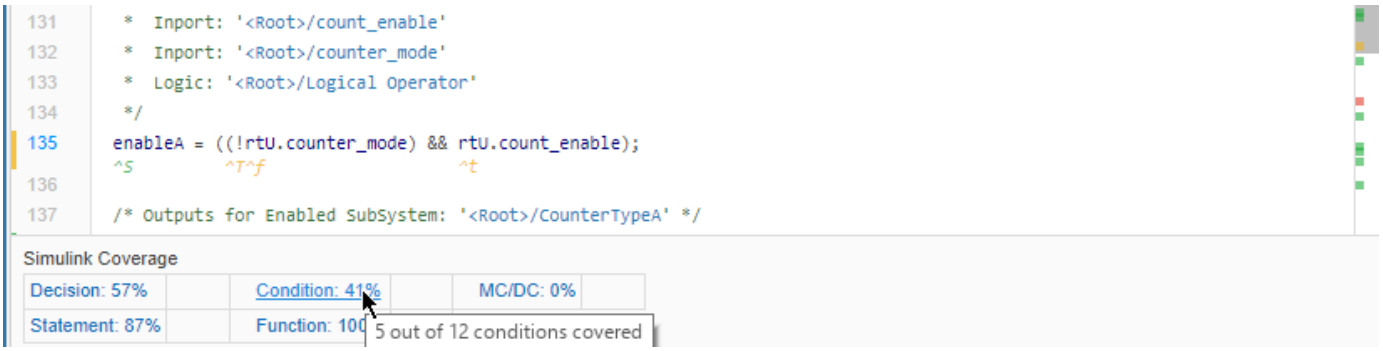


If you point your cursor at a coverage annotation, a tooltip with additional information appears.

If coverage is missing, you can click the **Not covered** link in the tooltip to open the **Filter Explorer** pane of the Coverage Results Explorer with a new justification rule for the outcome you selected.



At the bottom of the Code view, Simulink Coverage shows a summary of the code coverage report. Point your cursor at one of the listed metrics to view a tooltip with additional information.



Limitations

Coverage for models in SIL and PIL mode has these limitations:

- The model must meet the requirements listed in “Enable SIL or PIL Code Coverage for a Model” on page 91-6.
- Code coverage results must not include external C/C++ files in read-only folders.
- The **Coverage (Run)** button in the Coverage toolstrip forces a Normal simulation and will not yield SIL or PIL code coverage.
- The **Code** pane does not support annotations or tooltips for Modified Condition Decision Coverage (MCDC) outcomes. As a result, you cannot justify these outcomes from the **Code** pane.
- The **Code** pane does not support creating exclusion filter rules. To create exclusion filter rules, use the coverage report.

See Also

Related Examples

- “Custom Toolchain Directives Required for Code Coverage and Execution Profiling” on page 88-68
- “Software-in-the-Loop Code Coverage” (Simulink Coverage)
- “SIL/PIL Manager Verification Workflow” on page 88-76

Configure Code Coverage with Third-Party Tools

During a top-model or Model block SIL or PIL simulation, you can collect code coverage metrics for generated code using a third-party tool. Embedded Coder supports the following tools:

- The LDRA tool suite from LDRA Software Technology. For information about installing and using this tool, go to https://ldra.com.

  The software supports LDRA code coverage for SIL and PIL.

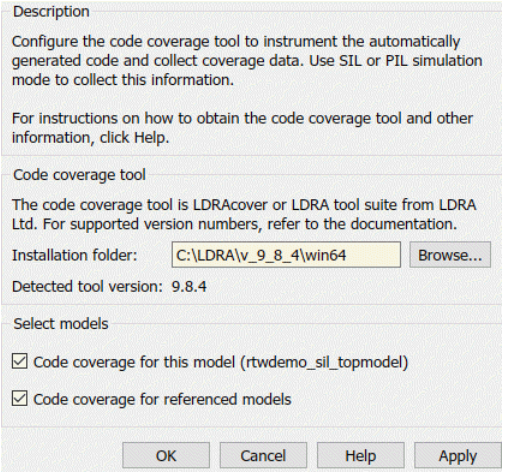
- BullseyeCoverage from Bullseye Testing Technology. For information about installing and using this tool, go to www.bullseye.com.

  The software supports BullseyeCoverage code coverage for SIL and, in certain cases, PIL.

Note SIL and PIL simulation support for BullseyeCoverage will be removed in a future release.

To configure a code coverage tool for a top-model or Model block SIL or PIL simulation, in the Configuration Parameters dialog box, select the **Verification** pane:

1. From the **Third-party tool** drop-down list, select a tool, for example, `BullseyeCoverage` or `LDRAcover or LDRA tool suite`.
2. Click **Configure** to open the Code Coverage Settings dialog box.
3. In the **Installation folder** field, specify the location where your coverage tool is installed. If you click **Browse**, the Select Installation Folder dialog box opens, which allows you to navigate to the folder where your coverage tool is installed. The software detects and displays the tool version.



   By default, the following parameters are enabled:

   - **Code coverage for this model** — Generate coverage data for the current (top) model.
   - **Code coverage for referenced models** — Generate data for models referenced by the current (top) model.

   If your top model has Model blocks, these parameters of the top model override the corresponding parameters of referenced models.

4 Click **OK**.

With the LDRA tool suite:

- The evaluation of cumulative code coverage begins from the point when you last added a new file to the existing set of source files. For example, existing code coverage results are deleted when you:
  - Run a simulation with a new model using the existing code generation folder.
  - Run a simulation that results in additional source code files being instrumented.
- If you switch between SIL and PIL simulations of a model, the software generates separate cumulative code coverage results for the SIL and PIL simulations.

For a model in a reference hierarchy, the software does not support simultaneous function execution time measurement and code coverage.

See Also

Related Examples

- “Configure and Run SIL Simulation” on page 88-15
- “SIL/PIL Manager Verification Workflow” on page 88-76
- “Configure Code Coverage Programmatically” on page 91-13
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 91-11
- “Collect Code Coverage Metrics with a Third-Party Tool” on page 91-25
- “Code Coverage Tool Support” on page 91-20
- “Minor SIL and PIL Differences for LDRA Tool Suite” on page 91-22
- “PIL Support for BullseyeCoverage” on page 91-23
- “Simulink Code Coverage Metrics” on page 91-2
- “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” on page 91-6

External Websites

- https://ldra.com
- www.bullseye.com

View Code Coverage Information at the End of SIL or PIL Simulations

In this section...

"View LDRA Tool Suite Results" on page 91-11
"View BullseyeCoverage Results" on page 91-11

If you configure third-party code coverage for a SIL or PIL simulation, when the simulation is complete, the code generation report opens automatically and you see hyperlinks in the Command Window.

View LDRA Tool Suite Results

If you specified the LDRA tool suite, you see three links in the Command Window:

```
...
### Starting SIL simulation for component: SILTopModel
### Stopping SIL simulation for component: SILTopModel
### Starting analysis of coverage data
### Use the following links to view code coverage results:
    LDRAcover or LDRA tool suite GUI
    LDRAcover or LDRA tool suite Code Coverage Overview Report
    HTML code generation report with code coverage annotations
### Completed code coverage analysis
>>
```

To open the LDRA GUI, click the `LDRAcover or LDRA tool suite GUI` link.

To open the LDRA report with your Web browser, click the `LDRAcover or LDRA tool suite Code Coverage Overview Report` link. For information about using the report, refer to the LDRA documentation.

LDRA analysis results for all code in the current code generation folder belong to a set. In this set, you can find analysis results for models that share the same code generation folder. The `LDRAcover or LDRA tool suite Code Coverage Overview Report` link identifies the location of the LDRA analysis results, which is determined by:

- The LDRA tool suite configuration.
- The name of the LDRA tool suite set associated with the current code generation folder.

To view summary data and code annotations with coverage information in the code generation report, click the `HTML code generation report with code coverage annotations` link. To view the code coverage information in the Code view, from the drop-down list to the right of the search box, select **Show code coverage**.

View BullseyeCoverage Results

Note SIL and PIL simulation support for BullseyeCoverage will be removed in a future release.

If you specified the BullseyeCoverage tool, you see two links in the Command Window:

```
### Starting SIL simulation for component: SILTopModel
### Stopping SIL simulation for component: SILTopModel
```

```
### Processing code coverage data
### Use the following links to view code coverage results:
    BullseyeCoverage browser (coverage for last run)
    HTML code generation report (cumulative coverage)
### Completed code coverage analysis
>>
```

To view the coverage report using the BullseyeCoverage Browser, click the `BullseyeCoverage browser (coverage for last run)` link.

The BullseyeCoverage Browser shows coverage data for instrumented files associated with your latest top-model simulation. The coverage data shown in the browser is not cumulative and pertains only to the most recent simulation. For information about the BullseyeCoverage Browser, go to www.bullseye.com.

To view summary data and code annotations with coverage information in the code generation report, click the `HTML code generation report (cumulative coverage)` link. To view the code coverage information in the Code view, from the drop-down list to the right of the search box, select **Show code coverage**.

See Also

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “Collect Code Coverage Metrics with a Third-Party Tool” on page 91-25
- “Code Coverage Summary and Annotations” on page 91-15

Configure Code Coverage Programmatically

For a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, you can configure third-party code coverage analysis by using command-line APIs.

Set Up Third Party Code Coverage From Command Line

For BullseyeCoverage, you can use this workflow:

1. Using `get_param`, retrieve the object containing coverage settings for the current model. For example:

   ```
   covSettings = get_param(gcs, 'CodeCoverageSettings')
   ```

   ```
   covSettings =

     CodeCoverageSettings with properties:

              TopModelCoverage: 'off'
       ReferencedModelCoverage: 'off'
                  CoverageTool: 'None'
   ```

   The property `TopModelCoverage` determines whether the software generates code coverage data for just the top model, while `ReferencedModelCoverage` determines whether the software generates coverage data for models referenced by the top model. If neither property is `'on'`, the code generator does not produce code coverage data during a SIL or PIL simulation.

   When you save your model, the properties `TopModelCoverage`, `ReferencedModelCoverage`, and `CoverageTool` are also saved.

2. Check the class of `covSettings`.

   ```
   class(covSettings)
   ```

   ```
   ans =

       'coder.coverage.CodeCoverageSettings'
   ```

3. Turn on BullseyeCoverage code coverage analysis for the top model and referenced models.

   ```
   covSettings.TopModelCoverage='on';
   covSettings.ReferencedModelCoverage='on';
   covSettings.CoverageTool='BullseyeCoverage';
   ```

   To specify the LDRA tool suite as the code coverage tool, set the property `CoverageTool` to `'LDRAcover or LDRA tool suite'`.

4. Using `set_param`, apply the new coverage settings to the model.

   ```
   set_param(gcs,'CodeCoverageSettings', covSettings);
   ```

5. Assuming the third-party code coverage tool is installed, specify the installation path.

   ```
   coder.coverage.BullseyeCoverage.setPath('C:\Program Files\BullseyeCoverage')
   ```

   For the LDRA tool suite, use `coder.coverage.LDRA.setPath('C:\...)`.

6. Check that the path is saved as a preference.

   ```
   coder.coverage.BullseyeCoverage.getPath
   ```

   For the LDRA tool suite, use `coder.coverage.LDRA.getPath`.

Code Coverage for Model That Uses Configuration Reference

If your model uses a configuration set that is not attached to the model, for example, a configuration reference, then the **Configure** button is dimmed in the Configuration Parameters dialog box. You cannot open the Code Coverage Settings dialog box and you must use line commands to specify code coverage settings.

To configure third-party code coverage analysis for a model that uses a configuration reference:

1. Get the active configuration set from the model.

   ```
   cs = getActiveConfigSet(gcs)
   ```

2. Retrieve the current code coverage settings.

   ```
   covSettings = get_param(cs, 'CodeCoverageSettings')
   ```

3. Specify the code coverage settings that you require. For example:

   ```
   covSettings.TopModelCoverage = 'on';
   covSettings.ReferencedModelCoverage='on';
   covSettings.CoverageTool='BullseyeCoverage';
   ```

4. Get the configuration set that is specified by the configuration reference.

   ```
   csBase = cs.getRefConfigSet
   ```

5. Apply the required code coverage settings to the referenced configuration set.

   ```
   set_param(csBase, 'CodeCoverageSettings', covSettings);
   ```

   The active configuration set for the model contains the code coverage settings that you require.

See Also

Related Examples

- “Collect Code Coverage Metrics with a Third-Party Tool” on page 91-25
- “Configure Code Coverage with Third-Party Tools” on page 91-9

Code Coverage Summary and Annotations

In this section...
“LDRA Tool Suite Coverage” on page 91-15 “BullseyeCoverage Information” on page 91-17

If you specify a code coverage tool for a SIL or PIL simulation, the software provides summary data and code annotations with coverage information in the Code view. Each code annotation is associated with a code feature and indicates the nature of the feature coverage during code execution. The Code view also allows you to navigate easily between blocks in your model and the corresponding sections in the source code.

LDRA Tool Suite Coverage

The cumulative coverage data in the Code view is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from simulations with other top models that share referenced models with your current top model.

The software provides LDRA tool suite annotations in the Code view to help you to review code coverage.

Note Do not use the Code view alone to verify that you have achieved your coverage goals. You must refer to the LDRA report.

In the Code view, the color of each bar and annotations shows the amount of coverage of the corresponding line of code or code feature. Green indicates full coverage, yellow indicates partial coverage, and red indicates no coverage.

```
104   */
105 if (enableB) {
                ^b
106   /* Switch: '<S2>/Switch1' incorporates:
107    *  Constant: '<S2>/C1'
108    *  Inport: '<Root>/reset'
109    *  Inport: '<Root>/ticks_to_count'
110    *  Outport: '<Root>/count_b'
111    *  Sum: '<S2>/Add'
112    */
113   if (rtU.reset) {
                  ^
114     rtY.count_b = 0U;
115   }
116
117   rtY.count_b = (uint8_T)(uint32_T)((uint32_T)rtU.ticks_to_count + (uint32_T)
118     rtY.count_b);
119
120   /* End of Switch: '<S2>/Switch1' */
121 }
```

Place your cursor over the annotation ^b to see a tooltip. the tooltip indicates which branch destinations are covered and which are not covered.

```
104   */
105 if (enableB) {
                ^b
106   /* Switch: Branch destinations covered <line>c<column>: 123c3; branch destinations not covered: 105c16
107    *  Constant: '<S2>/C1'
108    *  Inport: '<Root>/reset'
109    *  Inport: '<Root>/ticks_to_count'
110    *  Outport: '<Root>/count_b'
111    *  Sum: '<S2>/Add'
112    */
113   if (rtU.reset) {
                  ^
114     rtY.count_b = 0U;
115   }
```

This table describes the LDRA code annotations that you might see in the Code view produced by SIL and PIL simulations.

Code feature	Annotation symbol	What happened during simulation
Function	Fcn	Function *name* returned through this exit point.
	=>	Function *name* never returned through this exit point.
Branch/condition	=>	Condition not encountered.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.
	tf	Condition evaluated both true and false.
Branch/decision	=>	Branch never encountered.
	=>b	Branch to at least one destination covered and branch to at least one other destination not covered.
	b	Branch fully exercised.
Modified Condition/Decision Coverage (MC/DC)	=>mc	Condition did not independently affect outcome of decision.
	mc	Condition independently affected outcome of decision.
Statement		Statements associated with line covered.
		Not all statements associated with line covered.
Code that is reformatted by LDRA tool suite and does not match the original source code. For example, source code with `#include` statements to include other files, and source code with `#define` statements for macros. For detailed coverage information, refer to the LDRA report.	=>Σ	Zero coverage — probes within source code line or files included by source code line not exercised.
	=>Σ	Coverage probes within source code line or any included file partially exercised.
	Σ	Coverage probes within source code line or included files fully exercised.

BullseyeCoverage Information

The cumulative coverage data in the Code view is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from simulations with other top models that share referenced models with your current top model.

The software provides BullseyeCoverage annotations in the Code view to help you to review code coverage.

In the Code view, the color of each bar and annotations shows the amount of coverage of the corresponding line of code or code feature. Green indicates full coverage, yellow indicates partial coverage, and red indicates no coverage.

```
=> F 105  if (enableB) {
     106    /* Switch: '<S2>/Switch1' incorporates:
     107     *  Constant: '<S2>/C1'
     108     *  Inport: '<Root>/reset'
     109     *  Inport: '<Root>/ticks_to_count'
     110     *  Outport: '<Root>/count_b'
     111     *  Sum: '<S2>/Add'
     112     */
```

This table describes the BullseyeCoverage code annotations that you might see in a code generation report produced by a SIL simulation.

Code feature	Annotation symbol	What happened during simulation
Decision	=>	Decision not executed.
	TF	Decision evaluated both true and false.
	=>T	Decision evaluated true only.
	=>F	Decision evaluated false only.
Function	=>	Function not called.
	Fcn	Function called.
Switch label	=>	Switch command not used.
	Sw	Switch command used.
Constant	k	Decision or condition was constant, which did not allow any variation in coverage.
Condition	=>	Condition not encountered.
	tf	Condition evaluated both true and false.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.
Try	=>	Try block never completed.
	Try	Try block covered.
Catch	=>	Catch block not covered.
	Cat	Catch block covered.

See Also

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 91-11
- “Trace Simulink Model Elements in Generated Code” on page 85-8
- “Collect Code Coverage Metrics with a Third-Party Tool” on page 91-25

Code Coverage Tool Support

Embedded Coder code coverage provides the following support for BullseyeCoverage and the LDRA tool suite.

Operating system	BullseyeCoverage		LDRA Tool Suite	
	Version supported	Compiler supported	Version supported	Compiler supported
Windows	8.15.41	Microsoft Visual C++ (MSVC)	9.8.4	• Microsoft Visual C++ (MSVC) • LCC • MinGW
Linux	8.15.41	`gcc`	9.8.4	`gcc`
Mac	Not supported		9.8.4	• Clang • `gcc`

See Also

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “Select and Configure C or C++ Compiler”

Tips and Limitations

Model Build and SIL/PIL Blocks Not Supported

Code coverage does not support:

- The model build process, for example, the `Ctrl+B` command.
- SIL or PIL blocks.

Code coverage settings are ignored by the `Ctrl+B` command and SIL or PIL blocks.

BullseyeCoverage License Wait

When you build your model, you might have to wait for a BullseyeCoverage license. If you want to see information about the wait, before you build your model, in the Configuration Parameters dialog box, select **Verbose build**.

Note SIL and PIL simulation support for BullseyeCoverage will be removed in a future release.

Current Working Folder Cannot be UNC Path

If your MATLAB current working folder is a Universal Naming Convention (UNC) path, code coverage fails.

Characters in matlabroot and File Path

If `matlabroot` or the path to your generated files contains a space or the `.` (period) character, code coverage might fail.

Header Files with Identical Names

Consider a model that is configured for LDRA code coverage. During the build process, if the software detects two header files with the same name in the folder for generated code, the software generates an error.

Code Coverage for Source Files in Shared Utility Folders

The software supports code coverage for source files generated in shared utility folders. If you configure code coverage for a model that uses shared utility code generation, when you build the model, you also build all source files in the shared utilities folder with code coverage enabled.

Whenever you build a model, the code coverage settings of the model must be consistent with source files that you previously built in the shared utilities folder. Otherwise, the software reports that code in the shared utilities folder is inconsistent with the current model configuration and must be rebuilt. For example, if you run a SIL simulation for a model with code coverage enabled and then run a SIL simulation for another model with code coverage disabled, the software must rebuild all source files in the shared utilities folder.

BullseyeCoverage Behavior with Inline Macros

The BullseyeCoverage tool, by default, does not provide code coverage data for inline macros.

For example, if a model generates a file `slprj/ert/_sharedutils/rt_SATURATE.h` that contains the macro

```
#define rt_SATURATE(sig,ll,ul) (((sig) >= (ul)) ? (ul) : (((sig) <= (ll)) ?
(ll) : (sig)) )
```

and the macro is in `sat_ert_rtw/sat.c`, then the coverage report provides a measurement for `sat.c`, but no coverage data for the conditions within the macro `rt_SATURATE`.

To configure the BullseyeCoverage tool to provide code coverage data for inline macros:

1. Open the BullseyeCoverage Browser.
2. Select **Tools** > **Options** to open the Options dialog box.
3. On the **Build** tab, select the **Instrument macro expansions** check box.
4. Click **OK**.
5. Rerun your simulation.

Alternatively, you can add the text `-macro` to the `BullseyeCoverage` configuration file. For more information, go to www.bullseye.com/help.

Note SIL and PIL simulation support for BullseyeCoverage will be removed in a future release.

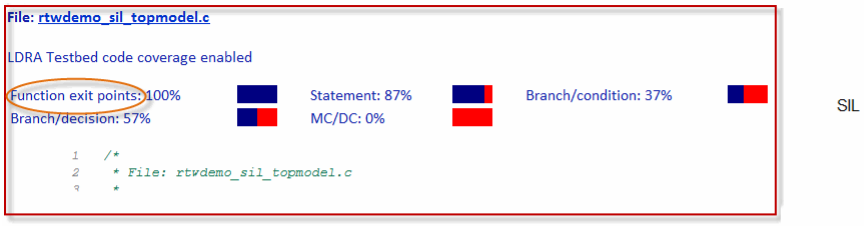
SIL and PIL Simulations with Open LDRA GUI

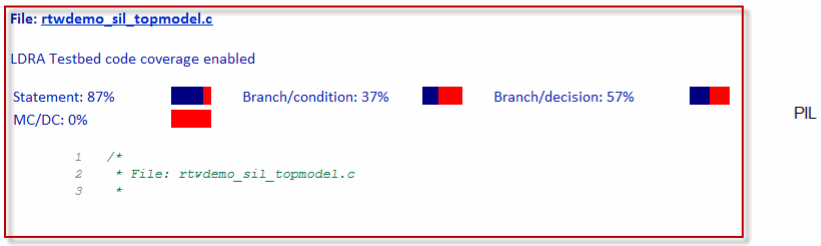
If you enable LDRA code coverage, you must verify that the LDRA tool suite GUI is not open when you run your SIL or PIL simulation. If the set name in the GUI differs from the set name used by the SIL or PIL simulation, the SIL or PIL simulation fails.

Minor SIL and PIL Differences for LDRA Tool Suite

The target connectivity API supports LDRA code coverage for top-model and Model block PIL.

There are minor differences in the code coverage information collected during SIL and PIL simulations. In particular, with PIL, the software does not explicitly show function exit point coverage. However, you can infer the coverage of function exit points by examining statement coverage.





PIL Zero Coverage LDRA Annotations

For a PIL simulation with LDRA code coverage specified, recorded coverage for some source files might be zero. In this case, the software provides summary information indicating that:

- There is coverage to measure.
- The coverage is zero.

You do not see information for individual probes on each line. The displayed summary information has an associated annotation tooltip:

```
0 out of N coverage probes were exercised (detailed breakdown unavailable)
```

Custom Header File Changes and Regeneration of LDRA Instrumentation

When you run a SIL simulation with LDRA code coverage enabled, the code generator uses LDRA software to instrument the generated code only if the code generator determines that the generated code has changed since the last generation of instrumentation code. You can change a custom header file without regenerating code. In this case, the code generator does not update the instrumentation code. The SIL simulation produces this error:

```
The call to the LDRA code coverage tool did not generate the expected report.
One possible cause of this error is that the Embedded Coder software did not
detect the change in a previously analyzed source file and skipped the static
analysis. The source file that changed might be a custom header file. For more
information, review the above output from the LDRA tool command. A static
analysis will be forced on the next simulation. You must rerun the simulation.
```

PIL Support for BullseyeCoverage

Code coverage with BullseyeCoverage is available for top-model and Model block PIL provided your PIL application can write directly to the host file system. Your target for the PIL application must provide `fopen` and `fread` access to the host file system.

If code coverage is not available when you run the PIL application on your target hardware, you might be able to collect code coverage measurements by running the PIL application on an instruction set simulator that supports direct file I/O with the host file system.

Note SIL and PIL simulation support for BullseyeCoverage will be removed in a future release.

Modify Legacy Code

If you modify legacy code and rerun a SIL or PIL simulation, the legacy code is recompiled. However, the code from the model may be up-to-date. In this case, the code generation report is not updated and does not show the modified legacy code. Instead, the code coverage information for the modified legacy code is displayed with reference to the original legacy code. You must regenerate the report. For more information, see “Generate Code Generation Report” on page 54-5.

See Also

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 91-9
- “Configure Code Coverage Programmatically” on page 91-13
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 91-11
- “Code Coverage Summary and Annotations” on page 91-15
- “Code Coverage Tool Support” on page 91-20

Collect Code Coverage Metrics with a Third-Party Tool

If you have a Simulink Coverage license, you can collect code coverage metrics during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation –– see “Collect Coverage for Code” (Simulink Coverage). This example shows how you can collect code coverage metrics with a third-party tool, that is, BullseyeCoverage or LDRA tool suite. You must install the third-party tool –– see “Code Coverage Tool Support” on page 91-20.

You measure model coverage during a normal mode simulation and then use the third-party tool in a SIL simulation to collect coverage metrics for the generated code. By using the hyperlinks in the model coverage and code coverage reports, you can compare model coverage and code coverage results. For more information about SIL simulations, see “Test Generated Code with SIL and PIL Simulations” on page 88-94.

Initial Setup

Open the model `SILTopModel`.

```
openExample('ecoder/SILPILVerificationExample', ...
             supportingFile='SILTopModel.slx')
model='SILTopModel';
```

Remove existing build folders.

```
buildFolder=RTW.getBuildDir(model);
if exist(buildFolder.BuildDirectory,'dir')
    rmdir(buildFolder.BuildDirectory,'s');
end
```

Configure generation of model coverage report.

```
set_param(model, 'RecordCoverage','on')
clear covCumulativeData
```

Set up the stimulus data by running the function `SILTopModelData`.

```
T=0.1; % sample time
[ticks_to_count, reset, counter_mode, count_enable, ...
 counter_mode_values_run1, counter_mode_values_run2, ...
 count_enable_values_run1, count_enable_values_run2] = ...
    SILTopModelData(T);
```

Run a Simulation in Normal Mode

The model is configured to collect model coverage metrics. When a simulation is complete, the model coverage report opens. Use the coverage display window to navigate from blocks in the model to the corresponding sections of the coverage report.

```
counter_mode.signals.values = counter_mode_values_run1;
count_enable.signals.values = count_enable_values_run1;
set_param(model,'SimulationMode','normal');
```

Set up Simulation Data Inspector for interactive viewing and comparison of simulation results.

```
Simulink.sdi.view;
Simulink.sdi.clear;
```

Run the simulation.

```
simout_normal_run1 = sim(model, 'ReturnWorkspaceOutputs', 'on');
```

Capture the results.

```
Simulink.sdi.createRun('Run 1 (normal mode)', 'namevalue',...
                       {'simout_normal_run1'}, {simout_normal_run1});
```

Run a Second Simulation in Normal Mode

For the first simulation, the report shows that the achieved coverage is less than 100%. Run a second simulation with different input signals that increase the level of MC/DC coverage to 100%. Note that the model coverage report is configured to show cumulative coverage across both simulation runs.

```
counter_mode.signals.values = counter_mode_values_run2;
count_enable.signals.values = count_enable_values_run2;
set_param(model,'SimulationMode','normal');

simout_normal_run2 = sim(model, 'ReturnWorkspaceOutputs', 'on');

Simulink.sdi.createRun('Run 2 (normal mode)', 'namevalue',...
                       {'simout_normal_run2'}, {simout_normal_run2});
```

Configure the Model to Measure Code Coverage

Before running a SIL simulation, check the availability of third-party tools, and configure the model to collect code coverage metrics. If a third-party tool is not available, the model uses Simulink Coverage.

```
covToolPath = '';
ldraPath = coder.coverage.LDRA.getPath;
bullseyePath = coder.coverage.BullseyeCoverage.getPath;

coverageSettings = get_param(model,'CodeCoverageSettings');
coverageSettings.TopModelCoverage='on';
if ~isempty(ldraPath)
    coverageSettings.CoverageTool='LDRAcover or LDRA tool suite';
elseif ~isempty(bullseyePath)
    coverageSettings.CoverageTool='BullseyeCoverage';
else
    coverageSettings.CoverageTool='None';
end
set_param(model,'CodeCoverageSettings',coverageSettings);
```

Run Simulations in SIL Mode

The normal mode simulations produce coverage metrics for the model. With a SIL simulation, you can apply the same input stimulus signals to the generated code and measure code coverage.

Run the first simulation in SIL mode.

```
counter_mode.signals.values = counter_mode_values_run1;
count_enable.signals.values = count_enable_values_run1;
set_param(model,'SimulationMode','software-in-the-loop');
```

```
set_param(model,'CodeExecutionProfiling','off');
set_param(model,'CodeProfilingInstrumentation','off');
simout_sil_run1 = sim(model, 'ReturnWorkspaceOutputs', 'on');
Simulink.sdi.createRun('Run 1 (SIL mode)', 'namevalue',...
                       {'simout_sil_run1'}, {simout_sil_run1});
```

Run the second simulation in SIL mode.

```
counter_mode.signals.values = counter_mode_values_run2;
count_enable.signals.values = count_enable_values_run2;
set_param(model,'SimulationMode','software-in-the-loop');
set_param(model,'CodeExecutionProfiling','off');
set_param(model,'CodeProfilingInstrumentation','off');
simout_sil_run2 = sim(model, 'ReturnWorkspaceOutputs', 'on');
Simulink.sdi.createRun('Run 2 (SIL mode)', 'namevalue',...
                       {'simout_sil_run2'}, {simout_sil_run2});
```

When a simulation is complete, click the link in the Command Window to open the code coverage report and view the cumulative code coverage results. The link is available only if you have a third-party tool installed.

Use hyperlinks in the code coverage report to go to corresponding locations in the block diagram. Then, by using the coverage display window, you can open corresponding sections of the model coverage report. Compare model coverage and code coverage results.

The Simulation Data Inspector opens automatically, allowing interactive viewing and analysis of the results. Use the Compare and Inspect panes to confirm that the SIL and normal mode logged signals are identical for both runs.

Concluding Remarks

In this example, you:

- Collected model coverage metrics during a normal mode simulation.
- Collected code coverage metrics during a SIL simulation.
- Navigated between the code coverage and model coverage reports.
- Cross-checked metrics from both reports.

See Also

Related Examples

- “Code Coverage”
- “Execution Profiling for Generated Code” on page 76-11
- “Verification, Testing, and Certification”
- https://www.mathworks.com/products/do-178.html

Embedded IDEs and Embedded Targets

Run-Time Data Interface Extensions in Simulink Coder

Customize Generated ASAP2 File

In this section...

"About ASAP2 File Customization" on page 92-2
"ASAP2 File Structure on the MATLAB Path" on page 92-2
"Customize the Contents of the ASAP2 File" on page 92-2
"ASAP2 Templates" on page 92-3
"Customize Computation Method Names" on page 92-5
"Suppress Computation Methods for FIX_AXIS" on page 92-6

Note TLC based customization will be deprecated in a future release. To customize the ASAP2 file, see "Customize Generated ASAP2 File" on page 65-2.

About ASAP2 File Customization

The Embedded Coder product provides a number of Target Language Compiler (TLC) files to enable you to customize the ASAP2 file generated from a Simulink model.

ASAP2 File Structure on the MATLAB Path

The ASAP2 related files are organized within the folders identified below:

- TLC files for generating ASAP2 file

  The *matlabroot*`/rtw/c/tlc/mw` (open) folder contains TLC files that generate ASAP2 files, `asamlib.tlc`, `asap2lib.tlc`, `asap2main.tlc`, and `asap2grouplib.tlc`. These files are included by the setting of model configuration parameter **System target file**.
- ASAP2 target files

  The *matlabroot*`/toolbox/rtw/targets/asap2/asap2` (open) folder contains the ASAP2 system target file and other control files.
- Customizable TLC files

  The *matlabroot*`/toolbox/rtw/targets/asap2/asap2/user` (open) folder contains files that you can modify to customize the content of your ASAP2 files.
- ASAP2 templates

  The *matlabroot*`/toolbox/rtw/targets/asap2/asap2/user/templates` (open) folder contains templates that define each type of `CHARACTERISTIC` in the ASAP2 file.

Customize the Contents of the ASAP2 File

The ASAP2 related TLC files enable you to customize the appearance of the ASAP2 file generated from a Simulink model. Most customization is done by modifying or adding to the files contained in the *matlabroot*`/toolbox/rtw/targets/asap2/asap2/user` (open) folder. This section refers to this folder as the `asap2/user` folder.

The user-customizable files provided are divided into two groups:

- The *static* files define the parts of the ASAP2 file that are related to the environment in which the generated code is used. They describe information specific to the user or project. The static files are not model dependent.
- The *dynamic* files define the parts of the ASAP2 file that are generated based on the structure of the source model.

The procedure for customizing the ASAP2 file is as follows:

1. Make a copy of the `asap2/user` folder before making modifications.
2. Remove the old `asap2/user` folder from the MATLAB path, or add the new `asap2/user` folder to the MATLAB path above the old folder. The MATLAB session uses the ASAP2 setup file, `asap2setup.tlc`, in the new folder.

   `asap2setup.tlc` specifies the folders and files to include in the TLC path during the ASAP2 file generation process. Modify `asap2setup.tlc` to control the folders and folders included in the TLC path.
3. Modify the static parts of the ASAP2 file. These include

   - Project and header symbols, which are specified in `asap2setup.tlc`
   - Static sections of the file, such as file header and tail, `A2ML`, `MOD_COMMON`, and so on These are specified in `asap2userlib.tlc`.
   - Specify the appearance of the dynamic contents of the ASAP2 file by modifying the existing ASAP2 templates or by defining new ASAP2 templates. Sections of the ASAP2 file affected include

     `RECORD_LAYOUT`: modify parts of the ASAP2 template files.

     `CHARACTERISTIC`: modify parts of the ASAP2 template files. For more information on modifying the appearance of `CHARACTERISTIC` records, see “ASAP2 Templates” on page 92-3.
   - `MEASUREMENT`: These are specified in `asap2userlib.tlc`.
   - `COMPU_METHOD`: These are specified in `asap2userlib.tlc`.

ASAP2 Templates

The appearance of `CHARACTERISTIC` records in the ASAP2 file is controlled using a different template for each type of `CHARACTERISTIC`. The `asap2/user` folder contains template definition files for scalars, 1-D Lookup Table blocks and 2-D Lookup Table blocks. You can modify these template definition files, or you can create additional templates as required.

The procedure for creating a new ASAP2 template is as follows:

1. Create a template definition file. See “Create Template Definition Files” on page 92-4.
2. Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, `asap2setup.tlc`.

Create Template Definition Files

This section describes the components that make up an ASAP2 template definition file. This description is in the form of code examples from `asap2lookup1d.tlc`, the template definition file for the `Lookup1D` template. This template corresponds to the `Lookup1D` parameter group.

Note When creating a new template, use the corresponding parameter group name in place of `Lookup1D` in the code shown.

Template Registration Function

The input argument is the name of the parameter group associated with this template:

```
%<LibASAP2RegisterTemplate("Lookup1D")>
```

RECORD_LAYOUT Name Definition Function

Record layout names (aliases) can be arbitrarily specified for each data type. This function is used by the other components of this file.

```
%function ASAP2UserFcnRecordLayoutAlias_Lookup1D(dtId) void
  %switch dtId
  %case tSS_UINT8
    %return "Lookup1D_UBYTE"
  ...
  %endswitch
%endfunction
```

Function to Write RECORD_LAYOUT Definitions

This function writes `RECORD_LAYOUT` definitions associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the template name after the underscore:

```
%function ASAP2UserFcnWriteRecordLayout_Lookup1D() Output
  /begin RECORD_LAYOUT
%<ASAP2UserFcnRecordLayoutAlias_Lookup1D(tSS_UINT8)>
    ...
  /end   RECORD_LAYOUT
%endfunction
```

Function to Write the CHARACTERISTIC

This function writes the `CHARACTERISTIC` associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the template name after the underscore.

The input argument to this function is a pointer to a parameter group record. The example shown is for a `Lookup1D` parameter group that has two members. The references to the associated `x` and `y` data records are obtained from the parameter group record as shown.

This function calls a number of built-in functions to obtain the required information. For example, `LibASAP2GetSymbol` returns the symbol (name) for the specified data record:

```
%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup)
Output
```

```
  %assign xParam = paramGroup.Member[0].Reference
  %assign yParam = paramGroup.Member[1].Reference
  %assign dtId = LibASAP2GetDataTypeId(xParam)
    /begin CHARACTERISTIC
     /* Name */              %<LibASAP2GetSymbol(xParam)>
     /* Long identifier */   "%<LibASAP2GetLongID(xParam)>"
     ...
    /end CHARACTERISTIC
%endfunction
```

Customize Computation Method Names

In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model oriented representation. Simulink Coder software provides the ability to customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file readability, or names that meet organizational requirements.

To customize computation method names, use the MATLAB function `getCompuMethodName`, which is defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.

The `getCompuMethodName` function constructs a computation method name. The function prototype is

```
cmName = getCompuMethodName(dataTypeName, cmUnits)
```

where *`dataTypeName`* is the name of the data type associated with the computation method, *`cmUnits`* is the units as specified in the `Unit` property of a `Simulink.Parameter` or `Simulink.Signal` object (for example, `rpm` or `m/s`), and *`cmName`* returns the constructed computation method name.

The default constructed name returned by the function has the format

```
<localPrefix><datatype>_<cmUnits>
```

where

- `<local_Prefix>` is a local prefix, `CM_`, defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.
- `<datatype>` and `<cmUnits>` are the arguments you specified to the `getCompuMethodName` function.

Additionally, in the generated ASAP2 file, the constructed name is prefixed with `<ASAP2CompuMethodName_Prefix>`, a model prefix defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc`.

For example, if you call the `getCompuMethodName` function with the *`dataTypeName`* argument `'int16'` and the *`cmUnits`* argument `'m/s'`, and generate an ASAP2 file for a model named `myModel`, the computation method name would appear in the generated file as follows:

```
/begin COMPU_METHOD
   /* Name of CompuMethod */ myModel_CM_int16_m_s
   /* Units */ "m/s"
   ...
/end COMPU_METHOD
```

Suppress Computation Methods for FIX_AXIS

Versions 1.51 and later of the ASAP2 specification state that for certain cases of lookup table axis descriptions (integer data type and no doc units), a computation method is not required and the Conversion Method parameter must be set to the value `NO_COMPU_METHOD`. You can control whether or not computation methods are suppressed when not required using the Target Language Compiler (TLC) option `ASAP2GenNoCompuMethod`. This TLC option is disabled by default. If you enable the option, ASAP2 file generation does not generate computation methods for lookup table axis descriptions when not required, and instead generates the value `NO_COMPU_METHOD`. For example:

```
/begin CHARACTERISTIC
/* Name            */
lu1d_fix_axisTable_data
...
/begin AXIS_DESCR
    ...
    /* Conversion Method    */
NO_COMPU_METHOD
    ...
/end CHARACTERISTIC
```

The `ASAP2GenNoCompuMethod` option is defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc`.

See Also

Related Examples

- “How Generated Code Exchanges Data with an Environment”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Build Process Integration in Simulink Coder

- “Control Build Process Compiling and Linking” on page 93-2
- “Control Library Location and Naming During Build” on page 93-3
- “Customize Post-Code-Generation Build Processing” on page 93-7
- “Configure Generated Code with TLC” on page 93-12
- “Use makecfg to Customize Generated Makefiles for S-Functions” on page 93-15
- “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17
- “Register Custom Toolchain and Build Executable” on page 93-21
- “Customize Build Process with STF_make_rtw_hook File” on page 93-25
- “Customize Build Process with sl_customization.m” on page 93-29
- “Replace STF_rtw_info_hook Supplied Target Data” on page 93-33

Control Build Process Compiling and Linking

After generating code for a model, the build process determines whether to compile and link an executable program. Various factors guide this determination:

- **Generate code only** option

  When you select this option, the code generator produces code for the model, including a makefile.

- **Generate makefile** option

  When you clear this option, the code generator does not produce a makefile for the model. You must specify post code generation processing, including compilation and linking, as a user-defined command, as explained in “Customize Post-Code-Generation Build Processing” on page 93-7.

- Makefile-only target

  The Microsoft Visual C++ Project Makefile versions of the `grt` and Embedded Coder target configurations generate a Visual C++ project makefile (*`model`*`.mak`). To build an executable, you must open *`model`*`.mak` in the Visual C++ IDE and compile and link the model code.

- `HOST` template makefile variable

  The template makefile variable `HOST` identifies the type of system upon which your executable is intended to run. The variable can be set to one of three possible values: `PC`, `UNIX`, or `ANY`.

  By default, `HOST` is set to `UNIX` in template makefiles designed for use with The Open Group UNIX platforms (such as `grt_unix.tmf`), and to `PC` in the template makefiles designed for use with development systems for the PC (such as `grt_vcx64.tmf`).

  If the Simulink software is running on the same type of system as the system specified by the `HOST` variable, then the executable is built. Otherwise,

  - If `HOST = ANY`, an executable is still built. This option is useful when you want to cross-compile a program for a system other than the one that the Simulink software is running on.
  - Otherwise, processing stops after generating the model code and the makefile; the following message is displayed on the MATLAB command line.

    ```
    ### Make will not be invoked - template makefile is for a different host
    ```

Control Library Location and Naming During Build

When you generate precompiled, non-precompiled, and model reference libraries, you can control the library location and library name by using configuration parameters. These parameters control values in generated makefiles during model builds:

- For build processes that use the toolchain approach, control the generated library location by using the `TargetPreCompLibLocation` configuration parameter.
- For build processes that use the template makefile approach, control the generated library location by using the `TargetPreCompLibLocation` configuration parameter and control the generated library name by using the `TargetLibSuffix` configuration parameter.

Library Control Parameters

Use the library control parameters to:

- Specify the location of precompiled libraries, such as blockset libraries or the Simulink Coder block library. Typically, a target has cross-compiled versions of these libraries and places them in a target-specific folder.
- Control the suffix applied to library file names (for example, `_target.a` or `_target.lib`).

Targets can set the parameters inside the system target file (STF) select callback. For example:

```
function mytarget_select_callback_handler(varargin)
    hDig=varargin{1};
    hSrc=varargin{2};
    slConfigUISetVal(hDig, hSrc, 'TargetPreCompLibLocation',...
    'c:\mytarget\precomplibs');
    slConfigUISetVal(hDig, hSrc, 'TargetLibSuffix',...
    '_target.library');
```

The TMF has corresponding expansion tokens:

```
    |>EXPAND_LIBRARY_LOCATION<|
    |>EXPAND_LIBRARY_SUFFIX<|
```

Alternatively, you can use a call to the `set_param` function. For example:

```
set_param(model,'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

Note If your model contains referenced models, you can use the make option `USE_MDLREF_LIBPATHS` to control whether libraries used by the referenced models are copied to the build folder of the parent model. For more information, see “Location of Model Reference Libraries” on page 93-5.

Identify Library File Type for Toolchain Approach

The toolchain approach for model builds does not use the value of the `TargetLibSuffix` configuration parameter to select the library file name *`suffix`* and *`extension`*.

With the toolchain approach, the final binary name is composed of the *`modelname`*, the *`compilername`*, and the *`extension`* provided by the build tool description in the toolchain definition:

```
model_compilername.extension
```

You can identify the static library file name extension from the build tool description in the toolchain definition. To get this information for the default toolchain, use this procedure:

1. Get the default toolchain name. For example, enter:

   ```
   tc_name = coder.make.getDefaultToolchain()
   ```
2. Get the default toolchain handle. For example, enter:

   ```
   tc = coder.make.getToolchainInfoFromRegistry(tc_name)
   ```
3. Get the handle to the toolchain object. For example, enter:

   ```
   tool_archiver = tc.getBuildTool('Archiver');
   ```
4. Get the extension. For example, enter:

   ```
   ext_archiver = tool_archiver.getFileExtension('Static Library');
   ```

Note If you do not set the `TargetLibSuffix` parameter, template makefile and toolchain approaches produce the same static library file name *`extension`*.

Specify the Location of Precompiled Libraries

Use the `TargetPreCompLibLocation` configuration parameter to:

- Override the precompiled library location specified in the `rtwmakecfg.m` file (see "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 93-17 for details)
- Precompile and distribute target-specific versions of product libraries (for example, the DSP System Toolbox product)

For a precompiled library, such as a blockset library or the Simulink Coder block library, the location specified in `rtwmakecfg.m` is typically a location specific to the blockset or the Simulink Coder product. The code generator expects that the library exists in this location and links against the library during builds.

For some applications, such as custom targets, it is preferable to locate the precompiled libraries in a target-specific or other alternate location rather than in the location specified in `rtwmakecfg.m`. For a custom target, the code generator expects that the target-specific cross-compiler creates the library, and you place the library in the target-specific location. Compile and place libraries supported by the target in the target-specific location, so they can be used during the build process.

You can set up the `TargetPreCompLibLocation` parameter in its select callback. The path that you specify for the parameter must be a fully qualified, absolute path to the library location. Relative paths are not supported. For example:

```
slConfigUISetVal(hDlg, hSrc, 'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

Alternatively, you set the parameter with a call to the `set_param` function. For example:

```
set_param(model,'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

During makefile generation, the build process replaces the tokens with the location from the `rtwmakecfg.m` file. For example, if the library name in the `rtwmakecfg.m` file is `'rtwlib'`, the template makefile build approach expands the token from:

```
LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|\
_target.library
```

to:

```
LIBS += c:\mytarget\precomplibs\rtwlib_target.library
```

By default, `TargetPreCompLibLocation` is an empty character vector. The build process uses the location in `rtwmakecfg.m` for the token replacement.

Location of Model Reference Libraries

The build process assigns a path such as `../slprj/grt/sub/sub_rtwlib.lib` to `MODELREF_LINK_LIBS`. The build process uses the path to gain access to the library file at link time.

Control the Suffix Applied to Library File Names

With the template makefile approach for model builds, use the `TargetLibSuffix` configuration parameter to control the suffix applied to library names (for example, `_target.lib` or `_target.a`). The specified suffix scheme must include a period (.). You can apply `TargetLibSuffix` to the following libraries:

- Libraries on which a target depends, as specified in the `rtwmakecfg.m` API. You can use `TargetLibSuffix` to change the suffix of both precompiled and non-precompiled libraries configured from the `rtwmakecfg` API. For details, see “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 93-17.

  In this case, a target can set the parameter in its select callback. For example:

  ```
  slConfigUISetVal(hDlg, hSrc, 'TargetLibSuffix',...
  '_target.library');
  ```

  Alternatively, you can use a call to the `set_param` function. For example:

  ```
  set_param(model,'TargetLibSuffix','_target.library');
  ```

  During the TMF-to-makefile conversion, the build process replaces the token `|>EXPAND_LIBRARY_SUFFIX<|` with the specified suffix. For example, if the library name specified in the `rtwmakecfg.m` file is `'rtwlib'`, the TMF expands from:

  ```
  LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|\
  |>EXPAND_LIBRARY_SUFFIX<|
  ```

  to:

  ```
  LIBS += c:\mytarget\precomplibs\rtwlib_target.library
  ```

  By default, `TargetLibSuffix` is set to an empty character vector. In this case, the build process replaces the token `|>EXPAND_LIBRARY_SUFFIX<|` with an empty character vector.

- Model libraries created with model reference. For these cases, associated makefile variables do not require the `|>EXPAND_LIBRARY_SUFFIX<|` token. Instead, the build process includes `TargetLibSuffix` implicitly. For example, for a top model named `topmodel` with referenced models named `refmodel1` and `refmodel2`, the TMF of the top model is expanded from:

  ```
  MODELLIB           = |>MODELLIB<|
  MODELREF_LINK_LIBS = |>MODELREF_LINK_LIBS<|
  ```

  to:

```
MODELLIB           = topmodellib_target.library
MODELREF_LINK_LIBS = \
refmodel1_rtwlib_target.library refmodel2_rtwlib_target.library
```

By default, the `TargetLibSuffix` parameter is an empty character vector. In this case, the build process chooses a default suffix for these three tokens using a file extension of `.lib` on Windows hosts and `.a` on UNIX hosts. For model reference libraries, the default suffix also includes `_rtwlib`. For example, on a Windows host, the expanded makefile values are:

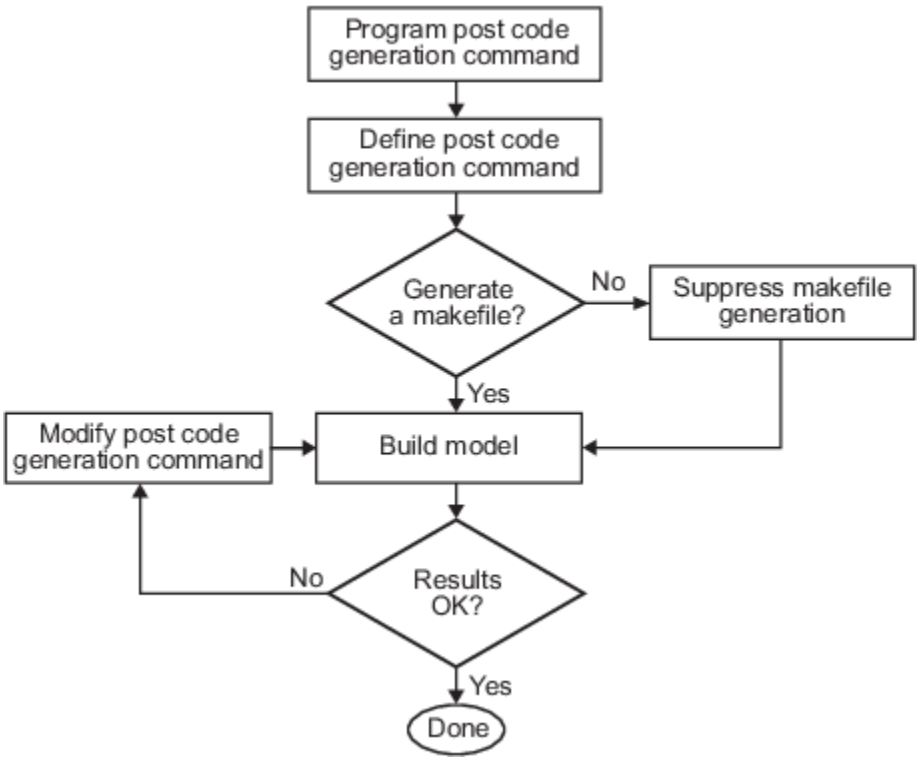
```
MODELLIB           = topmodellib.lib
MODELREF_LINK_LIBS = refmodel1_rtwlib.lib refmodel2_rtwlib.lib
```

Customize Post-Code-Generation Build Processing

The code generator provides a set of tools, including a build information object, you can use to customize build processing that occurs after code generation. You can use such customizations for target development or the integration of third-party tools into your application development environment.

Workflow for Setting Up Customizations

The following figure and the steps that follow show the general workflow for setting up post-code-generation customizations.



1. Program the post code generation command on page 93-8.
2. Define the post code generation command on page 93-9.
3. Suppress makefile generation on page 93-11, if applicable.
4. Build the model.
5. Modify the command and rebuild the model until the build results are acceptable.

Build Information Object

At the start of a model build, the build process logs the following build option and dependency information to a temporary build information object:

- Compiler options

- Preprocessor identifier definitions
- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

You can retrieve information from and add information to this object by using an extensive set of functions. For a list of available functions and detailed function descriptions, see “Code Compilation Customization”. “Program a Post Code Generation Command” on page 93-8 explains how to use the functions to control post code generation build processing.

Program a Post Code Generation Command

For certain applications, you could want to control aspects of the build process after the code generation. For example, you can use this approach if you develop your own target, or you want to apply an analysis tool to the generated code before continuing with the build process. You can apply this level of control to the build process by programming and then defining a post code generation command.

A post code generation command is a MATLAB language file that typically calls functions that get data from or add data to the build information object of the model. You can program the command as a script or function.

If you program the command as a:	Then the:
Script	Script can gain access to the model name and the build information directly
Function	Function can pass the model name and the build information as arguments

If your post code generation command calls user-defined functions, make sure that the functions are on the MATLAB path. If the build process cannot find a function you use in your command, the build process errors out.

You can then call a combination of build information functions, as listed in “Code Compilation Customization”, to customize the post code generation build processing of the model.

The following example shows a fragment of a post code generation command that gets the file names and paths of the source and include files generated for a model for analysis.

```
function analyzegencode(buildInfo)
% Get the names and paths of source and include files
% generated for the model and then analyze them.

% buildInfo - build information for my model.

% Define cell array to hold data.
MyBuildInfo={};

% Get source file information.
MyBuildInfo.srcfiles=getSourceFiles(buildInfo, true, true);
MyBuildInfo.srcpaths=getSourcePaths(buildInfo, true);

% Get include (header) file information.
MyBuildInfo.incfiles=getIncludeFiles(buildInfo, true, true);
```

```
MyBuildInfo.incpaths=getIncludePaths(buildInfo, true);

% Analyze generated code.
.
.
.
```

Define a Post Code Generation Command

After you program a post code generation command, inform the build process that the command exists and to add it to the build processing of the model. Define the command with the `PostCodeGenCommand` model configuration parameter. When you define a post code generation command, the build process evaluates the command after generating and writing the generated code to disk and before generating a makefile.

As the following syntax lines show, the arguments that you specify when setting the configuration parameter varies depending on whether you program the command as a script, function, or set of functions.

Note When defining the command as a function, you can specify an arbitrary number of input arguments. To pass the name and build information of the model to the function, specify identifiers `modelName` and `buildInfo` as arguments.

Script

```
set_param(model, 'PostCodeGenCommand',...
'pcgScriptName');
```

Function

```
set_param(model, 'PostCodeGenCommand',...
 'pcgFunctionName(modelName)');
```

Multiple Functions

```
pcgFunctions=...
'pcgFunction1Name(modelName);...
pcgFunction2Name(buildInfo)';
set_param(model, 'PostCodeGenCommand',...
 pcgFunctions);
```

The following call to `set_param` defines `PostCodGenCommand` to evaluate the function `analyzegencode`.

```
set_param(model, 'PostCodeGenCommand',...
'analyzegencode(buildInfo)');
```

Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment

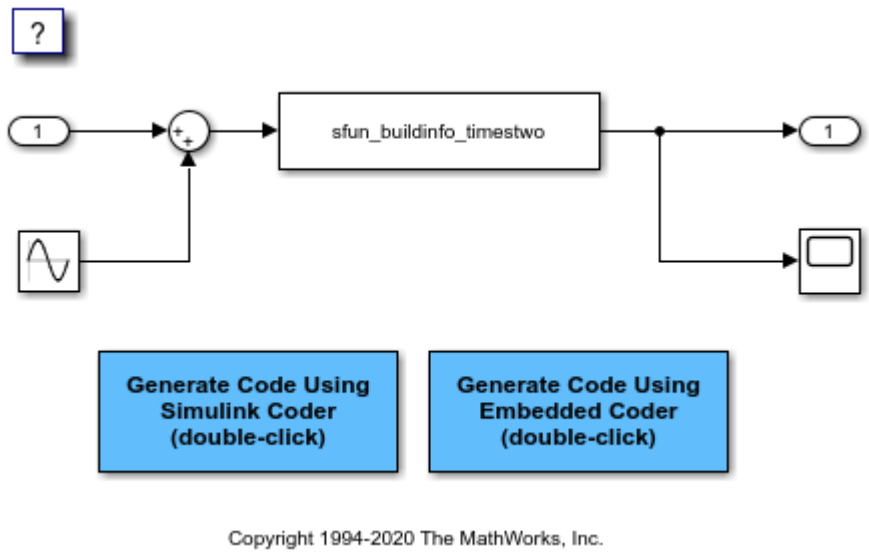
This example shows how to use the build information programming interface and the model configuration parameter **Post code generation command** (`PostCodeGenCommand`) to customize the build process. For this example, **Post code generation command** is set to `rtwdemo_buildinfo_data`. The build process invokes that function after code generation.

The example also shows how to use build configuration function `rtwmakecfg.m`.

Open Example Model

Open the example model `rtwdemo_buildinfo`.

```
open_system('rtwdemo_buildinfo');
```



Generate Code from Model

Generate code. The code generator produces the file `BuildInfo.html`, which documents the build information object.

Examine the Build Process Customizations and Output

Examine the build process customizations and the post code generation query of the build information object. In a Web browser, open the file `BuildInfo.html`. The file provides hyperlinks that you can use to examine the code produced for the model. Alternatively, in the MATLAB Command Window:

- To view the function configured to execute during the post code generation stage of the build process, enter:

```
get_param('rtwdemo_buildinfo','PostCodeGenCommand');
```

- To study the programming interface for the build information object, enter:

```
rtwdemo_buildinfo_data(buildInfo);
```

- To study how the example uses the `rtwmakecfg` function, enter:

```
edit rtwmakecfg.m;
```

- To view the build information object data that the build process saves in the `buildInfo.mat` file, open the file `rtwdemo_<target>_rtw\buildInfo.mat`.

- To view the contents of the post code generation function `rtwdemo_buildinfo_data.m`, enter:

```
edit rtwdemo_buildinfo_data.m;
```

At the end of the file, the function calls `packNGo` to package the source and object files that the build information object identifies for relocation.

Related Topics

- "Code Compilation Customization"
- "Customize Post-Code-Generation Build Processing"
- "Use rtwmakecfg.m API to Customize Generated Makefiles"
- "Relocate or Share Generated Code"

Suppress Makefile Generation

The code generator lets you suppress makefile generation during the build process. For example, you can use this support when you integrate tools into the build process that do not use makefiles.

To instruct the code generator not to produce a makefile, do one of the following:

- Clear the **Generate makefile** option on the **Code Generation** pane of the Configuration Parameters dialog box.
- Set the value of the configuration parameter `GenerateMakefile` to `off`.

When you suppress makefile generation,

- You cannot explicitly specify a make command or template makefile.
- Specify your own instructions for a post code generation processing, including compilation and linking, in a post code generation command as explained in "Program a Post Code Generation Command" on page 93-8 and "Define a Post Code Generation Command" on page 93-9.

See Also

`RTW.BuildInfo`

Configure Generated Code with TLC

You can use the Target Language Compiler (TLC) to fine-tune your generated code. TLC supports extended code generation variables and options in addition to parameters available on the **Code Generation** pane on the Configuration Parameters dialog box. There are two ways to set TLC variables and options, as described in this section.

Note Do not customize TLC files in the folder *matlabroot*/rtw/c/tlc even though the capability exists to do so. It is possible that such TLC customizations are not applied during the code generation process. Such customizations can lead to unpredictable results.

Assigning Target Language Compiler Variables

The `%assign` statement lets you assign a value to a TLC variable, as in:

```
%assign MaxStackSize = 4096
```

This assignment is also known as creating a *parameter name/parameter value pair*.

For a description of the `%assign` statement, see “Target Language Compiler Directives”. Write your `%assign` statements in the `Configure RTW code generation settings` section of the system target file.

The following table lists the code generation variables you can set with the `%assign` statement.

Target Language Compiler Optional Variables

Variable	Description
`MaxStackSize=N`	When the **Enable local block outputs** check box is selected, the total allocation size of local variables that are declared by block outputs in the model cannot exceed `MaxStackSize` (in bytes). `MaxStackSize` can be a positive integer. If the total size of local block output variables exceeds this maximum, the remaining block output variables are allocated in global, rather than local, memory. The default value for `MaxStackSize` is `rtInf`, that is, unlimited stack size. **Note:** Local variables in the generated code from sources other than local block outputs, such as from a Stateflow diagram or MATLAB Function block, and stack usage from sources such as function calls and context switching are not included in the `MaxStackSize` calculation. For overall executable stack usage metrics, do a target-specific measurement by using run-time (empirical) analysis or static (code path) analysis with object code.
`MaxStackVariableSize=N`	When the **Enable local block outputs** check box is selected, this selection limits the size of a local block output variable declared in the code to `N` bytes, where `N>0`. A variable whose size exceeds `MaxStackVariableSize` is allocated in global, rather than local, memory. The default is 4096.

`RemoveFixptWordSizeChecks=value`	For a model that uses fixed-point data types, control the generation of preprocessor directives that perform fixed-point word size checks. If *`value`* is:
	
	• `0`, generate the preprocessor directives. The generated code contains fixed-point word size checks.
	• `1`, suppress the generation of the preprocessor directives. The generated code does not contain fixed-point word size checks.
	
	You can also suppress the generation of the preprocessor directives by running this command:
	
	`set_param(gcs, 'TLCOptions', ...`
	`'-aRemoveFixptWordSizeChecks=1')`
	
	If you generate code that does not contain the fixed-point word size checks, you can use processor-in-the-loop (PIL) simulations that check **Hardware Implementation** settings with reference to the target hardware. For more information, see “Verification of Code Generation Assumptions” on page 88-71.

Variable	Description
`WarnNonSaturatedBlocks=value`	Flag to control display of overflow warnings for blocks that have saturation capability, but have it turned off (not selected) in their dialog box. Options include: • `0` — Warning is not displayed. • `1` — Displays one warning for the model during code generation • `2` — Displays one warning that contains a list of offending blocks

Set Target Language Compiler Options

You can specify TLC command-line options for code generation using the model parameter `TLCOptions` in a `set_param` function call. For information about these options, see “Specify TLC for Code Generation” and “Configure TLC”.

See Also

Related Examples

- “Target Language Compiler Directives”

Use makecfg to Customize Generated Makefiles for S-Functions

With the toolchain or template makefile approach for building code, you can customize generated makefiles for S-functions. Through the customization, you can specify additional items for the S-function build process:

- Source files and folders
- Include files and folders
- Library names
- Preprocessor macro definitions
- Compiler flags
- Link objects

To customize the generated makefile:

1. In the S-function MEX file folder that contains your S-function components, create a `makecfg.m` or *`specificSFunction`*`_makecfg.m` file. If you create a `makecfg.m` file, it is active for any S-function in the S-function MEX file folder that the model uses. A *`specificSFunction`*`_makecfg.m` file is active only if the model uses the corresponding S-function.
2. In the file that you create, use `RTW.BuildInfo` functions to specify additional items for the S-function build process. For example, you can use:
   - `addCompileFlags` to specify compiler options.
   - `addDefines` to specify preprocessor macro definitions.
3. Save the created file.

After code generation, in the S-function MEX file folder, the code generator searches for `makecfg.m` and *`specificSFunction`*`_makecfg.m` files. If the files are present, the code generator uses these files to customize the generated makefile, *`model`*`.mk`.

For example, consider a folder that contains `signalConvert.mexa64` (S-function binary file) and `signalConvert.tlc` (inlined S-function implementation) after the TLC phase of the build process. The S-function requires an additional source code file, `filterV1.c`, which is located in `anotherFolder`. You can create a file, `signalConvert_makecfg.m`, that uses `RTW.BuildInfo` functions to specify `filterV1.c` for the build process.

```
function signalConvert_makecfg(objBuildInfo)

absolutePath = fullfile('$(START_DIR)', 'anotherFolder');
addSourceFiles(objBuildInfo, 'filterV1.c', absolutePath);
```

See Also

Related Examples

- “Build Process Workflow for Real-Time Systems”
- “Approaches for Building Code Generated from Simulink Models”
- “Import Calls to External Code into Generated Code with Legacy Code Tool”

- “Use rtwmakecfg.m API to Customize Generated Makefiles”

Use rtwmakecfg.m API to Customize Generated Makefiles

Note To customize generated makefiles for S-functions, MathWorks recommends the use of the `makecfg` approach instead of the `rtwmakecfg` function. For more information, see “Use makecfg to Customize Generated Makefiles for S-Functions” on page 93-15.

Both the toolchain approach and the template makefile approach for builds let you add the following items to generated makefiles:

- Source folders
- Include folders
- Library names
- Module objects

About the rtwmakecfg Function

Using an `rtwmakecfg` function, you add this information to the makefile during the build operation for S-functions. The `rtwmakecfg` function is useful when specifying added sources and libraries to build a model that contains one or more of your S-function blocks.

To add information pertaining to an S-function to the makefile:

1. Create the MATLAB language `rtwmakecfg` function in the `rtwmakecfg.m` file. The code generator associates this file with your S-function based on its folder location. “Create the rtwmakecfg Function” on page 93-17 describes the requirements for the `rtwmakecfg` function and the data it returns.
2. If you are using the template makefile approach, modify the TMF of your target such that it supports macro expansion for the information that the `rtwmakecfg` function returns. “Modify the Template Makefile for rtwmakecfg” on page 93-19 describes the required modifications. If you are using the toolchain approach, the information that the rtwmakecfg function returns is used by the generated makefile; no further configuration is required.

After the TLC phase of the build process, when generating a makefile, the code generator searches for an `rtwmakecfg.m` file in the folder that contains the S-function MEX file. If it finds the file, the build process calls the `rtwmakecfg` function.

Create the rtwmakecfg Function

Create the `rtwmakecfg.m` file containing the `rtwmakecfg` function in the same folder as your S-function component (a MEX-file with a platform-dependent extension, such as *`.mexext`* on Microsoft Windows systems). The function must return a structured array that contains these fields.

Field	Description
`makeInfo.includePath`	A cell array that specifies additional include folder names, organized as a row vector. The build process expands the folder names into include instructions in the generated makefile.

Field	Description
`makeInfo.sourcePath`	A cell array that specifies additional source folder names, organized as a row vector. Include the folder names of files entered into the **S-function modules** field on the S-Function Block Parameters dialog box or into the `SFunctionModules` parameter of the block if they are not in the same folder as the S-function. The build process expands the folder names into make rules in the generated makefile.
`makeInfo.sources`	A cell array that specifies additional source file names (C or C++), organized as a row vector. Do not include the name of the S-function or files entered into the **S-function modules** field on the S-Function Block Parameters dialog box or into the `SFunctionModules` parameter of the block. The build process expands the file names into make variables that contain the source files. Specify only file names (with extension). Specify path information with the `sourcePath` field.
`makeInfo.linkLibsObjs`	A cell array that specifies additional, fully qualified paths to object or library files against which the generated code links. The build process does not compile the specified objects and libraries. However, it includes them when linking the final executable. This inclusion can be useful for incorporating libraries that you do not want the build process to recompile or for which the source files are not available. You can also use this element to integrate source files from languages other than C and C++. This integration is possible if you first create a C compatible object file or library outside of the build process.
`makeInfo.precompile`	A Boolean flag that indicates whether the libraries specified in the `rtwmakecfg.m` file exist in a specified location (`precompile==1`) or if you must create the libraries in the build folder during the build process (`precompile==0`).
`makeInfo.library`	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The build process expands the information into make rules in the generated makefile. For a list of the library fields, see the next table.

The `makeInfo.library` field consists of the following elements.

Element	Description
`makeInfo.library(n).Name`	A character array that specifies the name of the library (without an extension).
`makeInfo.library(n).Location`	A character array that specifies the folder in which the library is located when precompiled. For more information, see the description of `makeInfo.precompile` in the preceding table. A target can use the `TargetPreCompLibLocation` parameter to override this value. See “Specify the Location of Precompiled Libraries”.
`makeInfo.library(n).Modules`	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the object extension.

Note The `makeInfo.library` field must fully specify each library and how to build it. The modules list in the `makeInfo.library(n).Modules` element cannot be empty. To specify a link-only library, use the `makeInfo.linkLibsObjs` field instead.

Example:

```
disp(['Running rtwmakecfg from folder: ',pwd]);
   makeInfo.includePath = { fullfile(pwd, 'somedir2') };
   makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
   makeInfo.sources  = { 'src1.c', 'src2.cpp'};
   makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                             fullfile(pwd, 'somedir4', 'mylib.library')};
   makeInfo.precompile = 1;
   makeInfo.library(1).Name     = 'myprecompiledlib';
   makeInfo.library(1).Location = fullfile(pwd,'somdir2','lib');
   makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };
```

Note If a path that you specify in the `rtwmakecfg.m` API contains spaces, the build process does not convert the path to its nonspace equivalent. If the build environments you intend to support do not support spaces in paths, refer to “Build Process Support for File and Folder Names”.

Modify the Template Makefile for rtwmakecfg

To expand the information that an `rtwmakecfg` function generates, modify the following sections in the TMF of your target:

- `Include Path`
- `C Flags` and/or `Additional Libraries`
- `Rules`

It is possible that these TMF code examples do not apply to your make utility. For additional examples, see the GRT or ERT TMFs located in *`matlabroot`*`/toolbox/coder/compile/tmf`.

Add Folder Names to the Makefile Include Path

The following TMF code example adds folder names to the include path in the generated makefile:

```
ADD_INCLUDES = \
|>START_EXPAND_INCLUDES<|   -I|>EXPAND_DIR_NAME<| \
|>END_EXPAND_INCLUDES<|
```

Also, the `ADD_INCLUDES` macro must be added to the `INCLUDES` line.

```
INCLUDES = -I. -I.. $(ADD_INCLUDES) $(USER_INCLUDES)
```

Add Library Names to the Makefile

The following TMF code example adds library names to the generated makefile.

```
LIBS =
|>START_PRECOMP_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_PRECOMP_LIBRARIES<|
|>START_EXPAND_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_EXPAND_LIBRARIES<|
```

For more information, see “Control Library Location and Naming During Build” on page 93-3.

Add Rules to the Makefile

The TMF code example adds rules to the generated makefile.

```
|>START_EXPAND_RULES<|
$(BLD)/%.o: |>EXPAND_DIR_NAME<|/%.c $(SRC)/$(MAKEFILE) rtw_proj.tmw
    @$(BLANK)
    @echo ### "|>EXPAND_DIR_NAME<|\$*.c"
    $(CC) $(CFLAGS) $(APP_CFLAGS) -o $(BLD)$(DIRCHAR)$*.o \
    |>EXPAND_DIR_NAME<|$(DIRCHAR)$*.c > $(BLD)$(DIRCHAR)$*.lst
|>END_EXPAND_RULES<|

|>START_EXPAND_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_EXPAND_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_PRECOMP_LIBRARIES<|
```

Register Custom Toolchain and Build Executable

This example shows how to register and use a toolchain to build an executable file. This example uses the Intel® compiler. However, the concepts and programming interface apply for other toolchains. Once you register a toolchain, you can configure a model such that the code generator uses the toolchain to build an executable file for the model.

Toolchain

A toolchain is a collection of tools required to compile and link code for a specified platform. A toolchain consists of multiple tools, such as a compiler, linker, and archiver. You can configure the tools in a toolchain with multiple options and group tool specifications into types of configurations.

Create a New Folder and Copy Relevant Files

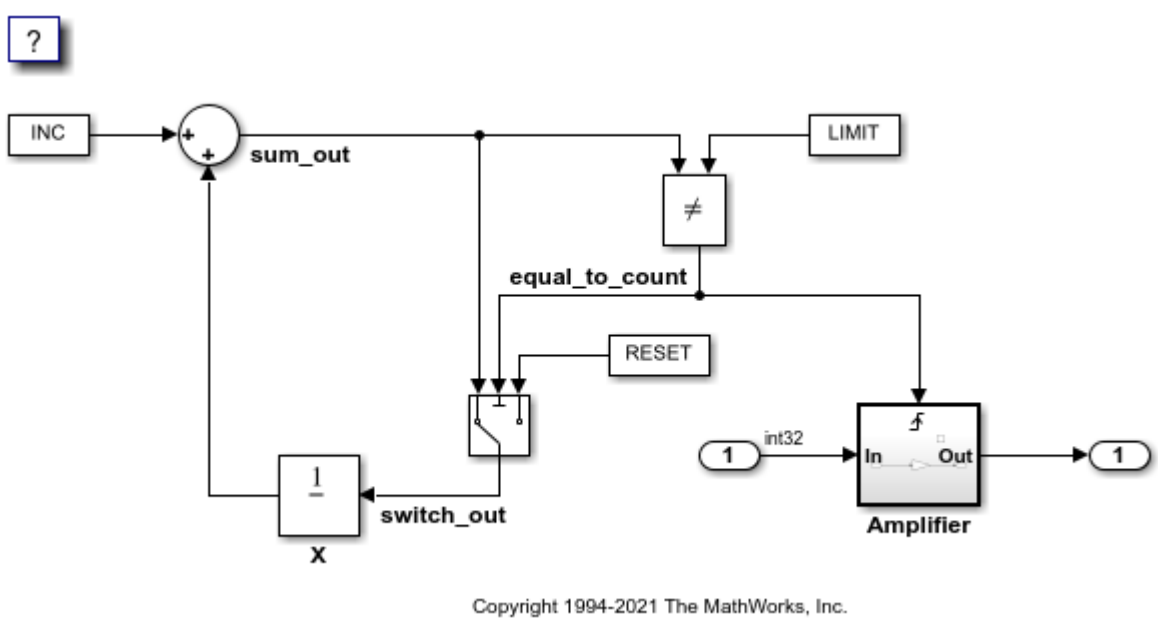
The following code creates a folder in your current working folder (pwd). The new folder contains files for this example. If you do not want to affect the current folder, or cannot generate files in this folder, change your working folder before invoking the command.

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','toolchain_demo'),'toolchain_demo','f')
cd toolchain_demo
```

Open a Model

Open the `rtwdemo_counter` model. By default, this model is configured to use the Generic Real-Time Target (GRT) configuration and the toolchain approach for building an executable file. The capability demonstrated in this example is available for models configured to use the toolchain approach.

```
model = 'rtwdemo_counter';
open_system(model)
```



Choose a Toolchain

Open the **Configuration Parameters** dialog box. Select **Code Generation**. The **Toolchain settings** section contains parameters for configuring a toolchain. From the **Toolchain** drop-down list, select the toolchain installed on your development system for building an executable file from the code generated from your model.

By default, the **Faster Runs** build configuration is selected. Click **Show settings** to see the toolchain flags specified for building the generated code. Choose a build configuration based on your current application development goal.

Close the **Configuration Parameters** dialog box.

Create a ToolchainInfo Object

This example shows how you can register a custom toolchain and add it as a selectable **Toolchain** option.

The first step to registering a custom toolchain is to create a `ToolchainInfo` object that contains information about the toolchain. Methods are available to set toolchain specifications. You can share a `ToolchainInfo` object across installations.

Open the toolchain definition file for an Intel compiler. This file creates a `ToolchainInfo` object that contains information about the Intel toolchain on a 64-bit Windows® platform.

```
edit intel_tc
```

You can display the errors and warnings in the diagnostics messages pane from builds generated by custom compilers. For more information, see “Diagnostic Message Pane”.

Run the toolchain definition file to generate the `ToolchainInfo` object.

```
tc = intel_tc;
```

Save the `ToolchainInfo` object `tc` to a MAT file.

```
save intel_tc tc
```

Register the Custom Toolchain

After you create the `ToolchainInfo` object for the new toolchain, register it. Register a toolchain in `RTW.TargetRegistry`. To register the toolchain, write an `rtwTargetInfo.m` file. Then, add that file to the MATLAB path so the system loads it automatically.

```
edit rtwTargetInfo
```

Reset the `TargetRegistry` to use the new `rtwTargetInfo.m` file.

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

Choose the Custom Toolchain

Reopen the **Configuration Parameters** dialog box. Click **Toolchain**. You should see the new toolchain in the list. Select the `Intel v18` toolchain.

Programmatically, you can accomplish the same task with the following commands:

```
cs = getActiveConfigSet(model);
set_param(cs, 'Toolchain', tc.Name)
```

Verify your selection.

```
toolchain = get_param(cs, 'Toolchain')
```

```
toolchain =

    'Intel v18 | nmake makefile (64-bit Windows)'
```

Build the Model Using the Custom Toolchain

You can now build the model with the new custom toolchain.

Note: If you do not have the Intel toolchain installed, you can use the following command to generate the code and makefile only.

```
set_param(cs, 'GenCodeOnly', 'on')
```

Build the model to generate the code and makefile that uses the new toolchain.

```
slbuild(model)
```

```
### Starting build procedure for: rtwdemo_counter
### Successful completion of code generation for: rtwdemo_counter

Build Summary

Top model targets built:

Model            Action          Rebuild Reason
========================================================================================
rtwdemo_counter  Code generated  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 16.819s
```

Obtain the build directory information.

```
dirInfo = RTW.getBuildDir(model);
```

Examine the generated makefile.

```
edit(fullfile(dirInfo.BuildDirectory, [model '.mk']))
```

If Intel compilers are installed, then when the build process is complete, you can run the executable file.

```
if ispc
   system([model '.exe'])
else
   system(model)
end
```

Restore

You can optionally remove the folder you created earlier.

```
cd ..
rmdir('toolchain_demo', 's')
```

Reset the `TargetRegistry` to remove the toolchain that you registered above.

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

Close the model.

```
close_system(model, 0)
```

Clear the variables introduced in the workspace.

```
clear INC K LIMIT RESET model tc cs toolchain
```

Customize Build Process with STF_make_rtw_hook File

The build process lets you supply optional custom code in hook methods that are executed at specified points in the code generation and make process. You can use hook methods to add target-specific actions to the build process.

STF_make_rtw_hook File

You can modify hook methods in a file generically referred to as *STF*`_make_rtw_hook.m`, where *STF* is the name of a system target file, such as `ert` or `mytarget`. This file implements a function, *STF*`_make_rtw_hook`, that dispatches to a specific action, depending on the `hookMethod` argument passed in.

The build process calls *STF*`_make_rtw_hook`, passing in the `hookMethod` argument and other arguments. You implement only those hook methods that your build process requires.

If your model contains reference models, you can implement an *STF*`_make_rtw_hook.m` for each reference model as required. The build process calls each *STF*`_make_rtw_hook` for reference models, processing these files recursively (in dependency order).

Conventions for Using the STF_make_rtw_hook File

For the build process to call the *STF*`_make_rtw_hook`, check that the following conditions are met:

- The *STF*`_make_rtw_hook.m` file is on the MATLAB path.
- The file name is the name of your system target file (`STF`), appended to the text `_make_rtw_hook.m`. For example, if you generate code with a custom system target file `mytarget.tlc`, name your hook file `mytarget_make_rtw_hook.m`, and name the hook function implemented within the file `mytarget_make_rtw_hook`.
- The hook function implemented in the file uses the function prototype described in “STF_make_rtw_hook.m Function Prototype and Arguments” on page 93-25.

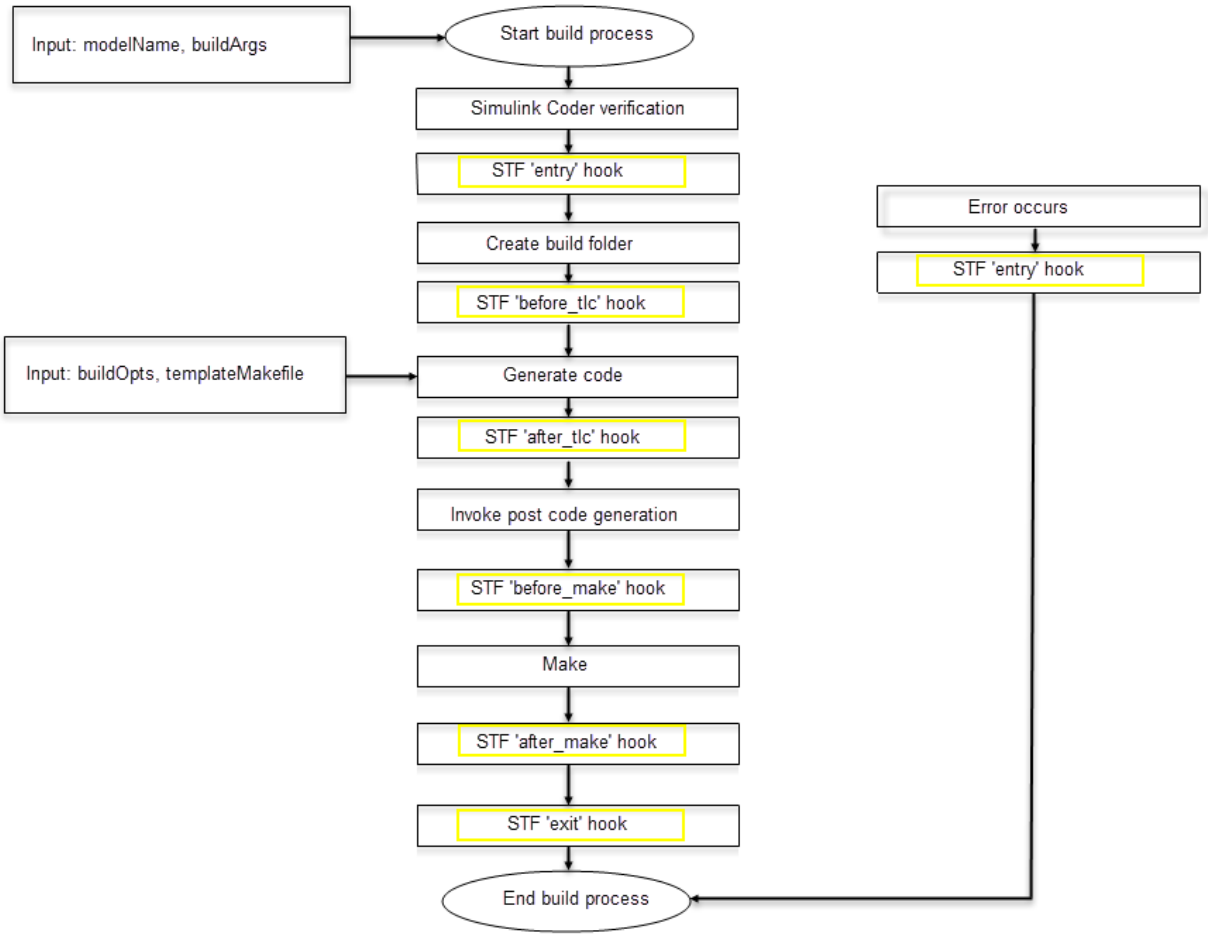
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for *STF*`_make_rtw_hook` is:

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile,
buildOpts, buildArgs, buildInfo)
```

The arguments are defined as:

- `hookMethod`: Character vector specifying the stage of build process from which the *STF*`_make_rtw_hook` function is called. The following flow chart summarizes the build process, highlighting the hook points. Valid values for `hookMethod` are `'entry'`, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, `'exit'`, and `'error'`. The *STF*`_make_rtw_hook` function dispatches to the relevant code with a `switch` statement.



- `modelName`: Character vector specifying the name of the model. Valid at all stages of the build process.
- `rtwRoot`: Reserved.
- `templateMakefile`: Name of template makefile.
- `buildOpts`: A MATLAB structure that contains the Boolean field `codeWasUpToDate`. Valid for the `'before_make'`, `'after_make'`, and `'exit'` stages only.
- `buildArgs`: Character vector containing the argument to `make_rtw`. When you invoke the build process, `buildArgs` is copied from the argument following `"make_rtw"` in the **Configuration Parameters** + **Code Generation** + **Make command** field.

  For example, the following make arguments from the **Make command** field

  ```
  make_rtw VAR1=0 VAR2=4
  ```

  generate the following:

  ```
  % make -f untitled.mk VAR1=0 VAR2=4
  ```

The `buildArgs` argument does not apply for toolchain approach builds because these builds do not allow adding make arguments to the `make_rtw` call. On the compiler command line, to provide custom definitions (for example, `VAR1=0 VAR2=4`) that apply for both TMF approach and toolchain approach builds, use the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Defines** field.

- `buildInfo`: The `RTW.BuildInfo` object that contains information for compiling and linking generated code. Available only for the `'after_tlc'`, `'before_make'`, `'after_make'`, and `'exit'` stages. The information in the object at the end of the `'after_tlc'` stage might not be complete. In later stages, the `'before_make'` and `'after_make'` hook methods can also add information to the object. For more information about using the `RTW.BuildInfo` object, see "Code Compilation Customization".

Applications for STF_make_rtw_hook.m

This section shows how you can use the *STF*`_make_rtw_hook.m` hook methods.

In general, use the `'entry'` hook to initialize the build process, for example, to change or validate settings before code is generated. One application for the `'entry'` hook is to rerun the auto-configuration script that initially ran at target selection time to compare model parameters before and after the script executes, for validation purposes.

The other hook points, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, `'exit'`, and `'error'` are useful for interfacing with external tool chains, source control tools, and other environment tools.

For example, you can use the *STF*`_make_rtw_hook.m` file at a stage after `'entry'` to obtain the path to the build folder. At the `'exit'` stage, you can then locate generated code files within the build folder and check them into your version control system. You can use `'error'` to clean up static or global data used by the hook function when an error occurs during code generation or the build process.

Note The build process temporarily changes the MATLAB working folder to the build folder for stages `'before_make'`, `'after_make'`, `'exit'`, and `'error'`. Your *STF*`_make_rtw_hook.m` file must not make incorrect assumptions about the location of the build folder. At a point after the `'entry'` stage, you can obtain the path to the build folder. In the following MATLAB code example, the build folder path is returned as a character vector to the variable `buildDirPath`.

```
buildDirPath = rtwprivate('get_makertwsettings',gcs,'BuildDirectory');
```

Note Do not use the *STF*`_make_rtw_hook.m` file to change the model configuration. For example, do not use a hook method to:

- Switch between model variants.
- Call the `set_param` function.

Changing the model configuration might produce unexpected code generation results.

Control Code Regeneration Using STF_make_rtw_hook.m

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. (For details on

the criteria, see “Control Regeneration of Top Model Code” on page 59-49.) If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code. If the checks indicate that the top model generated code is current with respect to the model, and model settings do not require full regeneration, the build process omits regeneration of the top model code.

Regardless of whether the top model code is regenerated, the build process subsequently calls the build process hooks, including *STF*`_make_rtw_hook` functions and the post code generation command. The following mechanisms allow you to perform actions related to code regeneration in the *STF*`_make_rtw_hook` functions:

- To force code regeneration, use the following function call from the `'entry'` hook:

  ```
  rtw.targetNeedsCodeGen('set', true);
  ```

- In hooks from `'before_tlc'` through `'exit'`, the `buildOpts` structure passed to the hook has a Boolean field `codeWasUpToDate`. The field is set to `true` if model code was up to date and code was not regenerated, or `false` if code was not up to date and code was regenerated. You can customize hook actions based on the value of this field. For example:

  ```
  ...
  case 'before_tlc'
      if buildOpts.codeWasUpToDate
          %Perform hook actions for up to date model
      else
          %Perform hook actions for full code generation
      end
  ...
  ```

Use STF_make_rtw_hook.m for Your Build Procedure

To create a custom *STF*`_make_rtw_hook` hook file for your build procedure, copy and edit the example `ert_make_rtw_hook.m` file, which is located in the folder*`matlabroot`*`/toolbox/coder/embeddedcoder` (open), as follows:

1. Copy `ert_make_rtw_hook.m` to a folder in the MATLAB path. Rename it in accordance with the naming conventions described in “Conventions for Using the STF_make_rtw_hook File” on page 93-25. For example, to use it with the GRT target `grt.tlc`, rename it to `grt_make_rtw_hook.m`.
2. Rename the `ert_make_rtw_hook` function within the file to match the file name.
3. Implement the hooks that you require by adding code to case statements within the `switch hookMethod` statement.

Hook Method after_tlc

The `after_tlc` hook method is available solely for backwards compatibility.

The format of the generated code during the `after_tlc` stage is not the final format.

See Also

Related Examples

- “Customize Build Process with sl_customization.m” on page 93-29

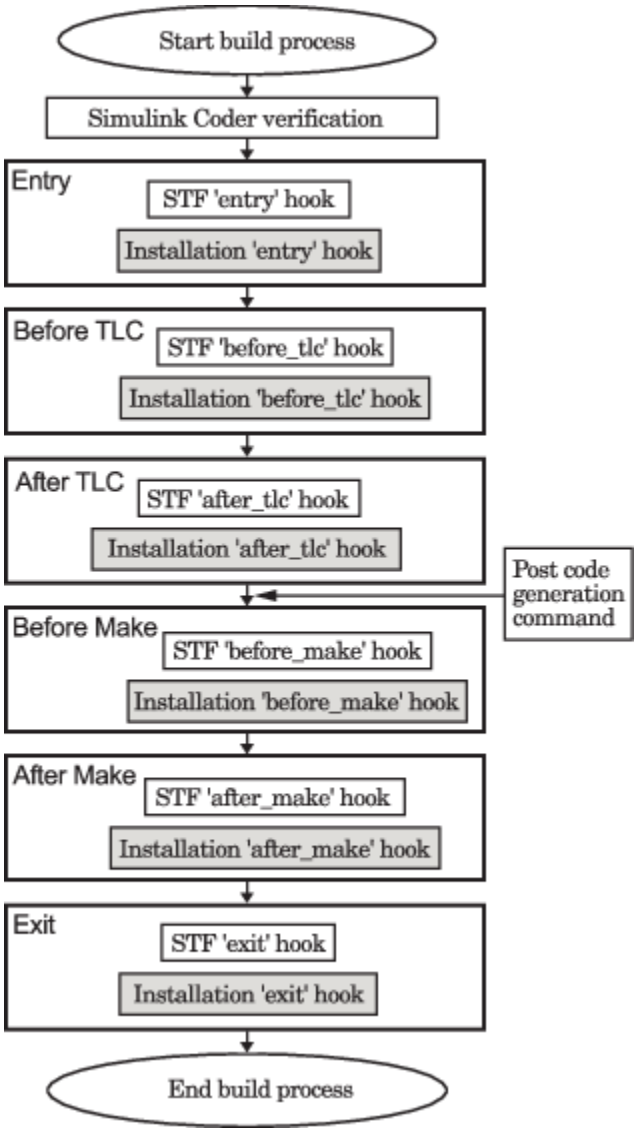
Customize Build Process with sl_customization.m

The Simulink customization file `sl_customization.m` is a mechanism that allows you to use MATLAB to customize the build process interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see “Register Customizations with Simulink”.

The sl_customization.m File

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in “Customize Build Process with STF_make_rtw_hook File” on page 93-25) and post-code generation commands (described in “Customize Post-Code-Generation Build Processing” on page 93-7).

The following figure shows the relationship between installation-level hooks and the other available mechanisms for customizing the build process.



Register Build Process Hook Functions Using sl_customization.m

To register installation-level hook functions that are invoked during the build process, you create a MATLAB function called `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

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

```
hObj = cm.RTWBuildCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following method for registering build process hook customizations:

- `addUserHook(hObj, hookType, hook)`

  Registers the MATLAB hook script or function specified by `hook` for the build process stage represented by `hookType`. The valid values for `hookType` are `'entry'`, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, and `'exit'`.

Use this method to register installation-specific hook functions in your instance of the `sl_customization` function.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart the Simulink session or enter the following command in the Command Window to enable the changes:

```
sl_refresh_customizations
```

Variables Available for sl_customization.m Hook Functions

The following variables are available for `sl_customization.m` hook functions to use:

- `modelName` — The name of the Simulink model (valid for all stages)
- `dependencyObject` — An object containing the dependencies of the generated code (valid only for the `'after_make'` stage)

A hook script can directly access the valid variables. A hook function can pass the valid variables as arguments to the function. For example:

```
hObj.addUserHook('after_make', 'afterMakeFunction(modelName,dependencyObject);');
```

Example of Build Process Customization with sl_customization.m

The `sl_customization.m` file shown in the example, **sl_customization.m for Build Process Customizations**, uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the `'entry'` and `'after_tlc'` stages of the build process. For the hook function source code, see the **CustomRTWEntryHook.m** and **CustomRTWPostProcessHook.m** examples.

Example 93.1. sl_customization.m for Build Process Customizations

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

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

% Register build process hooks
hObj.addUserHook('entry', 'CustomRTWEntryHook(modelName);');
hObj.addUserHook('after_tlc', 'CustomRTWPostProcessHook(modelName);');

end
```

Example 93.2. CustomRTWEntryHook.m

```
function [str, status] = CustomRTWEntryHook(modelName)
str =sprintf('Custom entry hook for model ''%s.''',modelName);
disp(str)
status =1;
```

Example 93.3. CustomRTWPostProcessHook.m

```
function [str, status] = CustomRTWPostProcessHook(modelName)
str =sprintf('Custom post process hook for model ''%s.''',modelName);
```

```
disp(str)
status =1;
```

If you include the above three files on the MATLAB path of the Simulink installation that you want to customize, the coded hook function messages appear in the displayed output for builds. For example, if you open the ERT-based model `UserDefinedDataTypes`, open the **Code Generation** pane of the Configuration Parameters dialog box, and press **Ctrl+B** to initiate a build, the following messages are displayed:

```
>> UserDefinedDataTypes

### Starting build procedure for model: UserDefinedDataTypes
Custom entry hook for model 'UserDefinedDataTypes.'
Custom post process hook for model 'UserDefinedDataTypes.'
### Successful completion of build procedure for model: rtwdemo_udt
>>
```

See Also

Related Examples

- “Use STF_make_rtw_hook.m for Your Build Procedure” on page 93-28

Replace STF_rtw_info_hook Supplied Target Data

Prior to MATLAB Release 14, custom targets supplied target-specific information with a hook file (referred to as *STF*`_rtw_info_hook.m`). The *STF*`_rtw_info_hook` specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The *STF*`_rtw_info_hook` mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify properties that were formerly specified in your *STF*`_rtw_info_hook` file.

For backward compatibility, existing *STF*`_rtw_info_hook` files are available. However, you should convert your target and models to use of the **Hardware Implementation** pane. See “Configure Production and Test Hardware”.

Custom Target Development in Simulink Coder

- “Custom Targets” on page 94-2
- “Sample Custom Targets” on page 94-4
- “Target Development Folders, Files, and Builds” on page 94-6
- “Customize System Target Files” on page 94-19
- “Customize Template Makefiles” on page 94-44
- “Custom Target Optional Features” on page 94-57
- “Support Toolchain Approach with Custom Target” on page 94-59
- “Support Model Referencing” on page 94-61
- “Support Compiler Optimization Level Control” on page 94-65
- “Support C Function Prototype Control” on page 94-67
- “Support C++ Class Interface Control” on page 94-69
- “Support Concurrent Execution of Multiple Tasks” on page 94-71
- “Device Drivers” on page 94-72

Custom Targets

A custom target is a software package that you create to deploy MATLAB and Simulink designs to target hardware. Using the package, you can perform tasks to optimize, prototype, verify, and deploy an application to your hardware. This table lists tasks that you can perform by using `target` classes and functions.

Task	See

Register new hardware devices and hardware boards for **Hardware Implementation** settings.	“Register New Hardware Devices” on page 10-13
	
	`target.Board`
	
	`target.Processor`
	
	`target.LanguageImplementation`

Build a target application using a CMake-based toolchain.	“Create Custom CMake Toolchain Definition” on page 59-17 “Generated Code Compilation”

Deploy, download, and run an application on target hardware.	`target.ExecutionTool`
	
	`target.SystemCommandExecutionTool`

Tune parameters and monitor target application signals during an external mode simulation.	“Set Up External Mode Connectivity Between Simulink and Target Hardware” on page 60-35 “External Mode Simulation by Using XCP Communication” on page 60-8 “Customize XCP Server Software” on page 60-45
Test numerical equivalence between your MATLAB code or Simulink model and the generated code by running a processor-in-the-loop (PIL) simulation.	“Set Up PIL Connectivity by Using target Package” on page 88-48 “Configure and Run PIL Simulation” on page 88-24
Perform code execution profiling through a PIL simulation.	“Set Up PIL Connectivity by Using target Package” on page 88-48 “Create Execution-Time Profile for Generated Code” on page 81-2

This table lists additional tasks that you can perform alongside a custom target defined by `target` classes and functions.

Task	See
Generate or register a main file to schedule the generated code	“Deploy Applications to Target Hardware” on page 71-2 “Model Multicore Concurrent Tasking Application” on page 3-81 Generate an example main program Target operating system “Generate Source and Header Files with a Custom File Processing (CFP) Template” on page 55-62 “Support C Function Prototype Control” on page 94-67 “Support C++ Class Interface Control” on page 94-69
Register build hooks.	“Customize Build Process with sl_customization.m” on page 93-29 “Customize Post-Code-Generation Build Processing” on page 93-7
Optimize generated code by using a processor-specific code-replacement library.	“Code Replacement”
Integrate real-time operating systems and device drivers.	“Device Drivers” on page 94-72
Define a custom system target file with hooks and custom settings to add target-specific actions to the build process.	“Target Platform Device Customization” “Customize System Target Files” on page 94-19 “Customize Build Process with STF_make_rtw_hook File” on page 93-25

You can also create custom software for ARM processors, which is based on MathWorks reference targets. For example, see:

- “Embedded Coder Support Package for ARM Cortex-A Processors”
- “Embedded Coder Support Package for ARM Cortex-M Processors”

Sample Custom Targets

There are technical solutions on the MathWorks Web site that you can use as a starting point to create your own target solution. The solutions provide guides to the following tasks for creating custom targets:

- Methods of embedding code onto a custom processor
- Creating a system target file
- Customizing the makefile and main file
- Adding compiler, chip, and board specific information
- Integrating legacy code and device drivers
- Creating blocks and libraries
- Implementing processor-in-the-loop (PIL) testing.

Consider these options:

- Download the embedded targets development guide `zip` file from this web page:

  Is there an example guide on developing an embedded target...?

  The `zip` file provides example files and a guide to developing a custom embedded target. The guide is divided into two parts, one on creating a generic custom target and another on creating a target for the Freescale® S12X processor using the Cosmic Compiler.

  Read the example guide along with this document to understand the tasks for developing embedded targets.
- For more detailed example files for specific processors, see:

  - Is there an example Freescale S12X target... using the Cosmic Compiler?
  - Is there an example Freescale S12X target... using the CodeWarrior Compiler?

  These example kits contain example models, code generation files, and instruction guides on generating and testing code for the processor. The Cosmic example illustrates the use of the target connectivity API for processor-in-the-loop (PIL) testing. The CodeWarrior example does not have PIL but shows CAN Calibration Protocol (CCP) and Simulink external mode.

  The intent of the example kits is to provide working examples that you can use as a base to create your own target solution. The intent is not to provide a full featured and maintained Embedded Target product like those provided by MathWorks or third-party products, as listed on the Embedded Coder Hardware Support Web page.
- You can watch a video that shows an overview of the example kit `www.mathworks.com/videos/programming-the-freescale-s12x-target-68811.html`.
- For an example target for the ARM9 (STR9) processor, see Is there an example ARM9 (STR9) target... using the GNU ARM Compiler and Hitex STR9-comStick?.

If you have questions on specific targets, email mytarget@mathworks.com.

The example kits and this document describe Embedded Coder features such as customized `ert` system target files and processor-in-the-loop testing, but you can study the examples as a starting point for use with Simulink Coder targets.

See Also

More About

- “Custom Targets”
- “Customize System Target Files”
- “Customize Template Makefiles”
- “Custom Target Optional Features”
- “Support Toolchain Approach with Custom Target”
- “Device Drivers”

Target Development Folders, Files, and Builds

Target development mechanics work with a number of folder and file types. The following topics provide the information to develop custom targets, configure folder usage, and use custom targets in the build process.

Folder and File Naming Conventions

You can use a single folder for your custom target files, or if desired you can use subfolders, for example containing files associated with specific development environments or tools.

For a custom target implementation, the recommended folder and file naming conventions are

- Use *only* lowercase in folder names, filenames, and extensions.
- Do not embed spaces in folder names. Spaces in folder names cause errors with many third-party development environments.
- Include desired folders in the MATLAB path
- Do *not* place your custom target folder anywhere in the MATLAB folder tree (that is, in or under the `matlabroot` folder). If you place your folder under `matlabroot` you risk losing your work if you install a new MATLAB version (or reinstall the current version).

The following sections explain how to organize your target folders and files and add them to your MATLAB path. They also provide high-level descriptions of the files.

In this document, `mytarget` is a placeholder name that represents folders and files that use the target's name. The names `dev_tool1`, `dev_tool2`, and so on represent subfolders containing files associated with development environments or tools. This document describes an example structure where the folder `mytarget` contains subfolders for `mytarget`, `blocks`, `dev_tool1`, `dev_tool2`. The top level folder `mytarget` is the *target root folder*.

Components of a Custom Target

- “Overview” on page 94-6
- “Code Components” on page 94-7
- “Control Files” on page 94-8

Overview

The components of a custom target are files located in a hierarchy of folders. The top-level folder in this structure is called the *target root folder*. The target root folder and its contents are named, organized, and located on the MATLAB path according to conventions described in “Folder and File Naming Conventions” on page 94-6.

The components of a custom target include

- Code components: C source code that supervises and supports execution of generated model code.
- Control files:
  - A system target file to control the code generation process.

  - File(s) to control the building of an executable from the generated code. In a traditional make-based environment, a template makefile (TMF) generates a makefile for this purpose. Another approach is to generate project files in support of a modern integrated development environment (IDE) such as the Freescale Semiconductor CodeWarrior IDE.
  - Hook files: Optional TLC and MATLAB program files that can be invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.
- Other target files: Files that let you integrate your target into the MATLAB environment. For example, you can provide an `info.xml` file to make your target block libraries and examples available from a MATLAB session.

The next sections introduce key concepts and terminology you need to know to develop each component. References to more detailed information sources are provided.

Code Components

An executable program containing code generated from a Simulink model consists of a number of code modules and data structures. These fall into two categories: application components and execution support files.

Application Components

Application components are those which are specific to a particular model; they implement the functions represented by the blocks in the model. Application components are not specific to the target. Application components include

- Modules generated from the model
- User-written blocks (S-functions)
- Parameters of the model that are visible, and can be interfaced to, external code

Execution Support Files

A number of code modules and data structures, referred to collectively as the *execution support files*, are responsible for managing and supporting the execution of the generated program. The execution support files modules are not automatically generated. Depending on the requirements of your target, you must implement certain parts of the execution support files. Execution Support Files summarizes the execution support files.

Execution Support Files

You Provide...	The Code Generator Provides...
Customized main program	Generic main program
Timer interrupt handler to run model	Execution engine and integration solver (called by timer interrupt handler)
Other interrupt handlers	Example interrupt handlers (Asynchronous Interrupt blocks)
Device drivers	Example device drivers
Data logging, parameter tuning, signal monitoring, and external mode support	Data logging, parameter tuning, signal monitoring, and external mode APIs

User-Written Execution Support Files

The code generator provides most of the execution support files. Depending on the requirements of your target, you must implement some or all of the following elements:

- A timer *interrupt service routine* (ISR). The timer runs at the program's base sample rate. The timer ISR is responsible for operations that must be completed within a single clock period, such as computing the current output sample. The timer ISR usually calls the `rt_OneStep` function.

  If you are targeting a real-time operating system (RTOS), your generated code usually executes under control of the timing and task management mechanisms provided by the RTOS. In this case, you may not have to implement a timer ISR.
- The *main program*. Your main program initializes the blocks in the model, installs the timer ISR, and executes a background task or loop. The timer periodically interrupts the main loop. If the main program is designed to run for a finite amount of time, it is also responsible for cleanup operations — such as memory deallocation and masking the timer interrupt — before terminating the program.

  If you are targeting a real-time operating system (RTOS), your main program most likely spawns tasks (corresponding to the sample rates used in the model) whose execution is timed and controlled by the RTOS.

  Your main program typically is based on a generated or static main program. For details on the structure of the execution support files, code execution, and guidelines for customizing main programs, see “Deploy Applications to Target Hardware” on page 71-2.
- *Device drivers*. Drivers communicate with I/O devices on your target hardware. In production code, device drivers are normally implemented as inlined S-functions.
- *Other interrupt handlers*. If your models need to support asynchronous events, such as hardware generated interrupts and asynchronous read and write operations, you must supply interrupt handlers. The Interrupt Templates library provides examples.
- *Data logging, parameter tuning, signal monitoring, and external mode support*. It is atypical to implement rapid prototyping features such as external mode support in an embedded target. However, it is possible to support these features by using standard code generator APIs. See “Generated Code Interfacing” for details.

Control Files

The code generation and build process is directed by a number of TLC and MATLAB files collectively called *control files*. This section introduces and summarizes the main control files.

Top-Level Control File (make_rtw)

The build process is initiated when you press **Ctrl+B**. At this point, the build process parses the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box, expecting to find the name of a MATLAB command that controls the build process (as well as optional arguments to that command). The default command is `make_rtw`, and the default top-level control file for the build process is `make_rtw.m`.

Note `make_rtw` is an internal MATLAB command used by the build process. Normally, target developers do not need detailed knowledge of how `make_rtw` works. (The details for target developers are described in “Target Development and the Build Process” on page 94-14.) You should not invoke `make_rtw` directly from MATLAB code, and you should not customize `make_rtw.m`.

The `make_rtw.m` file contains the logic required to execute your target-specific control files, including a number of hook points for execution of your custom code. `make_rtw` does the following:

- Passes optional arguments in to the build process
- Performs required preprocessing before code generation
- Executes the system target file to perform code generation (and optional HTML report generation)
- Processes the TMF to generate a makefile
- Invokes a make utility to execute the makefile and build an executable
- Performs required post-processing (such as generating calibration data files or downloading the generated executable to the target)

System Target File

The Target Language Compiler (TLC) generates target-specific C or C++ code from a partial description of your Simulink block diagram (*`model`*`.rtw`). The Target Language Compiler reads *`model`*`.rtw` and executes a program consisting of several target files (`.tlc` files.) The system target file, at the top level of this program, controls the code generation process. The output of this process is a number of source files, which are fed to your development system's make utility.

You need to create a customized system target file to set code generation parameters for your target. You should copy, rename, and modify the standard ERT system target file (*`matlabroot`*`/rtw/c/ert/ert.tlc`).

The detailed structure of the system target file is described in “Customize System Target Files” on page 94-19.

Note The system target file selects whether the target supports the toolchain approach or template makefile approach for code generation. See “Customize System Target Files” on page 94-19.

Template Makefile (TMF)

A TMF provides information about your model and your development system. The build process uses this information to create a makefile (`.mk` file) that builds an executable program.

Some targets implement more than one TMF, in order to support multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

The Embedded Coder software provides a large number of TMFs suitable for different types of development computer systems. These TMFs are located in *`matlabroot`*`/toolbox/coder/compile/tmf`. The standard TMFs are described in “Template Makefiles and Make Options”.

The detailed structure of the TMF is described in “Customize Template Makefiles” on page 94-44.

Note The system target file selects whether the target supports the toolchain approach or template makefile approach for code generation. See “Customize System Target Files” on page 94-19.

Hook Files

The build process allows you to supply optional *hook files* that are executed at specified points in the code generation and make process. You can use hook files to add target-specific actions to the build process.

The hook files must follow well-defined naming and location requirements. “Folder and File Naming Conventions” on page 94-6 describes these requirements.

Key Folders Under Target Root (mytarget)

- “Target Root Folder (mytarget)” on page 94-10
- “Target Folder (mytarget/mytarget)” on page 94-10
- “Target Block Folder (mytarget/blocks)” on page 94-10
- “Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)” on page 94-11
- “Target Source Code Folder (mytarget/src)” on page 94-11

Target Root Folder (mytarget)

This folder contains the key subfolders for the target (see “Folder and File Naming Conventions” on page 94-6). You can also locate miscellaneous files (such as a `readme` file) in the target root folder. The following sections describe required and optional subfolders and their contents.

Target Folder (mytarget/mytarget)

This folder contains files that are central to the target, such as the system target file (STF) and template makefile (TMF). “Key Files in Target Folder (mytarget/mytarget)” on page 94-12 summarizes the files that should be stored in `mytarget/mytarget`, and provides pointers to detailed information about these files.

Note `mytarget/mytarget` should be on the MATLAB path.

Target Block Folder (mytarget/blocks)

If your target includes device drivers or other blocks, locate the block implementation files in this folder. `mytarget/blocks` contains

- Compiled block MEX-files
- Source code for the blocks
- TLC inlining files for the blocks
- Library models for the blocks (if you provide your blocks in one or more libraries)

Note `mytarget/blocks` should be on the MATLAB path.

You can also store example models and supporting files in `mytarget/blocks`. Alternatively, you can create a `mytarget/mytargetdemos` folder, which should also be on the MATLAB path.

To display your blocks in the standard Simulink Library Browser and/or integrate your example models into the MATLAB session environment , you can create the files described below and store them in `mytarget/blocks`.

mytarget/blocks/slblocks.m

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser.

Example 94.1. Example slblocks.m File

```
function blkStruct = slblocks
% Information for "Blocksets and Toolboxes" subsystem
blkStruct.Name = sprintf('Embedded Target\n for MYTARGET');
blkStruct.OpenFcn = 'mytargetlib';
blkStruct.MaskDisplay = 'disp(''MYTARGET'')';

% Information for Simulink Library Browser
Browser(1).Library = 'mytargetlib';
Browser(1).Name    = 'Embedded Target for MYTARGET';
Browser(1).IsFlat  = 1;% Is this library "flat" (i.e. no subsystems)?

blkStruct.Browser = Browser;
```

mytarget/blocks/demos.xml

This file provides information about the components, organization, and location of example models. MATLAB software uses this information to place the example in the MATLAB session environment.

Example 94.2. Example demos.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
   <name>Embedded Target for MYTARGET</name>
   <type>simulink</type>
   <icon>$toolbox/matlab/icons/boardicon.gif</icon>
   <description source = "file">mytarget_overview.html</description>

   <demosection>
      <label>Multirate model</label>
      <demoitem>
         <label>MYTARGET demo</label>
         <file>mytarget_overview.html</file>
         <callback>mytarget_model</callback>
      </demoitem>
   </demosection>

</demos>
```

Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)

These folders contain files associated with specific development environments or tools (`dev_tool1`, `dev_tool2`, etc.). Normally, your target supports at least one such development environment and invokes its compiler, linker, and other utilities during the build process. `mytarget/dev_tool1` includes linker command files, startup code, hook functions, and other files required to support this process.

For each development environment, you should provide a separate folder.

Target Source Code Folder (mytarget/src)

This folder is optional. If the complexity of your target requires it, you can use `mytarget/src` to store common source code and configuration code (such as boot and startup code).

Key Files in Target Folder (mytarget/mytarget)

- “Introduction” on page 94-12
- “mytarget.tlc” on page 94-12
- “mytarget.tmf” on page 94-12
- “mytarget_genfiles.tlc” on page 94-12
- “mytarget_main.c” on page 94-13
- “STF_make_rtw_hook.m” on page 94-13
- “STF_rtw_info_hook.m (obsolete)” on page 94-13
- “info.xml” on page 94-13
- “mytarget_overview.html” on page 94-13

Introduction

The target folder `mytarget/mytarget` contains key files in your target implementation. These include the system target file, template makefile, main program module, and optional M and TLC hook files that let you add target-specific actions to the build process. The following sections describe the key target folder files.

mytarget.tlc

`mytarget.tlc` is the system target file. Functions of the system target file include

- Making the target visible in the System Target File Browser
- Definition of code generation options for the target (inherited and target-specific)
- Providing an entry point for the top-level control of the TLC code generation process

You should base your system target file on `ert.tlc`, the system target file provided by Embedded Coder software.

“Customize System Target Files” on page 94-19 gives detailed information on the structure of the system target file, and also gives instructions on how to customize a system target file to

- Display your target in the System Target File Browser
- Add your own target options to the Configuration Parameters dialog box
- Tailor the code generation and build process to the requirements of your target

mytarget.tmf

`mytarget.tmf` is the template makefile for building an executable for your target.

For basic information on the structure and operation of template makefiles, see “Customize Template Makefiles” on page 94-44.

mytarget_genfiles.tlc

This file is optional. `mytarget_genfiles.tlc` is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads. See “Using mytarget_genfiles.tlc” on page 94-31 for details.

mytarget_main.c

A main program module is required for your target. To provide a main module, you can either

- Modify the `rt_main.c` or `rt_cppclass_main.cpp` module provided by the software
- Generate `mytarget_main.c` or `.cpp` during the build process

For a description of the operation of main programs, see "Deploy Applications to Target Hardware" on page 71-2. The section also contains guidelines for generating and modifying a main program module.

STF_make_rtw_hook.m

STF`_make_rtw_hook.m` is an optional hook file that you can use to invoke target-specific functions or executables at specified points in the build process. *STF*`_make_rtw_hook.m` implements a function that dispatches to a specific action depending on the `method` argument that is passed into it.

"Customize Build Process with STF_make_rtw_hook File" describes the operation of the *STF*`_make_rtw_hook.m` hook file in detail.

STF_rtw_info_hook.m (obsolete)

Prior to Release 14, custom targets supplied target-specific information with a hook file (referred to as *STF*`_rtw_info_hook.m`). The *STF*`_rtw_info_hook` specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The *STF*`_rtw_info_hook` mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify the properties that were formerly specified in your *STF*`_rtw_info_hook` file.

For backward compatibility, existing *STF*`_rtw_info_hook` files are still available. However, you should convert your target and models to use the **Hardware Implementation** pane. See "Configure Run-Time Environment Options".

info.xml

This file provides information to MATLAB software that specifies where to display the target toolbox in the MATLAB session environment. For more information, see "Display Custom Documentation".

mytarget_overview.html

By convention, this file serves as home page for the target examples.

The `<description>` field in `demos.xml` should point to `mytarget_overview.html` (see "mytarget/blocks/demos.xml" on page 94-11).

Example mytarget_overview.html File

```
<html>
<head><title>Embedded Target for MYTARGET</title></head><body>
<p style="color:#990000; font-weight:bold; font-size:x-large">Embedded Target
for MYTARGET Example Model</p>

<p>This example provides a simple model that allows you to generate an executable
for a supported target board. You can then download and run the executable and
set breakpoints to study and monitor the execution behavior.</p>
```

```
</body>
</html>
```

Additional Files for Externally Developed Targets

- “Introduction” on page 94-14
- “mytarget/mytarget/mytarget_setup.m” on page 94-14
- “mytarget/mytarget/doc” on page 94-14

Introduction

If you are developing an embedded target that is not installed into the MATLAB tree, you should provide a target setup script and target documentation within `mytarget/mytarget`, for the convenience of your users. The following sections describe the required materials and where to place them.

mytarget/mytarget/mytarget_setup.m

This file script adds paths for your target to the MATLAB path. Your documentation should instruct users to run the script when installing the target.

You should include a call to the MATLAB function `savepath` in your `mytarget_setup.m` script. This function saves the added paths, so users need to run `mytarget_setup.m` only once.

The following code is an example `mytarget_setup.m` file.

```
function mytarget_setup()
curpath = pwd;
tgtpath = curpath(1:end-length('\mytarget'));
addpath(fullfile(tgtpath, 'mytarget'));
addpath(fullfile(tgtpath, 'dev_tool1'));
addpath(fullfile(tgtpath, 'blocks'));
addpath(fullfile(tgtpath, 'mytargetdemos'));
savepath;
disp('MYTARGET Target Path Setup Complete.');
```

mytarget/mytarget/doc

You should put the documentation related to your target in the folder `mytarget/mytarget/doc`.

Target Development and the Build Process

- “About the Build Process” on page 94-14
- “Build Process Phases and Information Passing” on page 94-15
- “Additional Information Passing Techniques” on page 94-16

About the Build Process

To develop an embedded target, you need a thorough understanding of the build process. Your embedded target uses the build process and might require you to modify or customize the process. A general overview of code generation and the build process is given in “Source Code Generation”.

This section supplements that overview with a description of the build process as customized by the Embedded Coder software. The emphasis is on points in the process where customization hooks are available and on passing information between different phases of the process.

This section concludes with “Additional Information Passing Techniques” on page 94-16, describing assorted tips and tricks for passing information during the build process.

Build Process Phases and Information Passing

It is important to understand where (and when) the build process obtains required information. Sources of information include

- The *`model`*`.rtw` file, which provides information about the generating model. The information in *`model`*`.rtw` is available to target TLC files.
- The code generation panes of the Configuration Parameters dialog box. Options (both general and target-specific) are provided through check boxes, menus, and edit fields. You can associate options with TLC variables in the `rtwoptions` data structure. Use the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Code information** > **Defines** field to define makefile tokens .
- The selected toolchain (for toolchain approach builds) or selected template makefile `.tmf` (for template makefile approach builds); these generate the model-specific makefile.
- Environment variables on the host computer. Environment variables provide additional information about installed development tools.
- Other target-specific files such as target-related TLC files, linker command files, or project files.

It is also important to understand the several phases of the build process and how to pass information between the phases. The build process comprises several high-level phases:

- Execution of the top-level file (`slbuild.m`) to sequence through the build process for a target
- Conversion of the model into the TLC input file (*`model`*`.rtw`)
- Generation of the target code by the TLC compiler
- Compilation of the generated code with `make` or other utilities
- Transmission of the final generated executable to the target hardware with a debugger or download utility

It is helpful to think of each phase of the process as a different “environment” that maintains its own data. These environments include

- MATLAB code execution environment (MATLAB)
- Simulink
- Target Language Compiler execution environment
- makefile
- Development environments such as and IDE or debugger

In each environment, you might get information from the various sources mentioned above. For example, during the TLC phase, execute MATLAB file might execute to obtain information from the MATLAB environment. Also, a given phase may generate information for a subsequent phase.

See “Key Files in Target Folder (mytarget/mytarget)” on page 94-12 for details on the available MATLAB file and TLC hooks for information passing, with code examples.

Additional Information Passing Techniques

This section describes a number of useful techniques for passing information among different phases of the build process.

tlcvariable Field in rtwoptions Structure

Parameters on the code generation panes of the Configuration Parameters dialog box can be associated with a TLC variable, and specified in the `tlcvariable` field of the option's entry in the `rtwoptions` structure. The variable value is passed on the command line when TLC is invoked. This provides a way to make code generation parameters and their values available in the TLC phase.

See “System Target File Structure” on page 94-19 for further information.

makevariable Field in rtwoptions Structure

You can associate code generation parameters with a template makefile token, that you specify in the `makevariable` field of the option's entry in the `rtwoptions` structure. If a token of the same name as the `makevariable` name exists in the TMF, the token is updated with the option value when the final makefile is created. If the token does not exist in the TMF, the `makevariable` is passed in on the command line when make is invoked. Thus, in either case, the `makevariable` is available to the makefile.

See “System Target File Structure” on page 94-19 for further information.

Accessing Host Environment Variables

You can access host shell environment variables at the MATLAB command line by entering the `getenv` command. For example:

```
getenv ('MSDEVDIR')

ans =

D:\Applications\Microsoft Visual Studio\Common\MSDev98
```

To access the same information from TLC, use the `FEVAL` directive to invoke `getenv`.

```
%assign eVar = FEVAL("getenv","<varname>")
```

Supplying Development Environment Information to Your Template Makefile

An embedded target must tie the build process to target-specific development tools installed on a host computer. For the make process to run these tools, the TMF must be able to determine the name of the tools, the path to the compiler, linker, and other utilities, and possibly the host operating system environment variable settings.

Require the end user to modify the target TMF. The user enters path information (such as the location of a compiler executable), and possibly host operating system environment variables, as make variables. This allows the TMF to be tailored to specific needs.

Using MATLAB Application Data

Application data provides a way for applications to save and retrieve data stored with the GUI. This technique enables you to create what is essentially a user-defined property for an object, and use this property to store data for use in the build process. For more information, see `setappdata` and `getappdata`.

The following code examples illustrates the use of application data to pass information to TLC.

This file, `tlc2appdata.m`, stores the `data` passed in as application data under the name passed in (`appDataName`).

```
function k = tlc2appdata(appDataName,data)
  disp([mfilename,': ',appDataName,' ', data]);
  setappdata(0,appDataName,data);
  k = 0;  % TLC expects a return value for FEVAL.
```

The following sample TLC file uses the `FEVAL` directive to invoke `tlc2appdata.m` to store arbitrary application data, under the name `z80`.

```
%% test.tlc
%%
%assign myApp = "z80"
%assign myData = "314159"
%assign dummy = FEVAL("tlc2appdata",myApp,myData)
```

To test this technique:

1. Create the `tlc2appdata.m` file as shown. Check that `tlc2appdata.m` is stored in a folder on the MATLAB path.
2. Create the TLC file as shown. Save it as `test.tlc`.
3. Enter the following command at the MATLAB prompt to execute the TLC file:

   ```
   tlc test.tlc
   ```

4. Get the application data at the MATLAB prompt:

   ```
   k = getappdata(0,'z80')
   ```

   The function returns the value 314159.
5. Enter the following command.

   ```
   who
   ```

   Note that application data is not stored in the MATLAB workspace. Also observe that the z80 data is not visible. Using application data in this way has the advantage that it does not clutter the MATLAB workspace. Also, it helps prevent you from accidentally deleting your data, since it is not stored directly in your workspace.

A real-world use of application data might be to collect information from the *`model`*`.rtw` file and store it for use later in the build process.

Adding Block-Specific Information to the Makefile

The `rtwmakecfg` mechanism provides a method for inlined S-functions such as driver blocks to add information to the makefile. This mechanism is described in “Use rtwmakecfg.m API to Customize Generated Makefiles”.

See Also

More About

- “Custom Targets”

Customize System Target Files

This section provides information on the structure of the system target file, guidelines for customizing a system target file, and a basic tutorial that helps you to produce a skeletal system target file.

Control Code Generation With the System Target File

The system target file controls the code generation stage of the build process. The system target file also lets you control the presentation of your target to the end user. The system target file provides

- Definitions of variables that are fundamental to the build process, such as the value for the `CodeFormat` TLC variable
- The main entry point to the top-level TLC program that generates code
- Target information for display in the System Target File Browser
- A mechanism for defining target-specific code generation options (and other parameters related to the build process) and for displaying them in the Configuration Parameters dialog box
- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

Note that, although the system target file is a Target Language Compiler (TLC) file, it contains embedded MATLAB code. Before creating or modifying a system target file, you should acquire a working knowledge of TLC and of the MATLAB language. “Target Language Compiler” and “Scripts vs. Functions” describe the features and syntax of both the TLC and MATLAB languages.

While reading this section, you may want to refer to the system target files provided with the code generator. Most of these files are stored in the target-specific folders under *`matlabroot`*`/rtw/c` (open). Additional system target files are stored under *`matlabroot`*`/toolbox/rtw/targets` (open).

System Target File Naming and Location Conventions

A system target file must be located in a folder on the MATLAB path for the target to be displayed in the System Target File Browser and invoked in the build process. Follow the location and naming conventions for system target files and related target files given in “Folder and File Naming Conventions” on page 94-6.

System Target File Structure

- “Overview” on page 94-20
- “Header Comments” on page 94-21
- “TLC Configuration Variables” on page 94-22
- “TLC Program Entry Point and Related %includes” on page 94-22
- “RTW_OPTIONS Section” on page 94-23
- “rtwgensettings Structure” on page 94-23
- “Additional Code Generation Options” on page 94-25
- “Model Reference Considerations” on page 94-25

Overview

This section is a guide to the structure and contents of a system target file. The following listing shows the general structure of a system target file. Note that this is not a complete code listing of a system target file. The listing consists of excerpts from each of the sections that make up a system target file.

```
%%----------------------------
%% Header Comments Section
%%----------------------------
%% SYSTLC: Example Real-Time Target
%%    TMF: my_target.tmf MAKE: make_rtw
%% Initial comments contain directives for STF Browser.
%% Documentation, date, copyright, and other info may follow.
          ...
%selectfile NULL_FILE
         ...
%%----------------------------
%% TLC Configuration Variables Section
%%----------------------------
%% Assign code format, language, target type.
%%
%assign CodeFormat = "Embedded-C"
%assign TargetType = "RT"
%assign Language   = "C"
%%
%%----------------------------
%% TLC Program Entry Point
%%----------------------------
%% Call entry point function.
%include "codegenentry.tlc"
%%
%%----------------------------
%% (OPTIONAL) Generate Files for Build Process
%%----------------------------
%include "mytarget_genfiles.tlc"
%%----------------------------
%% RTW_OPTIONS Section
%%----------------------------
/%
  BEGIN_RTW_OPTIONS
  %% Define rtwoptions structure array. This array defines target-specific
  %% code generation variables, and controls how they are displayed.
  rtwoptions(1).prompt = 'example code generation options';
        ...
  rtwoptions(6).prompt = 'Show eliminated blocks';
  rtwoptions(6).type = 'Checkbox';
        ...
  %----------------------------------------%
  % Configure RTW code generation settings %
  %----------------------------------------%
        ...
  %%----------------------------
  %% rtwgensettings Structure
  %%----------------------------
  %% Define suffix text for naming build folder here.
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  %% Callback compatibility declaration
  rtwgensettings.Version = '1';

  %% (OPTIONAL) target inheritance declaration
  rtwgensettings.DerivedFrom = 'ert.tlc';
  %% (OPTIONAL) other rtwGenSettings fields...
        ...
  END_RTW_OPTIONS
%/
%%----------------------------
%% targetComponentClass - MATHWORKS INTERNAL USE ONLY
%% REMOVE NEXT SECTION FROM USER_DEFINED CUSTOM TARGETS
%%----------------------------
```

```
/%
  BEGIN_CONFIGSET_TARGET_COMPONENT
  targetComponentClass = 'Simulink.ERTTargetCC';
  END_CONFIGSET_TARGET_COMPONENT
%/
```

If you are creating a custom target based on an existing system target file, you must remove the `targetComponentClass` section (bounded by the directives `BEGIN_CONFIGSET_TARGET_COMPONENT` and `END_CONFIGSET_TARGET_COMPONENT`). This section is reserved for the use of targets developed internally by MathWorks.

Header Comments

These lines at the head of the file are formatted as TLC comments. They provide required information to the System Target File Browser and to the build process. Note that you must place the browser comments at the head of the file, before other comments or TLC statements.

The presence of the comments enables the code generator to detect system target files. When the System Target File Browser is opened, the code generator scans the MATLAB path for TLC files that have formatted header comments. The comments contain the following directives:

- SYSTLC: The descriptor that appears in the browser.
- TMF: Name of the template makefile (TMF) to use during build process. When the target is selected, this filename is displayed in the Template makefile field of the **Code Generation** pane of the Configuration Parameters dialog box.
- MAKE: `make` command to use during build process. When the target is selected, this command is displayed in the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box.

The following header comments are from *`matlabroot`*`/rtw/c/ert/ert.tlc`.

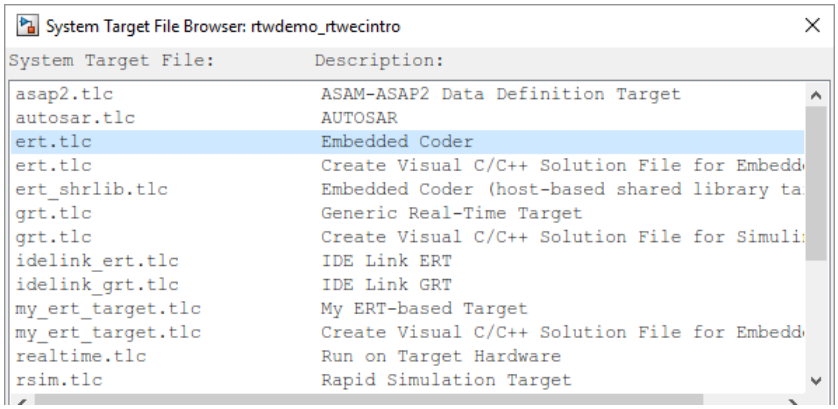
```
%% SYSTLC: Embedded Coder TMF: ert_default_tmf MAKE: make_rtw
%% SYSTLC: Create Visual C/C++ Solution File for Embedded Coder
.
.
.
```

Each comment can only contain a maximum of two lines.

If you do not specify the `TMF` field in the system target file, the file is still valid. To change the value for the`TemplateMakefile` parameter, you can instead use the callback function specified by `rtwgensettings.SelectCallback`.

You can also use the callback function specified by `rtwgensettings.SelectCallback` to change the value for external mode parameters, `ExtMode`, `ExtModeTransport`, `ExtModeMexFile`, or `ExtModeIntrfLevel`.

You can specify more than one group of directives in the header comments. Each such group is displayed as a different target configuration in the System Target File Browser. In the above example, the first two lines of code specify the default configuration of the ERT target. The next two lines specify a configuration that creates and builds a Microsoft Visual C++ Solution (`.sln`) file. The figure below shows how these configurations appear in the System Target File Browser.



See “Create a Custom Target Configuration” on page 94-33 for an example of customized header comments.

TLC Configuration Variables

This section of the system target file assigns global TLC variables that relate to the overall code generation process.

For an embedded target, in most cases you should simply use the global TLC variable settings used by the ERT target (`ert.tlc`). It is especially important that your system target file use the `'Embedded-C'` value for the `CodeFormat` TLC variable and uses the corresponding `rtwgensettings.DerivedFrom = 'ert.tlc'` in the `RTW_OPTIONS` section of the TLC file. Verify that values are assigned to the following variables:

- `CodeFormat`: The `CodeFormat` TLC variable selects generated code features. The `'Embedded-C'` value for this variable is used by the ERT target. Your ERT-based target should specify `'Embedded-C'` as the value for `CodeFormat`. This selection is designed for production code, minimal memory usage, static memory allocation, and a simplified interface to generated code.

  For information on other values for the `CodeFormat` TLC variable, see “Compare System Target File Support Across Products”.
- `Language`: The only valid value is `C`, which enables support for `C` or `C++` code generation as specified by the configuration parameter `TargetLang`.
- `TargetType`: The code generator defines the preprocessor symbols `RT` and `NRT` to distinguish simulation code from real-time code. These symbols are used in conditional compilation. The `TargetType` variable determines whether `RT` or `NRT` is defined.

  Most targets are intended to generate real-time code. They assign `TargetType` as follows.

  ```
  %assign TargetType = "RT"
  ```

  Some targets, such as the model reference simulation target, accelerated simulation target, RSim target, and S-function target, generate code for use in nonreal time only. Such targets assign `TargetType` as follows.

  ```
  %assign TargetType = "NRT"
  ```

TLC Program Entry Point and Related %includes

The code generation process normally begins with `codegenentry.tlc`. The system target file invokes `codegenentry.tlc` as follows.

```
%include "codegenentry.tlc"
```

Note `codegenentry.tlc` and the lower-level TLC files assume that `CodeFormat`, `TargetType`, and `Language` have been assigned. Set these variables before including `codegenentry.tlc`.

If you need to implement target-specific code generation features, you should include the TLC file `mytarget_genfiles.tlc` in your system target file. This file provides a mechanism for executing custom TLC code before and after invoking `codegenentry.tlc`. For information on this mechanism, see

- "Using mytarget_genfiles.tlc" on page 94-31 for an example of custom TLC code for execution after the main code generation entry point.
- "Target Development and the Build Process" on page 94-14 for general information on the build process, and for information on other build process customization hooks.

Another way to customize the code generation process is to call lower-level functions (normally invoked by `codegenentry.tlc`) directly, and include your own TLC functions at each stage of the process. This approach should be taken with caution. See "TLC Files" for more information.

The lower-level functions called by `codegenentry.tlc` are

- `genmap.tlc`: maps block names to corresponding language-specific block target files.
- `commonsetup.tlc`: sets up global variables.
- `commonentry.tlc`: starts the process of generating code.

RTW_OPTIONS Section

The `RTW_OPTIONS` section is bounded by the directives:

```
/%
  BEGIN_RTW_OPTIONS
.
.
.
  END_RTW_OPTIONS
%/
```

The first part of the `RTW_OPTIONS` section defines an array of `rtwoptions` structures. This structure is discussed in "Using rtwoptions to Display Custom Target Options" on page 94-25.

The second part of the `RTW_OPTIONS` section defines `rtwgensettings`, a structure defining the build folder name and other settings for the code generation process. See "rtwgensettings Structure" on page 94-23 for information about `rtwgensettings`.

rtwgensettings Structure

The final part of the system target file defines the `rtwgensettings` structure. This structure stores information that is written to the *`model`*`.rtw` file and used by the build process. The `rtwgensettings` fields of most interest to target developers are

- `rtwgensettings.Version`: Use this property to enable `rtwoptions` callbacks and to use the Callback API in `rtwgensettings.SelectCallback`.

Note To use callbacks, you *must* set:

```
rtwgensettings.Version = '1';
```

Add the statement above to the **Configure RTW code generation settings** section of the system target file.

- `rtwgensettings.DerivedFrom`: This structure field defines the system target file from which options are to be inherited. See "Inheriting Target Options" on page 94-30.
- `rtwgensettings.SelectCallback`: This structure field specifies a `SelectCallback` function. You must set `rtwgensettings.Version = '1';` or your callback will be ignored. `SelectCallback` is associated with the target rather than with any of its individual options. The `SelectCallback` function is triggered when you:

  - Select a target by setting the **System target file** configuration parameter.
  - Build the model.

  The `SelectCallback` function is useful for setting up (or disabling) configuration parameters specific to the target.

  The following code installs a `SelectCallback` function:

  ```
  rtwgensettings.SelectCallback = 'my_select_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `SelectCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions.

  Note If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See "Support Model Referencing" on page 94-61.

- `rtwgensettings.ActivateCallback`: this property specifies an `ActivateCallback` function. The `ActivateCallback` function is triggered when the active configuration set of the model changes. This could happen during model loading, and also when the user changes the active configuration set.

  The following code installs an `ActivateCallback` function:

  ```
  rtwgensettings.ActivateCallback = 'my_activate_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `ActivateCallback` function `(hDlg,hSrc)` are handles to private data used by the callback API functions.
- `rtwgensettings.PostApplyCallback`: this property specifies a `PostApplyCallback` function. The `PostApplyCallback` function is triggered when the user clicks the **Apply** or **OK** button after editing options in the Configuration Parameters dialog box. The `PostApplyCallback` function is called after the changes have been applied to the configuration set.

  The following code installs an `PostApplyCallback` function:

  ```
  rtwgensettings.PostApplyCallback = 'my_postapply_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `PostApplyCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions.

- `rtwgensettings.BuildDirSuffix`: To identify the build folder created by the build process, most system target files define a folder name suffix. To form the name of the build folder, the build process appends the suffix defined in the `rtwgensettings.BuildDirSuffix` field to the model name. You can modify the suffix by specifying a new string value for `rtwgensettings.BuildDirSuffix`. For example:

  ```
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  ```

  The build process creates a build folder named *`model`*`_mytarget_rtw`. For more information, see “Build Process Folders” on page 52-31.

  Do *not* use a function to specify the new string value. For example:

  ```
  rtwgensettings.BuildDirSuffix = my_get_build_suffix(bdroot);
  ```

Additional Code Generation Options

“Configure Generated Code with TLC” describes additional TLC code generation variables. End users of a target can assign these variables by entering a MATLAB command of the form

```
set_param(modelName,'TLCOptions','-aVariable=val');
```

(For more information, see “Specify TLC for Code Generation”.)

However, the preferred approach is to assign these variables in the system target file using statements of the form:

```
%assign Variable = val
```

For readability, we recommend that you add such assignments in the section of the system target file after the comment **Configure RTW code generation settings**.

Model Reference Considerations

See “Support Model Referencing” on page 94-61 for important information on system target file and other modifications you may need to make to support the code generator model referencing features.

Define and Display Custom Target Options

- “Using rtwoptions to Display Custom Target Options” on page 94-25
- “Example System Target File With Customized rtwoptions” on page 94-29
- “Inheriting Target Options” on page 94-30

Using rtwoptions to Display Custom Target Options

You control the options to display in the **Code Generation** pane in the Configuration Parameters dialog box by customizing the `rtwoptions` structure in your system target file.

The fields of the `rtwoptions` structure define variables and associated user interface elements to be displayed in the Configuration Parameters dialog box. Using the `rtwoptions` structure array, you can define target-specific options displayed in the dialog box and organize options into categories. You can also write callback functions to specify how these options are processed.

When the **Code Generation** pane opens, the `rtwoptions` structure array is scanned and the listed options are displayed. Each option is represented by an assigned user interface element (check box, edit field, menu, or push button), which displays the current option value.

The user interface elements can be in an enabled or disabled (appears dimmed) state. If an option is enabled, the user can change the option value. If an option is disabled, the option uses the default value and the user cannot change the option value.

You can also use the `rtwoptions` structure array to define special NonUI elements that cause callback functions to be executed, but that are not displayed in the **Code Generation** pane. See "NonUI Elements" on page 94-28.

The elements of the `rtwoptions` structure array are organized into groups. Each group of items begins with a header element of type `Category`. The default field of a `Category` header must contain a count of the remaining elements in the category.

The `Category` header is followed by options to be displayed on the **Code Generation** pane. The header in each category is followed by one or more option definition elements.

Each category of target options corresponds to options listed under **Code Generation** in the Configuration Parameters dialog box.

The table rtwoptions Structure Fields Summary summarizes the fields of the `rtwoptions` structure.

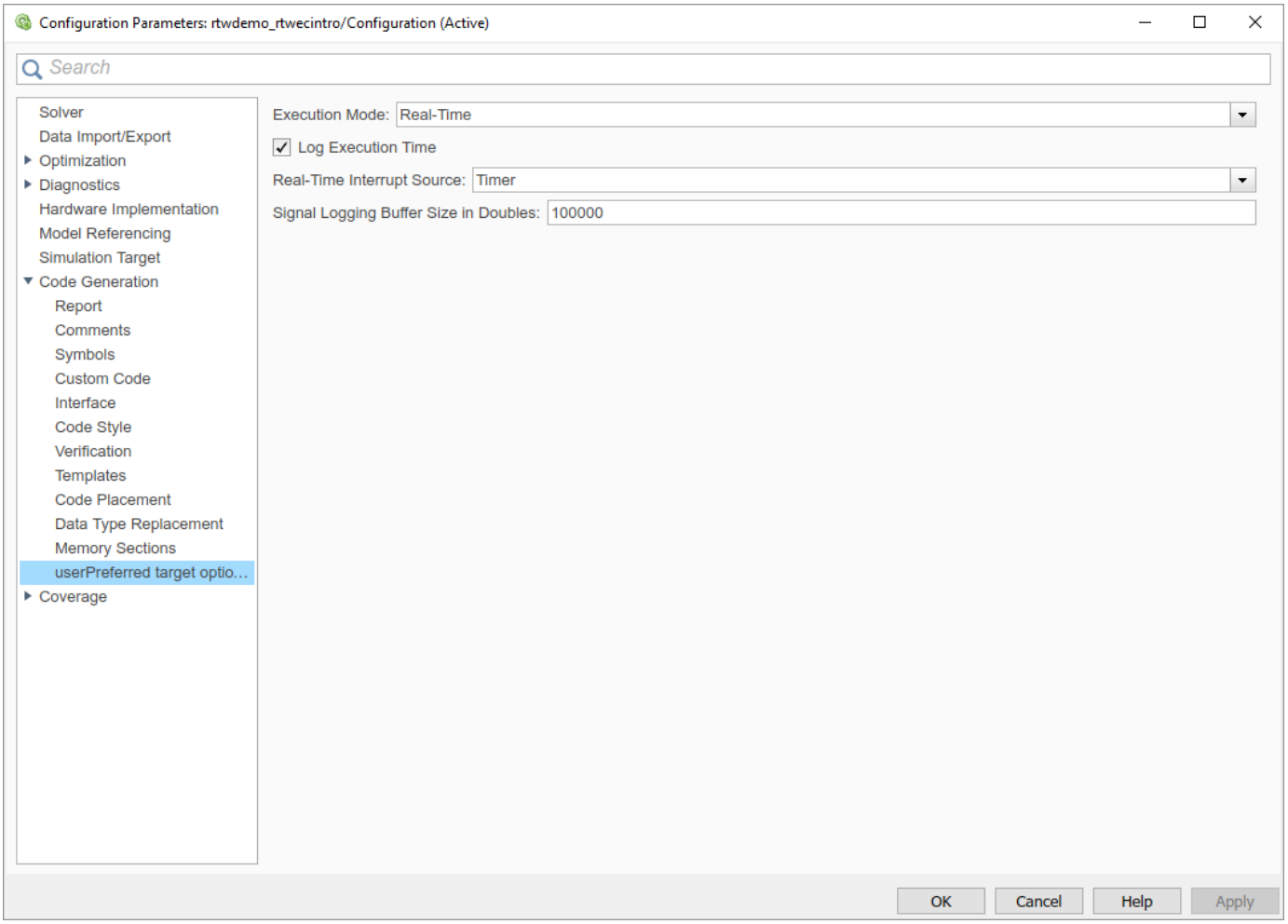
Example rtwoptions Structure

The following `rtwoptions` structure is excerpted from an example system target file, *`matlabroot/`*`toolbox/rtw/rtwdemos/rtwoptions_demo/usertarget.tlc`. The code defines an `rtwoptions` structure array. The default field of the first (header) element is set to `4`, indicating the number of elements that follow the header.

```
  rtwoptions(1).prompt        = 'userPreferred target options (I)';
  rtwoptions(1).type          = 'Category';
  rtwoptions(1).enable        = 'on';
  rtwoptions(1).default       = 4;   % number of items under this category
                                     % excluding this one.
  rtwoptions(1).popupstrings  = '';  % At the first item, user has to
  rtwoptions(1).tlcvariable   = '';  % initialize all supported fields
  rtwoptions(1).tooltip       = '';
  rtwoptions(1).callback      = '';
  rtwoptions(1).makevariable  = '';

  rtwoptions(2).prompt        = 'Execution Mode';
  rtwoptions(2).type          = 'Popup';
  rtwoptions(2).default       = 'Real-Time';
  rtwoptions(2).popupstrings  = 'Real-Time|UserDefined';
  rtwoptions(2).tlcvariable   = 'tlcvariable1';
  rtwoptions(2).tooltip       = ['See this text as tooltip'];

  rtwoptions(3).prompt        = 'Log Execution Time';
  rtwoptions(3).type          = 'Checkbox';
  rtwoptions(3).default       = 'on';
  rtwoptions(3).tlcvariable   = 'RL32LogTETModifier';
  rtwoptions(3).tooltip       = ['']; % no tooltip

  rtwoptions(4).prompt        = 'Real-Time Interrupt Source';
  rtwoptions(4).type          = 'Popup';
  rtwoptions(4).default       = 'Timer';
  rtwoptions(4).popupstrings  = 'Timer|5|6|7|8|9|10|11|12|13|14|15';
  rtwoptions(4).tlcvariable   = 'tlcvariable3';
  rtwoptions(4).callback      = 'usertargetcallback(hDlg, hSrc, ''tlcvariable3'')';
  rtwoptions(4).tooltip       = [''];
  rtwoptions(4).tooltip       = ['See TLC file for how to use reserved '...
    ' keyword ''hDlg'', and ''hSrc''.'];
...
  rtwoptions(5).prompt        = 'Signal Logging Buffer Size in Doubles';
  rtwoptions(5).type          = 'Edit';
  rtwoptions(5).default       = '100000';
```

```
rtwoptions(5).tlcvariable   = 'tlcvariable2';
rtwoptions(5).tooltip       = [''];
```

The first element adds a **userPreferred target options (I)** pane under **Code Generation** in the Configuration Parameters dialog box. The pane displays the options defined in `rtwoptions(2)`, `rtwoptions(3)`, `rtwoptions(4)`, and `rtwoptions(5)`.



If you want to define a large number of options, you can define multiple `Category` groups within a single system target file.

Note the `rtwoptions` structure and callbacks are written in MATLAB code, although they are embedded in a TLC file. To verify the syntax of your `rtwoptions` structure definitions and code, you can execute the commands at the MATLAB prompt by copying and pasting them to the MATLAB Command Window.

To learn more about `usertarget.tlc` and the example callback files provided with it, see "Example System Target File With Customized rtwoptions" on page 94-29. For more examples of target-specific `rtwoptions` definitions, see the *`target`*`.tlc` files under *`matlabroot`*`/rtw/c` (open).

rtwoptions Structure Fields Summary lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
`callback`	For examples of callback usage, see “Example System Target File With Customized rtwoptions” on page 94-29.
`closecallback` (obsolete)	Do not use `closecallback`. Use `rtwgensettings.PostApplyCallback` instead (see “rtwgensettings Structure” on page 94-23). `closecallback` is ignored. For examples of callback usage, see “Example System Target File With Customized rtwoptions” on page 94-29.
`default`	Default value of the option (empty if the `type` is `Pushbutton`).
`enable`	Must be `'on'` or `'off'`. If `'on'`, the option is displayed as an enabled item; otherwise, as a disabled item.
`makevariable`	Template makefile token (if any) associated with the option. The `makevariable` is expanded during processing of the template makefile. See “Template Makefile Tokens” on page 94-44.
`modelReferenceParameter-Check`	Specifies whether the option must have the same value in a referenced model and its parent model. If this field is unspecified or has the value `'on'` the option values must be same. If the field is specified and has the value `'off'` the option values can differ. See “Controlling Configuration Option Value Agreement” on page 94-63.
`NonUI`	Element that is not displayed, but is used to invoke a close or open callback. See “NonUI Elements” on page 94-28.
`opencallback` (obsolete)	Do not use `opencallback`. Use `rtwgensettings.SelectCallback` instead (see “rtwgensettings Structure” on page 94-23). For examples of callback usage, see “Example System Target File With Customized rtwoptions” on page 94-29.
`popupstrings`	If `type` is `Popup`, `popupstrings` defines the items in the menu. Items are delimited by the "|" (vertical bar) character. The following example defines the items of the **MAT-file variable name modifier** menu used by the GRT target. `'rt_|_rt|none'`
`prompt`	Label for the option.
`tlcvariable`	Name of TLC variable associated with the option.
`tooltip`	Help text displayed when mouse is over the item.
`type`	Type of element: `Checkbox`, `Edit`, `NonUI`, `Popup`, `Pushbutton`, or `Category`.

NonUI Elements

Elements of the `rtwoptions` array that have type `NonUI` exist solely to invoke callbacks. A `NonUI` element is not displayed in the Configuration Parameters dialog box. You can use a `NonUI` element if you want to execute a callback that is not associated with a user interface element, when the dialog

box opens or closes. See the next section, “Example System Target File With Customized rtwoptions” on page 94-29 for an example.

Note The default value of an element of type `NonUI` or `Edit` determines the set of values allowed for that element.

- If the default value is `'0'` or `'1'`:
  - For type `NonUI`, the element stores a Boolean value.
  - For type `Edit`, the element stores a value of type `int32`.
- If the default value contains an integer other than `'0'` or `'1'`, the element stores a value of type `int32`.
- If the default value does not contain an integer, the element is evaluated as a character vector.

Example System Target File With Customized rtwoptions

A working system target file, with MATLAB file callback functions, has been provided as an example of how to use the `rtwoptions` structure to display and process custom options on the **Code Generation** pane. The examples are compatible with the callback API.

The example target files are in the folder (open):

matlabroot`/toolbox/rtw/rtwdemos/rtwoptions_demo`

The example target files include:

- `usertarget.tlc`: The example system target file. This file illustrates how to define custom menus, check boxes, and edit fields. The file also illustrates the use of callbacks.
- `usertargetcallback.m`: A MATLAB file callback invoked by a menu.

Refer to the example files while reading this section. The example system target file, `usertarget.tlc`: illustrates the use of `rtwoptions` to display the following custom target options:

- The **Execution Mode** menu.
- The **Log Execution Time** check box.
- The **Real-Time Interrupt Source** menu. The menu executes a callback defined in an external file, `usertargetcallback.m`. The TLC variable associated with the menu is passed in to the callback, which displays the menu's current value.
- The edit field **Signal Logging Buffer Size in Doubles**.

Try studying the example code while interacting with the example target options in the Configuration Parameters dialog box. To interact with the example target file,

1. Make *matlabroot*`/toolbox/rtw/rtwdemos/rtwoptions_demo` (open) your working folder.
2. Open a model of your choice.
3. Open the Configuration Parameters dialog box and select the **Code Generation** pane.
4. Click **Browse**. The System Target File Browser opens. Select `usertarget.tlc`. Then click **OK**.
5. Observe that the **Code Generation** pane contains a custom sub-tab: **userPreferred target options (I)**.

6 As you interact with the options in this category and open and close the Configuration Parameters dialog box, observe the messages displayed in the MATLAB Command Window. These messages are printed from code in the system target file, or from callbacks invoked from the system target file.

Inheriting Target Options

`ert.tlc` provides a basic set of Embedded Coder code generation options. If your target is based on `ert.tlc`, your system target file should normally inherit the options defined in ERT.

Use the `rtwgensettings.DerivedFrom` field in the `rtwgensettings` structure to define the system target file from which options are to be inherited. You should convert your custom target to use this mechanism as follows.

Set the `rtwgensettings.DerivedFrom` field value as in the following example:

```
rtwgensettings.DerivedFrom = 'stf.tlc';
```

where `stf` is the name of the system target file from which options are to be inherited. For example:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
```

When the Configuration Parameters dialog box executes this line of code, it includes the options from `stf.tlc` automatically. If `stf.tlc` is a MathWorks internal system target file that has been converted to a new layout, the dialog box displays the inherited options using the new layout.

Handling Unsupported Options

If your target does not support all of the options inherited from `ert.tlc`, you should detect unsupported option settings and display a warning or error message. In some cases, if a user has selected an option your target does not support, you may need to abort the build process. For example, if your target does not support the **Generate an example main program** option, the build process should not be allowed to proceed if that option is selected.

Even though your target may not support all inherited ERT options, it is required that the ERT options are retained in the **Code Generation** pane of the Configuration Parameters dialog box. Do not simply remove unsupported options from the `rtwoptions` structure in the system target file. Options must be in the dialog box to be scanned by the code generator when it performs optimizations.

For example, you may want to prevent users from turning off the **Single output/update function** option. It may seem reasonable to remove this option from the dialog box and simply assign the TLC variable `CombineOutputUpdateFcns` to `on`. However, if the option is not included in the dialog box, the code generator assumes that output and update functions are *not* to be combined. Less efficient code is generated as a result.

Tips and Techniques for Customizing Your System Target File

- “Introduction” on page 94-31
- “Required and Recommended %includes” on page 94-31
- “Handling Aliases for Target Option Values” on page 94-31
- “Supporting Multiple Development Environments” on page 94-33
- “Updating Your Custom System Target File” on page 94-33

Introduction

The following sections include information on techniques for customizing your system target file, including:

- How to invoke custom TLC code from your system target file
- Approaches to supporting multiple development environments
- Considerations for when you update your system target file

Required and Recommended %includes

If you need to implement target-specific code generation features, we recommend that your system target file include the TLC file `mytarget_genfiles.tlc`.

Once your system target file has set up the required TLC environment, you must include `codegenentry.tlc` to start the standard code generation process.

`mytarget_genfiles.tlc` provides a mechanism for executing custom TLC code after the main code generation entry point. See “Using mytarget_genfiles.tlc” on page 94-31.

Using mytarget_genfiles.tlc

`mytarget_genfiles.tlc` (optional) is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads.

The build process can then invoke these generated files either directly from the make process or after the executable is created. This is done with the *STF*`_make_rtw_hook.m` mechanism, as described in “Customize Build Process with STF_make_rtw_hook File”.

The following TLC code shows an example `mytarget_genfiles.tlc` file.

```
%selectfile NULL_FILE

%assign ModelName = CompiledModel.Name

%% Create Debugger script
%assign model_script_file = "%<ModelName>.cfg"
%assign script_file = "debugger_script_template.tlc"

%if RTWVerbose
   %selectfile STDOUT
   ### Creating %<model_script_file>
   %selectfile NULL_FILE
%endif

%include "%<script_file>"
%openfile bld_file = "%<model_script_file>"
%<CreateDebuggerScript()>
%closefile bld_file
```

Handling Aliases for Target Option Values

This section describes utility functions that can be used to detect and resolve alias values or legacy values when testing user-specified values for the target device type (`ProdHWDeviceType`) and the code replacement library (`CodeReplacementLibrary`).

RTW.isHWDeviceTypeEq

To test if two target device type values represent the same hardware device, invoke the following function:

```
result = RTW.isHWDeviceTypeEq(type1,type2)
```

where *type1* and *type2* are character vectors containing target device type values or aliases.

The `RTW.isHWDeviceTypeEq` function returns true if *type1* and *type2* are character vectors representing the same hardware device. For example, the following call returns true:

```
RTW.isHWDeviceTypeEq('Specified','Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters “Device vendor” and “Device type”.

RTW.resolveHWDeviceType

To return the device type value for a hardware device, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveHWDeviceType(type)
```

where *type* is a character vector containing a target device type value or alias.

The `RTW.resolveHWDeviceType` function returns the device type value of the device. For example, the following calls both return `'Generic->Custom'`:

```
RTW.resolveHWDeviceType('Specified')
RTW.resolveHWDeviceType('Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters “Device vendor” and “Device type”.

RTW.isTflEq

To test if two code replacement library (CRL) names represent the same CRL, invoke the following function:

```
result = RTW.isTflEq(name1,name2)
```

where *name1* and *name2* are character vectors containing CRL values or aliases.

The `RTW.isTflEq` function returns true if *name1* and *name2* are character vectors representing the same code replacement library. For example, the following call returns true:

```
RTW.isTflEq('GNU','GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see Code replacement library.

RTW.resolveTflName

To return the CRL value for a code replacement library, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveTflName(name)
```

where *name* is a character vector containing a CRL value or alias.

The `RTW.resolveTflName` function returns the value of the referenced code replacement library. For example, the following calls both return `'GNU C99 extensions'`:

```
RTW.resolveTflName('GNU')
RTW.resolveTflName('GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see Code replacement library.

Supporting Multiple Development Environments

Your target may require support for multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

One approach to this requirement is to implement multiple system target files. Each system target file invokes a template makefile for the development environment. This amounts to providing two separate targets.

Updating Your Custom System Target File

Updating your custom system target file can impact the option values of a model that you load that uses the updated system target file. If you disable an option, the updated system target file uses the default value for that option. If a model has a different value for that option, when you load the model with the updated system target file, the value from the model is discarded and the system target file uses the default value instead.

Create a Custom Target Configuration

- “Introduction” on page 94-33
- “my_ert_target Overview” on page 94-34
- “Creating Target Folders” on page 94-35
- “Create ERT-Based, Toolchain Compliant System Target File” on page 94-35
- “Create ERT-Based Template MakeFile” on page 94-40
- “Create Test Model and S-Function” on page 94-41
- “Verify Target Operation” on page 94-42

Introduction

This tutorial can supplement the example target guides described in “Sample Custom Targets” on page 94-4. For an introduction and example files, try the example targets first.

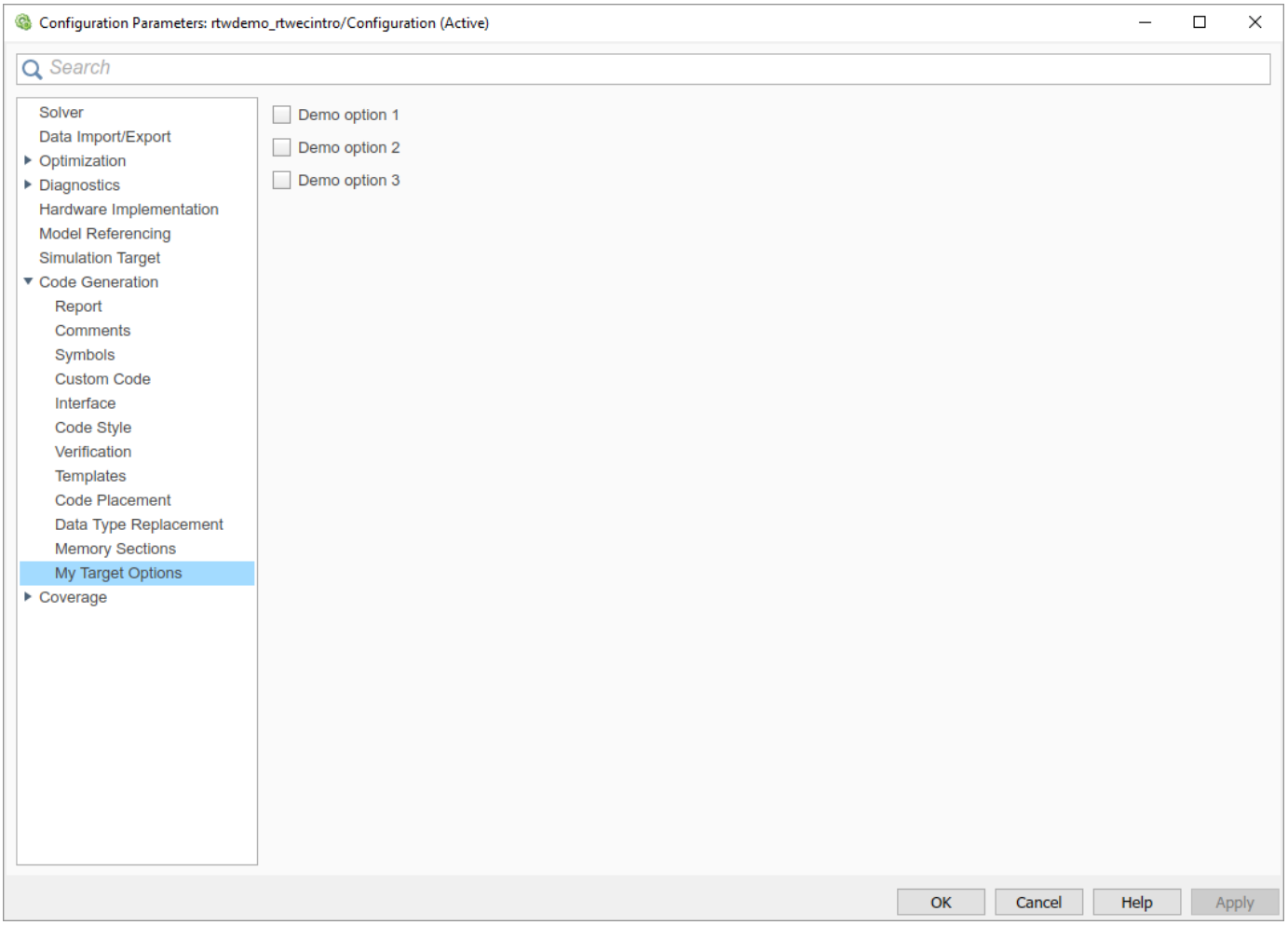
This tutorial guided you through the process of creating an ERT-based target, `my_ert_target`. This exercise illustrates several tasks, which are typical for creating a custom target:

- Setting up target folders and modifying the MATLAB path.
- Making modifications to a standard system target file and TMF such that the custom target is visible in the System Target File Browser, inherits ERT options, displays target-specific options, and generates code with the default host-based compiler.
- Testing the build process with the custom target, using a simple model that incorporates an inlined S-function.

During this exercise, you implement an operational, but skeletal, ERT-based target. This target can be useful as a starting point in a complete implementation of a custom embedded target.

my_ert_target Overview

In the following sections, you create a skeletal target, `my_ert_target`. The target inherits and supports the standard options of the ERT target and displays additional target-specific options in the Configuration Parameters dialog box (see "Target-Specific Options for my_ert_target" on page 94-34).



Target-Specific Options for my_ert_target

`my_ert_target` supports a toolchain-based build, generating code and executables that run on the host system. `my_ert_target` uses the `lcc` compiler on a Microsoft Windows platform. The chosen compiler is readily available and is distributed with the code generator. On a Microsoft Windows platform, if you use a different compiler, you can set up `lcc` temporarily as your default compiler through the following MATLAB command:

```
mex -setup
```

The software displays links for supported compilers that are installed on your computer. Click the `lcc` link.

Note On Linux systems, make sure that you have an installed C compiler. If so, you can use Linux folder syntax to complete this exercise.

`my_ert_target` can also support template makefile-based builds. For more information about using this target with the template makefile approach, see “Create ERT-Based Template MakeFile” on page 94-40.

You can test `my_ert_target` with a model that is compatible with the ERT target (see “Configure a System Target File” on page 50-2). Generated programs operate identically to ERT generated programs.

To simplify the testing of your target, test with `targetmodel`, a very simple fixed-step model (see “Create Test Model and S-Function” on page 94-41). The S-Function block in `targetmodel` uses the source code from the `timestwo` example, and generates fully inlined code. See “S-Function Examples” and “Inline S-Functions with TLC” for further discussion of the `timestwo` example S-function.

Creating Target Folders

Create folders to store the target files and add them to the MATLAB path, following the recommended conventions (see “Folder and File Naming Conventions” on page 94-6). You also create a folder to store the test model, S-function, and generated code.

This example assumes that your target and model folders are located within the folder `c:/work`. Do not place your target and model folders within the MATLAB folder tree (that is, in or under the *`matlabroot`* folder).

To create the folders and make them accessible:

1. Create a target root folder, `my_ert_target`. From the MATLAB Command Window on a Windows platform, enter:

   ```
   cd c:/work
   mkdir my_ert_target
   ```

2. Within the target root folder, create a subfolder to store your target files.

   ```
   mkdir my_ert_target/my_ert_target
   ```

3. Add these folders to your MATLAB path.

   ```
   addpath c:/work/my_ert_target
   addpath c:/work/my_ert_target/my_ert_target
   ```

4. Create a folder, `my_targetmodel`, to store the test model, S-function, and generated code.

   ```
   mkdir my_targetmodel
   ```

Create ERT-Based, Toolchain Compliant System Target File

Create a system target file for your target by copying and modifying the standard system target file for the ERT target. Then, validate the system target file by viewing the new target in the System Target File Browser and in the Configuration Parameters dialog box.

Editing the System Target File

To edit the system target file, use these steps:

1. Change your working folder to the folder you created in “Creating Target Folders” on page 94-35.

```
cd c:/work/my_ert_target/my_ert_target
```

2. Place a copy of *matlabroot*/rtw/c/ert/ert.tlc in c:/work/my_ert_target/my_ert_target and rename it to my_ert_target.tlc. The file ert.tlc is the system target file for the ERT target.
3. Open my_ert_target.tlc in a text editor of your choice.
4. Customize the system target file, replacing the header comment lines with directives that make your system target file visible in the System Target File Browser and define the associated TMF, make command, and external mode interface file (if any). For more information about these directives, see "Header Comments" on page 94-21 .

   Replace the header comments in my_ert_target.tlc with the following header comments.

```
%% SYSTLC: My ERT-based Target TMF: my_ert_target_lcc.tmf MAKE: make_rtw
```

5. The file my_ert_target.tlc inherits the standard ERT options, using the mechanism described in "Inheriting Target Options" on page 94-30. Therefore, the existing rtwoptions structure definition is superfluous. Edit the RTW_OPTIONS section such that it includes only the following code.

```
/%
  BEGIN_RTW_OPTIONS

  %----------------------------------------%
  % Configure RTW code generation settings %
  %----------------------------------------%

  rtwgensettings.BuildDirSuffix = '_ert_rtw';

  END_RTW_OPTIONS
 %/
```

6. Delete the code after the end of the RTW_OPTIONS section, which is delimited by the directives BEGIN_CONFIGSET_TARGET_COMPONENT and END_CONFIGSET_TARGET_COMPONENT. This code is for use only by internal MathWorks developers.
7. Modify the build folder suffix in the rtwgenSettings structure in accordance with the conventions described in "rtwgensettings Structure" on page 94-23.

   To set the suffix to a character vector for the _my_ert_target custom target, change the line

```
rtwgensettings.BuildDirSuffix = '_ert_rtw'
```

   to

```
rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw'
```

8. Modify the rtwgenSettings structure to inherit options from the ERT target and declare Release 14 or later compatibility as described in "rtwgensettings Structure" on page 94-23. Add the following code to the rtwgenSettings definition:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
rtwgensettings.Version = '1';
```

9. Add an rtwoptions structure that defines a target-specific options category with three check boxes just after the BEGIN_RTW_OPTIONS directive. The following code shows the complete RTW_OPTIONS section, including the previous rtwgenSettings changes.

```
/%
  BEGIN_RTW_OPTIONS

  rtwoptions(1).prompt         = 'My Target Options';
  rtwoptions(1).type           = 'Category';
  rtwoptions(1).enable         = 'on';
  rtwoptions(1).default        = 3;   % number of items under this category
```

```
                                        % excluding this one.
    rtwoptions(1).popupstrings  = '';
    rtwoptions(1).tlcvariable   = '';
    rtwoptions(1).tooltip       = '';
    rtwoptions(1).callback      = '';
    rtwoptions(1).makevariable  = '';

    rtwoptions(2).prompt         = 'Demo option 1';
    rtwoptions(2).type           = 'Checkbox';
    rtwoptions(2).default        = 'off';
    rtwoptions(2).tlcvariable    = 'DummyOpt1';
    rtwoptions(2).makevariable   = '';
    rtwoptions(2).tooltip        = ['Demo option1 (non-functional)'];
    rtwoptions(2).callback       = '';

    rtwoptions(3).prompt         = 'Demo option 2';
    rtwoptions(3).type           = 'Checkbox';
    rtwoptions(3).default        = 'off';
    rtwoptions(3).tlcvariable    = 'DummyOpt2';
    rtwoptions(3).makevariable   = '';
    rtwoptions(3).tooltip        = ['Demo option2 (non-functional)'];
    rtwoptions(3).callback       = '';

    rtwoptions(4).prompt         = 'Demo option 3';
    rtwoptions(4).type           = 'Checkbox';
    rtwoptions(4).default        = 'off';
    rtwoptions(4).tlcvariable    = 'DummyOpt3';
    rtwoptions(4).makevariable   = '';
    rtwoptions(4).tooltip        = ['Demo option3 (non-functional)'];
    rtwoptions(4).callback       = '';

    %----------------------------------------%
    % Configure RTW code generation settings %
    %----------------------------------------%

    rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw';
    rtwgensettings.DerivedFrom = 'ert.tlc';
    rtwgensettings.Version = '1';
    rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
    %SelectCallback provides toolchain approach support, but requires custom function
    %Omit this SelectCallback if using the template makefile approach

    END_RTW_OPTIONS
  %/
```

10 Save your changes to `my_ert_target.tlc` and close the file.

Create ToolchainCompliant Function

To enable builds using the toolchain approach, you create a function that corresponds to the `SelectCallback` near the end of the custom system target file. This function sets properties for toolchain compliance.

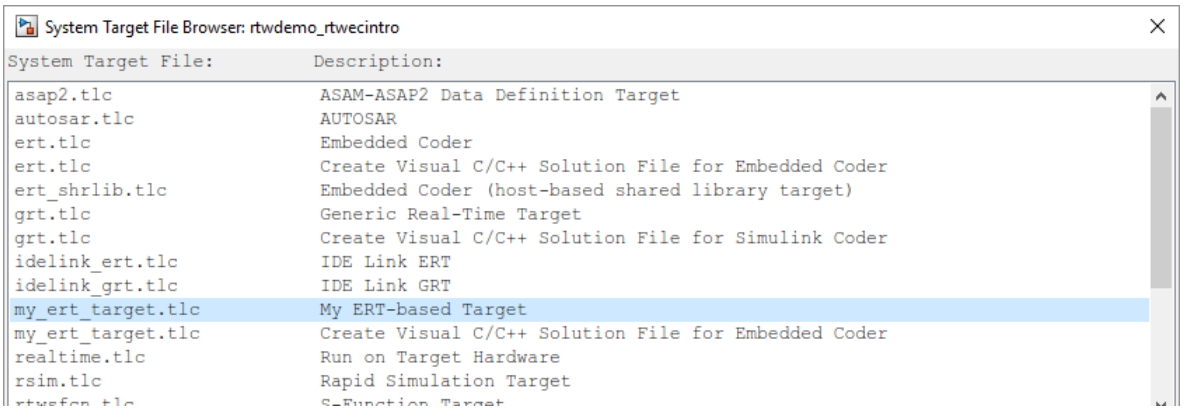
```
function enableToolchainCompliant(hSrc, hDlg)
  % The following parameters enable toolchain compliance.
  slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
  slConfigUISetVal(hDlg, hSrc, 'GenerateMakefile','on');

  % The following parameters are not required for toolchain compliance.
  % But, it is recommended practice to set these default values and
  % disable the parameters (as shown).
  slConfigUISetVal(hDlg, hSrc, 'RTWCompilerOptimization','off');
  slConfigUISetVal(hDlg, hSrc, 'MakeCommand','make_rtw');
  slConfigUISetEnabled(hDlg, hSrc, 'RTWCompilerOptimization',false);
  slConfigUISetEnabled(hDlg, hSrc, 'MakeCommand',false);
  hCS = hSrc.getConfigSet();
  hCS.refreshDialog;
end
```

Note If you are using the template makefile approach, omit calling the function enabling toolchain-compliance from your system target file. Instead, use the information in “Create ERT-Based Template MakeFile” on page 94-40.

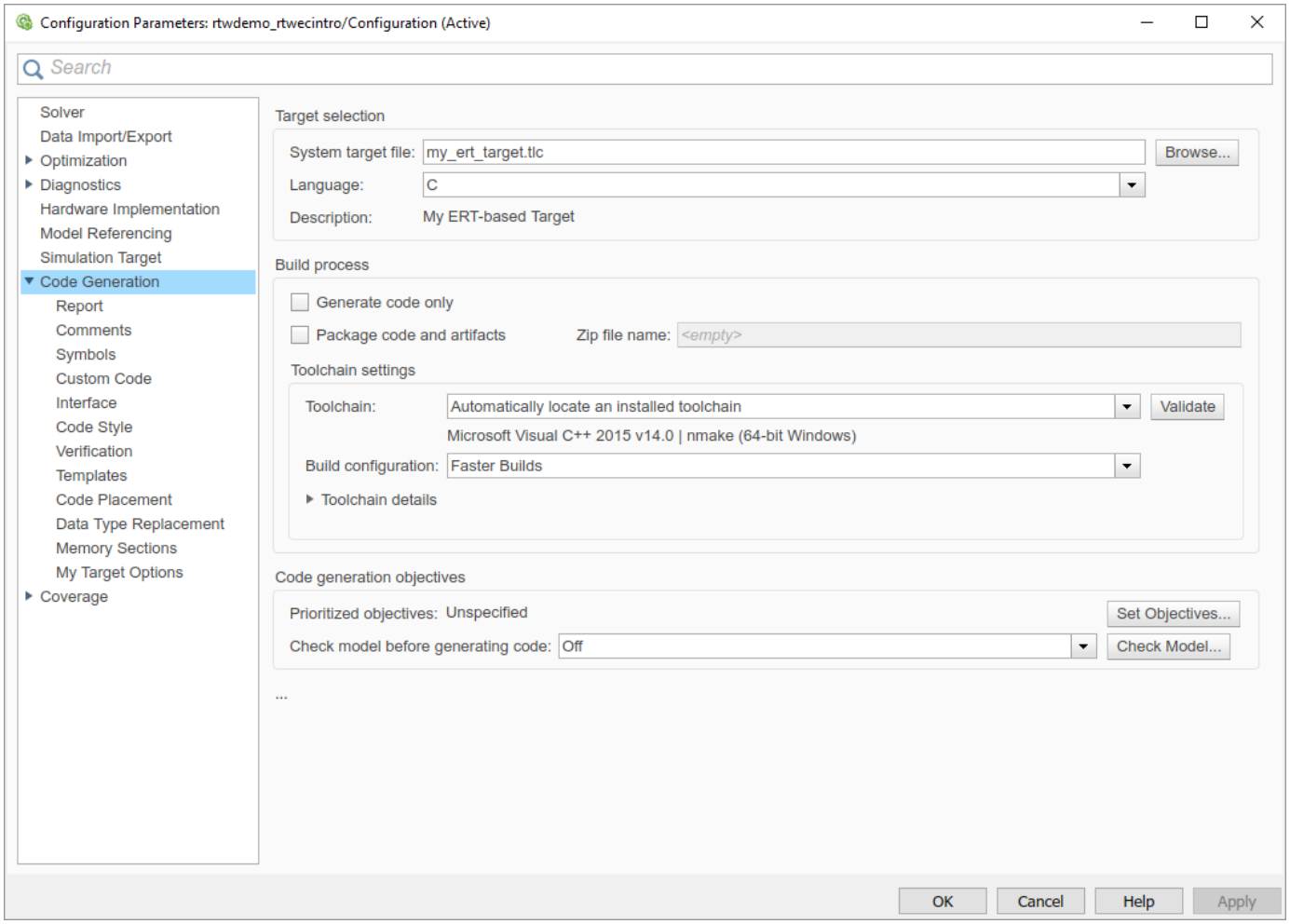
Viewing the System Target File

At this point, you can verify that the target inherits and displays ERT options as follows:

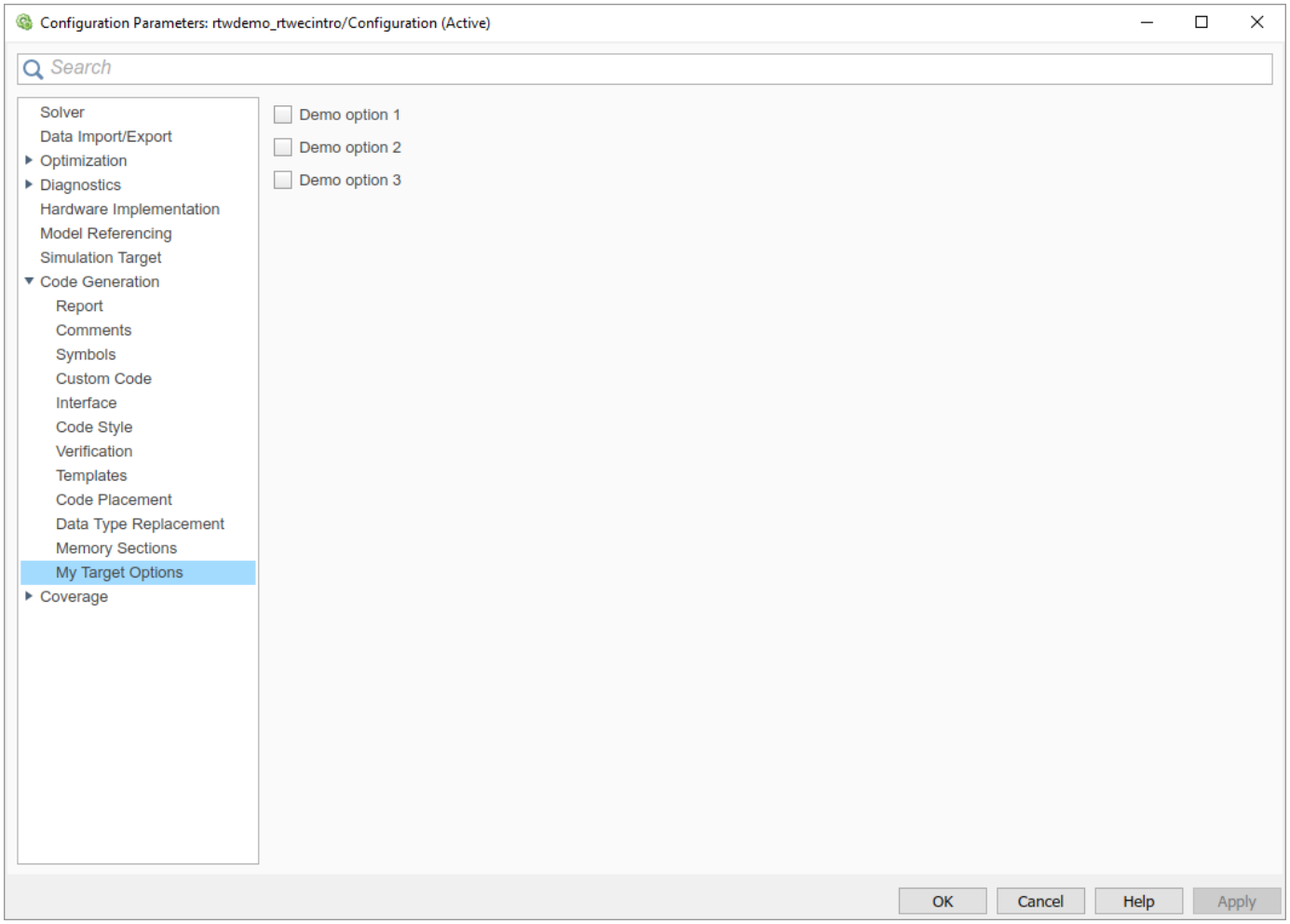
1. Create a new model.
2. Open the Model Explorer or the Configuration Parameters dialog box.
3. Select the **Code Generation** pane.
4. Click **Browse** to open the System Target File browser.
5. In the file browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in “Creating Target Folders” on page 94-35.)
6. Select `My ERT-based Target` and click **OK.**



7. The **Code Generation** pane now shows that the model is configured for the `my_ert_target.tlc` target. The **System target file**, **Language**, **Toolchain**, and **Build configuration** fields should appear:



8 Select the **My Target Options** pane. The target displays the three check box options defined in the `rtwoptions` structure.



9 Select the **Code Generation** pane and reopen the System Target File Browser.

10 Select the Embedded Coder target (`ert.tlc`). The target displays the standard ERT options.

11 Close the model. You do not need to save it.

The system target file for the skeletal target is complete. If you are using the toolchain approach, you are ready to invoke the build process for your target.

If you prefer to use the template makefile approach, the reference to a TMF, `my_ert_target_lcc.tmf`, in the system target file header comments prevents you from invoking the build process for your target until the TMF file is in place. First, you must create a `my_ert_target_lcc.tmf` file.

Create ERT-Based Template MakeFile

If you are using the toolchain makefile approach with a toolchain compliant custom target, omit the steps that apply to the template makefile approach. (Skip this section.)

If you are using the template makefile approach, follow the steps that apply to a template makefile and omit calling the function that enables toolchain-compliance from your system target file, which is described in “Create ERT-Based, Toolchain Compliant System Target File” on page 94-35.

Create a template makefile for your target by copying and modifying a supplied ERT template makefile. Select the template makefile that fits your target best. This example uses `ert_lcc64.tmf`, but you can also use `ert_vcx64.tmf` or `ert_unix.tmf`.

1. Make sure that your working folder is still set to the target file folder you created previously in "Creating Target Folders" on page 94-35.

   ```
   c:/work/my_ert_target/my_ert_target
   ```

2. Place a copy of *`matlabroot`*`/toolbox/coder/compile/tmf/ert_lcc64.tmf` in `c:/work/my_ert_target/my_ert_target` and rename it `my_ert_target_lcc.tmf`. The file `ert_lcc64.tmf` is the ERT compiler-specific template makefile for the LCC compiler.
3. Open `my_ert_target_lcc.tmf` in a text editor.
4. Change the `SYS_TARGET_FILE` parameter so that the file reference for your `.tlc` file is generated in the `make` file. Change the line

   ```
   SYS_TARGET_FILE = any
   ```

   to

   ```
   SYS_TARGET_FILE = my_ert_target.tlc
   ```

5. Save changes to `my_ert_target_lcc.tmf` and close the file.

Your target can now generate code and build a host-based executable. In the next sections, you create a test model and test the build process using `my_ert_target`.

Create Test Model and S-Function

In this section, you build a simple test model for later use in code generation:

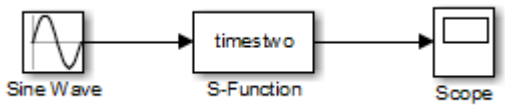
1. Set your working folder to `c:/work/my_targetmodel`.

   ```
   cd c:/work/my_targetmodel
   ```

   For the remainder of this tutorial, `my_targetmodel` is assumed to be the working folder. Your target writes the output files of the code generation process into a build folder within the working folder. When inlined code is generated for the `timestwo` S-function, the build process looks for the TLC implementation of the S-function in the working folder.
2. Copy the following C and TLC files for the `timestwo` S-function to your working folder:
   - *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/timestwo.c`
   - *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/timestwo.tlc`
3. Build the `timestwo` MEX-file in `c:/work/my_targetmodel`.

   ```
   mex timestwo.c
   ```

4. Create the following model, using an S-Function block from the Simulink User-Defined Functions library. Save the model in your working folder as `targetmodel`.



5. Double-click the S-Function block to open the Block Parameters dialog box. Enter the S-function name `timestwo`. Click **OK**. The block is now bound to the `timestwo` MEX-file.

6. Open Model Explorer or the Configuration Parameters dialog box and select the **Solver** pane.
7. Set the solver **Type** to `fixed-step` and click **Apply**.
8. Save the model.
9. Open the scope and run a simulation. Verify that the `timestwo` S-function multiplies its input by 2.0.

Keep the `targetmodel` model open for use in the next section, in which you generate code using the test model.

Verify Target Operation

In this section you configure `targetmodel` for the `my_ert_target` custom target, and use the target to generate code and build an executable:

1. In the Configuration Parameters dialog box, select the **Code Generation** pane.
2. Click **Browse** to open the System Target File Browser.
3. In the Browser, select `My ERT-based Target` and click **OK**.
4. The Configuration Parameters dialog box now displays the **Code Generation** pane for `my_ert_target`.
5. Select the **Code Generation** > **Report** pane and select the **Create code generation report** option.
6. Click **Apply** and save the model. The model is configured for `my_ert_target`.
7. Build the model. If the build succeeds, the MATLAB Command Window displays the message below.

   ```
   ### Created executable: ../targetmodel.exe
   ### Successful completion of build procedure for model:
   targetmodel
   ```

   Your working folder contains the `targetmodel.exe` file and the build folder, `targetmodel_my_ert_target_rtw`, which contains generated code and other files. The working folder also contains an `slprj` folder, used internally by the build process.

   The code generator also creates a code generation report.
8. To view the generated code, on the **C Code** tab, click **View Code**. In the **Contents** pane of the code generation report, click the `targetmodel.c` link.
9. In `targetmodel.c`, locate the model step function, `targetmodel_step`. Observe the following code.

   ```
   /* S-Function Block: <Root>/S-Function */
   /* Multiply input by two */
   targetmodel_B.SFunction = targetmodel_B.SineWave * 2.0;
   ```

   The presence of this code confirms that the `my_ert_target` custom target has generated an inlined output computation for the S-Function block in the model.

See Also

More About

- “Custom Targets”

- “Support Toolchain Approach with Custom Target”
- “Support Model Referencing”
- “Support Compiler Optimization Level Control”
- “Support C Function Prototype Control”
- “Support C++ Class Interface Control”
- “Support Concurrent Execution of Multiple Tasks”

Customize Template Makefiles

To configure or customize a template makefile (TMF), you should be familiar with how the `make` command works and how it processes a makefile (`.mk` file). You should also understand makefile build rules. For information on these topics, refer to the documentation provided with the make utility that you use.

Template Makefiles and Tokens

A template makefile contains tokens. The build process expands the tokens and creates makefiles:

- *`model`*`.mk` –– Compiles and links code generated from model components.
- `rtwshared.mk` –– Compiles generated shared utility code.

The makefiles (*`model_or_sharedutils`*`.mk`) use commands that are specific to your development computer.

Template Makefile Tokens

The `make_rtw` command (or a different command provided with some targets) directs the process of generating *`model_or_sharedutils`*`.mk`. The `make_rtw` command processes the TMF specified on the **Code Generation** pane of the Configuration Parameters dialog box. `make_rtw` copies the TMF, line by line, expanding each token encountered. Template Makefile Tokens Expanded by make_rtw lists the tokens and their expansions.

These tokens are used in several ways by the expanded makefile:

- To control the conditional behavior in the makefile. The conditionals are used to control the source file lists, library names, target to be built, and other build-related information.
- To provide the macro definitions for compiling the files, for example, `-DINTEGER_CODE=1`.

Template Makefile Tokens Expanded by make_rtw

Token	Expansion
General purpose	
`|>ADDITIONAL_LDFLAGS<|`	Linker flags automatically added by blocks.
`|>ALT_MATLAB_BIN<|`	Alternate full pathname for the MATLAB executable; value is different than value for `MATLAB_BIN` token when the full pathname contains spaces.
`|>ALT_MATLAB_ROOT<|`	Alternate full pathname for the MATLAB installation; value is different than value for `MATLAB_ROOT` token when the full pathname contains spaces.
`|>BUILDARGS<|`	Options passed to `make_rtw`. This token is provided so that the contents of your *`model_or_sharedutils`*`.mk` file change when you change the build arguments, thus forcing an update of modules when your build options change.
`|>COMBINE_OUTPUT_UPDATE_FCNS<|`	True (1) when you select model configuration parameter **Single output/update function**, otherwise False (0). Used for the macro definition `-DONESTEPFCN=1`.
`|>COMPILE_FLAGS_OTHER<|`	Compiler flags in a group other than `'OPTS'`, `'OPT_OPTS'`, and `'OPTIMIZATION_FLAGS'`. The build process produces a warning if `|>COMPILE_FLAGS_OTHER<|` is absent and these conditions apply: • The build information contains a compiler flag that is not in `'OPTS'`, `'OPT_OPTS'`, or `'OPTIMIZATION_FLAGS'`. • The compiler flag is not in the template makefile.
`|>COMPUTER<|`	Computer type. See the MATLAB `computer` command.

Token	Expansion

|>DEFINES_OTHER<|	Preprocessor macro definitions in a group other than 'OPTS', 'OPT_OPTS', 'OPTIMIZATION_FLAGS', and 'Custom'.
	
	A preprocessor macro definition in 'OPTS', 'OPT_OPTS', 'OPTIMIZATION_FLAGS', or 'Custom' that has a corresponding variable/token pair in the template makefile is not expanded in DEFINES_OTHER. This behavior helps to avoid the provision of duplicate preprocessor macro definitions.
	
	For example, some template makefiles contain these statements:
	
	`...`
	`NUMST = |>NUMST<|`
	`...`
	`CPP_REQ_DEFINES1 = ... -DNUMST=$(NUMST) ...`
	
	If the template makefile contains the variable/token pair NUMST = |>NUMST<| and NUMST is a preprocessor macro definition in buildInfo, then the build process does not expand it in DEFINES_OTHER.
	
	The build process produces a warning if |>DEFINES_OTHER<| is absent and these conditions apply:
	
	• The build information contains a preprocessor macro definition that is not in 'OPTS', 'OPT_OPTS', 'OPTIMIZATION_FLAGS', or 'Custom'.
	• The preprocessor macro definition, for example, NUMST, is not in the template makefile.

|>EXPAND_LIBRARY_LOCATION<|	Location of precompiled library file. The TargetPreCompLibLocation configuration parameter can override this setting. For examples, see “Control Library Location and Naming During Build”.
|>EXPAND_LIBRARY_NAME<|	Library name. For examples, see “Control Library Location and Naming During Build” and “Modify the Template Makefile for rtwmakecfg”.
|>EXPAND_LIBRARY_SUFFIX<|	Library suffix. The TargetLibSuffix configuration parameter can override this setting. For examples, see “Control Library Location and Naming During Build”.
|>EXT_MODE<| (Not required for R2018a and later provided template makefile specifies TOOLCHAIN_NAME)	True (1) to enable generation of external mode support code, otherwise False (0).
|>EXTMODE_TRANSPORT<| (Not required for R2018a and later provided template makefile specifies TOOLCHAIN_NAME)	Index of transport mechanism (for example, tcpip, serial) for external mode.

Token	Expansion
`|>EXTMODE_STATIC<|` (Not required for R2018a and later provided template makefile specifies `TOOLCHAIN_NAME`)	True (1) if static memory allocation is selected for external mode. False (0) if dynamic memory allocation is selected.
`|>EXTMODE_STATIC_SIZE<|` (Not required for R2018a and later provided template makefile specifies `TOOLCHAIN_NAME`)	Size of static memory allocation buffer for external mode.
`|>GENERATE_ERT_S_FUNCTION<|`	True (1) when model configuration parameter **Create block** is set to `SIL`, otherwise False (0). Used for control of the makefile target of the build.
`|>INCLUDE_MDL_TERMINATE_FCN<|`	True (1) when model configuration parameter **Terminate function required** is selected, otherwise False (0). Used for the macro definition `-DTERMFCN==1`.
`|>INTEGER_CODE<|`	True (1) when model configuration parameter **Support floating-point numbers** is cleared, otherwise False (0). `INTEGER_CODE` is a required macro definition when compiling the source code and is used when selecting precompiled libraries to link against.
`|>MAKEFILE_NAME<|`	*`model_or_sharedutils`*`.mk` — The name of the makefile that was created from the TMF.
`|>MAT_FILE<|`	True (1) when model configuration parameter **MAT-file logging** is selected, otherwise False (0). `MAT_FILE` is a required macro definition when compiling the source code and also is used to include logging code in the build process.
`|>MATLAB_BIN<|`	Location of the MATLAB executable program.
`|>MATLAB_ROOT<|`	Path to where MATLAB is installed.
`|>MEM_ALLOC<|`	`RT_MALLOC` or `RT_STATIC`. Indicates how memory is to be allocated.
`|>MEXEXT<|`	MEX-file extension. See the MATLAB `mexext` command.
`|>MODEL_MODULES<|`	Additional generated source modules. For example, you can split a large model into two files, *`model`*`.c` and *`model`*`1.c`. In this case, this token expands to *`model`*`1.c`.
`|>MODEL_MODULES_OBJ<|`	Object filenames (`.obj`) corresponding to additional generated source modules.
`|>MODEL_NAME<|`	Name of the Simulink block diagram currently being built.
`|>MULTITASKING<|`	True (1) if solver mode is multitasking, otherwise False (0).
`|>NCSTATES<|`	Number of continuous states.
`|>NUMST<|`	Number of sample times in the model.
`|>RELEASE_VERSION<|`	The MATLAB release version.
`|>S_FUNCTIONS_LIB<|`	List of S-function libraries available for linking.
`|>SOLVER<|`	Solver source filename, for example, `ode3.c`.
`|>SOLVER_OBJ<|`	Solver object (`.obj`) filename, for example, `ode3.obj`.
`|>START_DIR<|`	Code generation folder at the start of the build.

Token	Expansion
`|>TARGET_LANG_EXT<|`	`c` when model configuration parameter **Language** is set to `C`, `cpp` when **Language** is set to `C++`. Used in the makefile to control the extension on generated source files.
`|>TID01EQ<|`	True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).
S-function and build information support **Note** For examples of the tokens in this section, see "Modify the Template Makefile for rtwmakecfg".	

`|>START_EXPAND_INCLUDES<|`
`|>EXPAND_DIR_NAME<|`
`|>END_EXPAND_INCLUDES<|`
	List of folder names to add to the include path. Additionally, the `ADD_INCLUDES` macro must be added to the `INCLUDES` line.

`|>START_EXPAND_LIBRARIES<|`
`|>EXPAND_LIBRARY_NAME<|`
`|>END_EXPAND_LIBRARIES<|`
	List of library names.

`|>START_EXPAND_MODULES<|`
`|>EXPAND_MODULE_NAME<|`
`|>END_EXPAND_MODULES<|`
	Library module names within `|>START_EXPAND_LIBRARIES<|` and `|>START_PRECOMP_LIBRARIES<|` library lists.

`|>START_EXPAND_RULES<|`
`|>EXPAND_DIR_NAME<|`
`|>END_EXPAND_RULES<|`
	Makefile rules.

`|>START_PRECOMP_LIBRARIES<|`
`|>EXPAND_LIBRARY_NAME<|`
`|>END_PRECOMP_LIBRARIES<|`
	List of precompiled library names.

Model reference support **Note** For examples of the tokens in this section, see "Providing Model Referencing Support in the TMF" on page 94-62.	
`|>MODELLIB<|`	Name of the library file generated for the current model.
`|>MODELREFS<|`	List of models referenced by the top model.
`|>MODELREF_LINK_LIBS<|`	List of referenced model libraries against which the top model links.
`|>MODELREF_LINK_RSPFILE_NAME<|`	Name of a response file against which the top model links. This token is valid only for build environments that support linker response files. For an example of its use, see *`matlabroot`*`/toolbox/coder/compile/tmf/grt_vcx64.tmf`.
`|>MODELREF_TARGET_TYPE<|`	Type of target being built. Possible values are • `NONE`: Standalone model or top model referencing other models • `RTW`: Model reference target build • `SIM`: Model reference simulation target build

Token	Expansion
`|>RELATIVE_PATH_TO_ANCHOR<|`	Relative path, from the location of the generated makefile, to the MATLAB working folder.

These tokens are expanded by substitution of parameter values known to the build process. For example, if the source model contains blocks with two different sample times, the TMF statement

```
NUMST = |>NUMST<|
```

expands to:

```
NUMST = 2
```

In addition to the above, `make_rtw` expands tokens from other sources:

- Target-specific tokens defined in the target options of the Configuration Parameters dialog box
- Structures in the `rtwoptions` section of the system target file. Structures in the `rtwoptions` structure array that contain the field `makevariable` are expanded.

The following example is extracted from *`matlabroot`*`/rtw/c/grt/grt.tlc`. The section starting with `BEGIN_RTW_OPTIONS` contains MATLAB code that sets up `rtwoptions`. The following directive causes the `|>EXT_MODE<|` token to be expanded to `1` (on) or `0` (off), depending on how you set the external mode options.

```
rtwoptions(2).makevariable = 'EXT_MODE'
```

Invoke the make Utility

- “make Command” on page 94-49
- “make Utility Versions” on page 94-50

make Command

After creating *`model_or_sharedutils`*`.mk` from your TMF, the build process invokes a `make` command. To invoke `make`, the build process issues this command.

```
makecommand -f model_or_sharedutils.mk
```

`makecommand` is defined by the `MAKECMD` macro in your system target file TMF (see “Structure of the Template Makefile” on page 94-50). You can specify additional options to `make` by using the model configuration parameter **Make command**. (See the sections “Specify a Make Command” and “Template Makefiles and Make Options”.)

For example, specifying `OPT_OPTS=-O2` in the **Make command** field causes `make_rtw` to generate the following `make` command.

```
makecommand -f model_or_sharedutils.mk OPT_OPTS=-O2
```

A comment at the top of the TMF specifies the available `make` command options. If these options do not provide you with enough flexibility, you can configure your own TMF.

make Utility Versions

The make utility lets you control nearly every aspect of building your real-time program. There are several different versions of `make` available. The code generator provides the Free Software Foundation GNU `make` for both UNIX[8] and PC platforms in platform-specific subfolders under

matlabroot/bin

It is possible to use other versions of `make` with the code generator, although GNU Make is recommended. To be compatible with the code generator, verify that your version of `make` supports the following command format.

makecommand -f *model_or_sharedutils*.mk

Structure of the Template Makefile

A TMF has multiple sections, including the following:

- `Abstract` — Describes what the makefile targets. Here is a representative abstract from the ERT TMFs in *matlabroot*/toolbox/coder/compile/tmf:

```
# Abstract:
#       Template makefile for building a Windows-based stand-alone embedded
#       real-time version of Simulink model using generated C code and the
#          Microsoft Visual C/C++ compiler for x64.
#
#       Note that this template is automatically customized by the build
#       procedure to create "<model>.mk"
#
#       The following defines can be used to modify the behavior of the
#       build:
#         OPT_OPTS       - Optimization option. See DEFAULT_OPT_OPTS in
#                          vctools.mak for default.
#         OPTS           - User specific options.
#         CPP_OPTS       - C++ compiler options.
#         USER_SRCS      - Additional user sources, such as files needed by
#                          S-functions.
#         USER_INCLUDES  - Additional include paths
#                          (i.e. USER_INCLUDES="-Iwhere-ever -Iwhere-ever2")
#
#       To enable debugging:
#         set DEBUG_BUILD = 1, which will trigger OPTS=-Zi (may vary with
#                               compiler version, see compiler doc)
#
#       This template makefile is designed to be used with a system target
#       file that contains 'rtwgensettings.BuildDirSuffix' see ert.tlc
```

- `Macros read by make_rtw` section — Defines macros that tell `make_rtw` how to process the TMF. Here is a representative `Macros read by make_rtw` section from the GRT TMFs in *matlabroot*/toolbox/coder/compile/tmf:

```
#------------------------ Macros read by make_rtw -----------------------------
#
# The following macros are read by the build procedure:
#
#  MAKECMD         - This is the command used to invoke the make utility
#  HOST            - What platform this template makefile is targeted for
#                    (i.e. PC or UNIX)
#  BUILD           - Invoke make from the build procedure (yes/no)?
#  SYS_TARGET_FILE - Name of system target file.

MAKECMD         = nmake
HOST            = PC
```

8 UNIX is a registered trademark of The Open Group in the United States and other countries.

```
BUILD           = yes
SYS_TARGET_FILE = any
BUILD_SUCCESS    = ^#^#^# Created

# Opt in to simplified format by specifying compatible Toolchain
TOOLCHAIN_NAME = [\
    "Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)", \
    "Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)", \
    "Microsoft Visual C++ 2015 v14.0 | nmake (64-bit Windows)"]
```

The macros in this section might include:

- `MAKECMD` — Specifies the command used to invoke the make utility. For example, if `MAKECMD = mymake`, then the `make` command invoked is

  `mymake -f` *`model_or_sharedutils`*`.mk`
- `HOST` — Specifies the platform targeted by this TMF. This can be `PC`, `UNIX`, *`computer_name`* (see the MATLAB `computer` command), or `ANY`.
- `BUILD` — Instructs `make_rtw` whether or not it should invoke `make` from the build procedure. Specify `yes` or `no`.
- `SYS_TARGET_FILE` — Specifies the name of the system target file or the value `any`. This is used for consistency checking by `make_rtw` to verify the system target file specified in the **Target selection** panel of the **Code Generation** pane of the Configuration Parameters dialog box. If you specify `any`, you can use the TMF with any system target file.
- `BUILD_SUCCESS` — When the makefile creates a static library, it produces the `BUILD_SUCCESS` string as defined, for example, by this rule in `ert_vcx64.tmf`:

  ```
  $(PRODUCT) : $(OBJS) $(LIBS) $(MODELREF_LINK_LIBS)
      @cmd /C "echo ### Linking ..."
      $(LD) $(LDFLAGS) $(LIBS) \
      @$(CMD_FILE) @$(MODELREF_LINK_RSPFILE) -dll -def:$(MODEL).def -out:$@
      @cmd /C "echo $(BUILD_SUCCESS) dynamically linked library  $(PRODUCT)"
  ```

  When the build process finds the `BUILD_SUCCESS` string, it polls for the existence of the updated static library file. The build process does not link the library file to the executable file until the build process can confirm the presence of the library file, with an updated timestamp, on the file system.

  If the build process does not find `BUILD_SUCCESS` in the compilation log, then the build process assumes that the updated library archive is available on the file system. If required, the build process links the library to the executable file.

- `Tokens expanded by make_rtw` section — Defines the tokens that `make_rtw` expands. Here is a brief excerpt from a representative `Tokens expanded by make_rtw` section from the ERT TMFs in *`matlabroot`*`/toolbox/coder/compile/tmf/`:

  ```
  #---------------------- Tokens expanded by make_rtw ----------------------------
  #
  # The following tokens, when wrapped with "|>" and "<|" are expanded by the
  # build procedure.
  #
  #  MODEL_NAME          - Name of the Simulink block diagram
  #  MODEL_MODULES       - Any additional generated source modules
  #  MAKEFILE_NAME       - Name of makefile created from template makefile <model>.mk
  #  MATLAB_ROOT         - Path to where MATLAB is installed.
  ...

  MODEL                   = |>MODEL_NAME<|
  MODULES                 = |>MODEL_MODULES<|
  PRODUCT                 = |>PRODUCT<|
  MAKEFILE                = |>MAKEFILE_NAME<|
  ```

```
MATLAB_ROOT            = |>MATLAB_ROOT<|
...
```

For more information about TMF tokens, see Template Makefile Tokens Expanded by make_rtw.

- Subsequent sections vary based on compiler, host, and target. Some common sections include `Model and reference models`, `External mode`, `Tool Specifications` or `Tool Definitions`, `Include Path`, `C Flags`, `Additional Libraries`, and `Source Files`.
- `Rules` section — Contains the make rules used in building an executable from the generated source code. The build rules are typically specific to your version of make. The `Rules` section might be followed by related sections such as `Dependencies`.

Customize and Create Template Makefiles

- "Introduction" on page 94-52
- "Setting Up a Template Makefile" on page 94-52
- "Using Macros and Pattern Matching Expressions in a Template Makefile" on page 94-52
- "Customizing Generated Makefiles with rtwmakecfg" on page 94-54
- "Supporting Continuous Time in Custom Targets" on page 94-54
- "Model Reference Considerations" on page 94-55
- "Timeout Error When Building Shared Utility Code with Custom Template Makefile" on page 94-55

Introduction

This section describes the mechanics of setting up a custom template makefile (TMF) and incorporating it into the build process. It also discusses techniques for modifying a TMF and MATLAB file mechanisms associated with the TMF.

Before creating a custom TMF, you should read "Folder and File Naming Conventions" on page 94-6 to understand the folder structure and MATLAB path requirements for custom targets.

Setting Up a Template Makefile

To customize or create a new TMF, copy an existing GRT or ERT TMF from *`matlabroot`*`/toolbox/coder/compile/tmf`.

Place the copy in the same folder as the associated system target file. Usually, this is the `mytarget/mytarget` folder within the target folder structure. Then, rename your TMF (for example, `mytarget.tmf`) and modify it.

To allow the build process to locate and select your TMF, you must provide information in the system target file file header (see "System Target File Structure" on page 94-19). For a target that implements a single TMF, the standard way to specify the TMF to be used in the build process is to use the TMF directive of the system target file file header.

```
TMF: mytarget.tmf
```

Using Macros and Pattern Matching Expressions in a Template Makefile

This section shows, through an example, how to use macros and file-pattern-matching expressions in a TMF to generate commands in *`model_or_sharedutils`*`.mk`.

The make utility processes *model_or_sharedutils*.mk and generates a set of commands based upon dependency rules defined in *model_or_sharedutils*.mk. After make generates the set of commands for building or rebuilding test, make executes them.

For example, to build a program called test, make must link the object files. However, if the object files don't exist or are out of date, make must compile the source code. Thus there is a dependency between source and object files.

Each version of make differs slightly in its features and how rules are defined. For example, consider a program called test that gets created from two sources, file1.c and file2.c. Using most versions of make, the dependency rules would be

```
test: file1.o file2.o
        cc -o test file1.o file2.o

file1.o: file1.c
        cc -c file1.c

file2.o: file2.c
        cc -c file2.c
```

In this example, a UNIX[9] environment is assumed. In a PC environment the file extensions and compile and link commands are different.

In processing the first rule

```
test: file1.o file2.o
```

make sees that to build test, it needs to build file1.o and file2.o. To build file1.o, make processes the rule

```
file1.o: file1.c
```

If file1.o doesn't exist, or if file1.o is older than file1.c, make compiles file1.c.

The format of TMFs follows the above example. Our TMFs use additional features of make such as macros and file-pattern-matching expressions. In most versions of make, a macro is defined with

```
MACRO_NAME = value
```

References to macros are made with $(MACRO_NAME). When make sees this form of expression, it substitutes *value* for $(MACRO_NAME).

You can use pattern matching expressions to make the dependency rules more general. For example, using GNU[10] Make, you could replace the two “file1.o: file1.c” and “file2.o: file2.c” rules with the single rule

```
%.o : %.c
        cc -c $<
```

Note that $< in the previous example is a special macro that equates to the dependency file (that is, file1.c or file2.c). Thus, using macros and the "%" pattern matching character, the previous example can be reduced to

9 UNIX is a registered trademark of The Open Group in the United States and other countries.
10 GNU is a registered trademark of the Free Software Foundation.

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)

test: $(OBJS)
        cc -o $@ $(OBJS)

%.o : %.c
        cc -c $<
```

Note that the `$@` macro above is another special macro that equates to the name of the current dependency target, in this case `test`.

This example generates the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the text substitution feature for macro expansion. It replaces the source file extension (for example, `.c`) with the object file extension (`.o`). This example also generalized the build rule for the program, `test`, to use the special "`$@`" macro.

Customizing Generated Makefiles with rtwmakecfg

TMFs provide tokens that let you add the following items to generated makefiles:

- Source folders
- Include folders
- Run-time library names
- Run-time module objects

S-functions can add this information to the makefile by using an `rtwmakecfg.m` file function. This function is particularly useful when building a model that contains one or more of your S-Function blocks, such as device driver blocks.

To add information pertaining to an S-function to the makefile,

1. Create the function `rtwmakecfg` in file `rtwmakecfg.m`. The code generator associates this file with your S-function based on its folder location.
2. Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions.

After the TLC phase of the build process, when generating a makefile from the TMF, the build process searches for an `rtwmakecfg.m` file in the folder that contains the S-function component. If it finds the file, the build process calls the `rtwmakecfg` function. For more information, see “Use rtwmakecfg.m API to Customize Generated Makefiles”.

Supporting Continuous Time in Custom Targets

If you want your custom ERT-based target to support continuous time, you must update your template makefile (TMF) and the static main program module (for example, `mytarget_main.c`) for your target.

Template Makefile Modifications

Add the `NCSTATES` token expansion after the `NUMST` token expansion, as follows:

```
NUMST = |>NUMST<|
NCSTATES = |>NCSTATES<|
```

In addition, add NCSTATES to the CPP_REQ_DEFINES macro, as in the following example:

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DNUMST=$(NUMST) -DNCSTATES=$(NCSTATES) \
-DMAT_FILE=$(MAT_FILE)
-DINTEGER_CODE=$(INTEGER_CODE) \
-DONESTEPFCN=$(ONESTEPFCN) -DTERMFCN=$(TERMFCN) \
-DHAVESTDIO
-DMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE) \
```

Modifications to Main Program Module

The main program module defines a static main function that manages task scheduling for the supported tasking modes of single- and multiple-rate models. NUMST (the number of sample times in the model) determines whether the main function calls multirate or single-rate code. However, when a model uses continuous time, do not rely on NUMST directly.

When the model has continuous time and the flag TID01EQ is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and TID01EQ is true, the generated code has a single-rate call interface.

To support models that have continuous time, update the static main module to take TID01EQ into account, as follows:

1. Before NUMST is referenced in the file, add the following code:

   ```
   #if defined(TID01EQ) && TID01EQ == 1 && NCSTATES == 0
   #define DISC_NUMST (NUMST - 1)
   #else
   #define DISC_NUMST NUMST
   #endif
   ```

2. Replace instances of NUMST in the file by DISC_NUMST.

Model Reference Considerations

See “Support Model Referencing” on page 94-61 for important information on TMF modifications you may need to make to support the code generator model referencing features.

Note If you are using a TMF without the variable MODELREFS, the file might have been used with a previous release of Simulink software. If you want your TMF to support model referencing, add MODELREFS to the make file.

Timeout Error When Building Shared Utility Code with Custom Template Makefile

If the build process uses a custom template makefile to create a makefile for the shared utility code, the build process produces a timeout error when all of these conditions apply:

- The configuration parameter GenCodeOnly is 'off'.
- The BUILD_SUCCESS message is output during the make call to rtwshared.mk.
- The make call to rtwshared.mk does not update the final product (typically rtwshared.lib).

To avoid the error, update the custom template makefile so that the make call generates the BUILD_SUCCESS message only when the final product is generated (as identified by the make variable PRODUCT). Alternatively, if the build process does not use the generated makefile, disable makefile generation by setting the configuration parameter GenerateMakefile to 'off'.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”
- “Custom Target Optional Features”
- “Support Toolchain Approach with Custom Target”

Custom Target Optional Features

This section describes how to configure a custom embedded target to support these optional features.

To ...	Use Target Configuration Parameters ...	For more information, see ...
Indicate a custom target is toolchain-compliant	`UseToolchainInfoCompliant` `GenerateMakefile`	“Support Toolchain Approach with Custom Target”
Build a model that includes referenced models and uses a custom target	`ModelReferenceCompliant` `ParMdlRefBuildCompliant` (parallel build support)	“Support Model Referencing”
Control the compiler optimization level building generated code for a custom target	`CompOptLevelCompliant`	“Support Compiler Optimization Level Control”
Control C function prototypes of initialize and step functions that are generated for a model that uses a custom target	`ModelStepFunctionPrototypeControlCompliant` (ERT only)	“Support C Function Prototype Control”
Control C++ class interfaces that are generated for a model that uses a custom target	`CPPClassGenCompliant` (ERT only)	“Support C++ Class Interface Control”
Enable concurrent execution of multiple tasks on a multicore platform for a model that uses a custom target	`ConcurrentExecutionCompliant`	“Support Concurrent Execution of Multiple Tasks”

The required configuration changes are modifications to your system target file, and in some cases also modifications to your template makefile (TMF) or your custom static main program.

The API for system target file callbacks provides a function `SelectCallback` for use in system target files. `SelectCallback` is associated with the target rather than with its individual options. If you implement a `SelectCallback` function for a target, it is triggered whenever the user selects the target in the System Target File Browser.

The API provides the functions `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` for controlling custom target configuration options from a user-written `SelectCallback` function. (For function descriptions and examples, see the function reference pages.)

The general requirements for supporting one of the optional features include:

- To support model referencing or compiler optimization level control, the target must be derived from the GRT or the ERT target. To support C function prototype control or C++ class interface control, the target must be derived from the ERT target.
- The system target file must declare feature compliance by including one of the target configuration parameters listed above in a `SelectCallback` function call.

- Additional changes such as TMF modifications or static main program modifications may be required, depending on the feature. See the detailed steps in the subsections for individual features.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Support Toolchain Approach with Custom Target”
- “Support Model Referencing”
- “Support Compiler Optimization Level Control”
- “Support C Function Prototype Control”
- “Support C++ Class Interface Control”
- “Support Concurrent Execution of Multiple Tasks”

Support Toolchain Approach with Custom Target

This section describes how to configure a custom system target file to support builds with the toolchain approach.

In the Configuration Parameters dialog box, on the Code Generation pane of, you can set the build controls for the toolchain approach or the template makefile approach. The model configuration parameters that contribute to determining which build controls appear include these parameters.

Model Parameter	Value	Notes
`UseToolchainInfoCompliant`	`on`	For toolchain approach, set this parameter to `'on'`. For TMF approach, set this parameter to `'off'`.
`GenerateMakefile`	`on`	For toolchain approach, set this parameter to `'on'`.

When the dialog box detects that the selected target has these properties, the dialog box recognizes the target as toolchain-compliant and displays the build controls for the toolchain approach.

Because the custom target file cannot set these properties directly, use a `SelectCallback` function in the custom target file to set the properties. The `SelectCallback` function call in the RTW_OPTION section of the TLC file can take the form:

```
rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
```

A corresponding callback function can contain:

```
function enableToolchainCompliant(hSrc, hDlg)
  % The following parameters enable toolchain compliance.
  slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
  slConfigUISetVal(hDlg, hSrc, 'GenerateMakefile','on');

  % The following parameters are not required for toolchain compliance.
  % But, it is recommended practice to set these default values and
  % disable the parameters (as shown).
  slConfigUISetVal(hDlg, hSrc, 'RTWCompilerOptimization','off');
  slConfigUISetVal(hDlg, hSrc, 'MakeCommand','make_rtw');
  slConfigUISetEnabled(hDlg, hSrc, 'RTWCompilerOptimization',false);
  slConfigUISetEnabled(hDlg, hSrc, 'MakeCommand',false);
  hCS = hSrc.getConfigSet();
  hCS.refreshDialog;
end
```

When you select your custom target, the configuration parameters dialog box displays the toolchain approach build controls. For an example, see “Create a Custom Target Configuration” on page 94-33.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

See Also

More About

- “Customize System Target Files”
- “Support Model Referencing”

- “Support Compiler Optimization Level Control”
- “Support C Function Prototype Control”
- “Support C++ Class Interface Control”
- “Support Concurrent Execution of Multiple Tasks”

Support Model Referencing

If you want to use a custom system target file for building a model that has referenced models, you must configure the custom system target file to support model referencing.

Requirements for Model Referencing with a Custom System Target File

To build a model that references other models:

- Use a custom system target file that is derived from a GRT or ERT system target file.
- The custom system target file must declare model reference compliance –– see “Declaring Model Referencing Compliance” on page 94-61.
- The template makefile must define some entities that support model referencing –– see “Providing Model Referencing Support in the TMF” on page 94-62.

Declaring Model Referencing Compliance

To declare model reference compliance for your target, you must implement a callback function that sets the `ModelReferenceCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your system target file. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following system target file code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration.

Your callback function should set the `ModelReferenceCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ModelReferenceCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ModelReferenceCompliant',false);
hSrc.refreshDialog;
```

If you might use the target to build models containing large model reference hierarchies, consider configuring the target to support parallel builds, as discussed in “Reduce Build Time for Referenced Models by Using Parallel Builds”.

To configure a target for parallel builds, your callback function must also set the `ParMdlRefBuildCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ParMdlRefBuildCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ParMdlRefBuildCompliant',false);
hSrc.refreshDialog;
```

For more information about the system target file callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

Providing Model Referencing Support in the TMF

To configure the template makefile (TMF) for model referencing:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

```
MODELREFS                 = |>MODELREFS<|
MODELLIB                  = |>MODELLIB<|
MODELREF_LINK_LIBS        = |>MODELREF_LINK_LIBS<|
MODELREF_LINK_RSPFILE     = |>MODELREF_LINK_RSPFILE_NAME<|
RELATIVE_PATH_TO_ANCHOR   = |>RELATIVE_PATH_TO_ANCHOR<|
MODELREF_TARGET_TYPE      = |>MODELREF_TARGET_TYPE<|
```

   The following code excerpt shows how makefile tokens are expanded for a referenced model.

```
MODELREFS                 =
MODELLIB                  = engine3200cc_rtwlib.a
MODELREF_LINK_LIBS        =
MODELREF_LINK_RSPFILE     =
RELATIVE_PATH_TO_ANCHOR   = ../../..
MODELREF_TARGET_TYPE      = RTW
```

   The following code excerpt shows how makefile tokens are expanded for the top model that references the referenced model.

```
MODELREFS                 = engine3200cc transmission
MODELLIB                  =
MODELREF_LINK_LIBS        = engine3200cc_rtwlib.a transmission_rtwlib.a
MODELREF_LINK_RSPFILE     =
RELATIVE_PATH_TO_ANCHOR   = ..
MODELREF_TARGET_TYPE      = NONE
```

Token	Expands to
`MODELREFS` for the top model	List of referenced model names.
`MODELLIB`	Name of the library generated for the model.
`MODELREF_LINK_LIBS` token for the top model	List of referenced model libraries that the top model links against.
`MODELREF_LINK_RSPFILE` token for the top model	Name of a response file that the top model links against. This token is valid only for build environments that support linker response files. For an example of its use, see *`matlabroot/`*`toolbox/coder/compile/tmf/grt_vcx64.tmf`.
`RELATIVE_PATH_TO_ANCHOR`	Relative path, from the location of the generated makefile, to the MATLAB working folder.
`MODELREF_TARGET_TYPE`	Signifies the type of target being built. Possible values are • `NONE`: Standalone model or top model referencing other models • `RTW`: Model reference target build • `SIM`: Model reference simulation target build

2. Add the `RELATIVE_PATH_TO_ANCHOR` include path to the overall `INCLUDES` variable.

```
INCLUDES = -I. -I$(RELATIVE_PATH_TO_ANCHOR) $(ADD_INCLUDES) \
       $(USER_INCLUDES) $(SHARED_INCLUDES)
```

3. Change the `SRCS` variable in your TMF so that it initially lists only common modules. Additional modules are then appended conditionally, as described in the next step. For example, change

```
SRCS  = $(MODEL).c $(MODULES) ert_main.c $(ADD_SRCS) $(EXT_SRC)
```

to

```
SRCS = $(MODULES)
```

4 Create variables to define the final target of the makefile. You can remove variables that may have existed for defining the final target. For example, remove

```
PROGRAM   = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
  # Top model for RTW
  PRODUCT            = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL)
  BIN_SETTING        = $(LD) $(LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
  BUILD_PRODUCT_TYPE = "executable"
  # ERT based targets
  SRCS               += $(MODEL).c ert_main.c $(EXT_SRC)
  # GRT based targets
  # SRCS             += $(MODEL).c grt_main.c rt_sim.c $(EXT_SRC) $(SOLVER)

else
  # sub-model for RTW
  PRODUCT            = $(MODELLIB)
  BUILD_PRODUCT_TYPE = "library"
endif
```

Note If the template makefile is associated with a toolchain, remove `$(MODEL).c` or `$(MODEL).$(TARGET_LANG_EXT)` from the `SRCS` list.

5 Create rules for the final target of the makefile (replace existing final target rules). For example:

```
ifeq ($(MODELREF_TARGET_TYPE),NONE)
  # Top model for RTW
  $(PRODUCT) : $(OBJS) $(LIBS) $(MODELREF_LINK_LIBS)
               $(BIN_SETTING) $(LINK_OBJS) $(MODELREF_LINK_LIBS)
               $(LIBS)
               @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
  # sub-model for RTW
  $(PRODUCT) : $(OBJS) $(LIBS)
               @rm -f $(MODELLIB)
               $(ar) ruvs $(MODELLIB) $(LINK_OBJS)
               @echo "### Created $(MODELLIB)"
               @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
endif
```

6 Create a rule to allow referenced models to compile files that reside in the MATLAB working folder (`pwd`).

```
%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
    $(CC) -c $(CFLAGS) $<
```

Note If you are using a TMF without the variable `MODELREFS`, the file might have been used with a previous release of Simulink software. If you want your TMF to support model referencing, add either variable `MODELREFS` to the make file.

Controlling Configuration Option Value Agreement

By default, the value of a configuration option defined in the system target file for a TLC-based custom target must be the same in a referenced model and its parent model. To relax this

requirement, include the `modelReferenceParameterCheck` field in the `rtwoptions` structure element that defines the configuration option, and set the value of the field to `'off'`. For example:

```
rtwoptions(2).prompt         = 'My Custom Parameter';
rtwoptions(2).type           = 'Checkbox';
rtwoptions(2).default        = 'on';
rtwoptions(2).modelReferenceParameterCheck  = 'on';
rtwoptions(2).tlcvariable    = 'mytlcvariable';
...
```

The configuration option **My Custom Parameter** can differ in a referenced model and its parent model. See “Customize System Target Files” on page 94-19 for information about TLC-based system target files, and rtwoptions Structure Fields Summary for a list of `rtwoptions` fields.

Preventing Resource Conflicts (Optional)

Hook files are optional TLC and MATLAB program files that are invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.

If you are adapting your custom target for code generation compatibility with model reference features, consider adding checks to your hook files for handling referenced models differently than top models to prevent resource conflicts.

For example, consider adding the following check to your *STF*`_make_rtw_hook.m file`:

```
% Check if this is a referenced model
mdlRefTargetType = get_param(codeGenModelName,'ModelReferenceTargetType');
isNotModelRefTarget = strcmp(mdlRefTargetType, 'NONE'); % NONE, SIM, or RTW
if isNotModelRefTarget
  % code that is specific to the top model
else
  % code that is specific to the referenced model
end
```

You may need to do a similar check in your TLC code.

```
%if !IsModelReferenceTarget()
  %% code that is specific to the top model
%else
  %% code that is specific to the referenced model
%endif
```

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”
- “Model Reference Basics”
- “Generate Code for Model Reference Hierarchy”

Support Compiler Optimization Level Control

This section describes how to configure a custom embedded target to support compiler optimization level control. Without the described modifications, you cannot use model configuration parameter **Compiler optimization level** to control the compiler optimization level for building generated code. For more information about compiler optimization level control, see Compiler optimization level.

About Compiler Optimization Level Control and Custom Targets

The requirements for supporting compiler optimization level control are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file must declare compiler optimization level control compliance, as described in “Declaring Compiler Optimization Level Control Compliance” on page 94-65.
- The target makefile must honor the setting for model configuration parameter **Compiler optimization level**, as described in “Providing Compiler Optimization Level Control Support in the Target Makefile” on page 94-66.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

Declaring Compiler Optimization Level Control Compliance

To declare compiler optimization level control compliance for your target, you must implement a callback function that sets the `CompOptLevelCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your system target file. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following system target file code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration.

Your callback function should set the `CompOptLevelCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'CompOptLevelCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'CompOptLevelCompliant',false);
hSrc.refreshDialog;
```

For more information about the system target file callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `CompOptLevelCompliant` target configuration parameter is set to `on`, the **Compiler optimization level** parameter is displayed in the **Code Generation** pane of the Configuration Parameters dialog box for your model.

Providing Compiler Optimization Level Control Support in the Target Makefile

As part of supporting compiler optimization level control for your target, you must modify the target makefile to honor the setting for model configuration parameter **Compiler optimization level**. Use a GRT or ERT target provided by MathWorks as a model for making the modifications.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”

Support C Function Prototype Control

This section describes how to configure a custom embedded target to support function prototype control. Without the described modifications, you will not be able to use available interfaces for customizing entry-point function interfaces.

Function prototype control is not supported for component models configured with a service interface. For more information, see “Model Code Interfaces” on page 3-2 and “Configure Code Interfaces for Code Generation” on page 3-4.

For more information, see “Configure Generated C Function Interface for Model Entry-Point Functions” on page 33-114.

About C Function Prototype Control and Custom Targets

The requirements for supporting C function prototype control are as follows:

- The system target file must be derived from the ERT system target file.
- The system target file must declare C function prototype control compliance, as described in “Declaring C Function Prototype Control Compliance” on page 94-67.
- If your system target file uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, the static main program must call the function prototype controlled initialize and step functions, as described in “Providing C Function Prototype Control Support in the Custom Static Main Program” on page 94-68.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

Declaring C Function Prototype Control Compliance

To declare C function prototype control compliance for your target, you must implement a callback function that sets the `ModelStepFunctionPrototypeControlCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your system target file. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following system target file code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration.

Your callback function should set the `ModelStepFunctionPrototypeControlCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant',false);
hSrc.refreshDialog;
```

For more information about the system target file callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ModelStepFunctionPrototypeControlCompliant` target configuration parameter is set to `on`, you can use the Configure C Step Function Interface dialog box to control the function prototype of the base-rate step function generated for a rate-based model.

Providing C Function Prototype Control Support in the Custom Static Main Program

If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, you must update the static main program to call the function prototype controlled initialize and step functions. You can do this in either of the following ways:

1. Manually adapt your static main program to declare model data and call the function prototype controlled initialize and step functions.
2. Generate your main program by selecting model configuration parameter **Generate an example main program**. The generated main program declares model data and calls the function prototype controlled initialize and step function.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”

Support C++ Class Interface Control

This section describes how to configure a custom embedded target to support C++ class interface control. Without the described modifications, you will not be able to use `C++ class` code interface packaging to generate and configure C++ class interfaces to model code. For more information about C++ class interface control, see “Interactively Configure C++ Interface” on page 44-2.

About C++ Class Interface Control and Custom Targets

The requirements for supporting C++ class interface control are as follows:

- The target must be derived from the ERT system target file.
- The system target file must declare C++ class interface control compliance, as described in “Declaring C++ Class Interface Control Compliance” on page 94-69.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

Declaring C++ Class Interface Control Compliance

To declare C++ class interface control compliance for your target, you must implement a callback function that sets the `CPPClassGenCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your system target file. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following system target file code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration.

Your callback function should set the `CPPClassGenCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'CPPClassGenCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'CPPClassGenCompliant',false);
hSrc.refreshDialog;
```

For more information about the system target file callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `CPPClassGenCompliant` target configuration parameter is set to `on`, you can use the `C++ class` code interface packaging to generate and configure C++ class interfaces to model code.

Providing C++ Class Interface Control Support in the Custom Static Main Program

Selecting `C++ class` code interface packaging for your model enables model configuration parameter **Generate an example main program**. When you select this parameter,the code generator produces an example main program, `ert_main.cpp`. The generated example main program declares model data and calls the C++ class interface configured model step method, and illustrates how the generated code can be deployed.

To customize the build process and disable generation and inclusion of an example main program, see the `setTargetProvidesMain` function. Disabling example main generation permits including a custom main program.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”

Support Concurrent Execution of Multiple Tasks

If a custom embedded target must support concurrent execution of multiple tasks on a multicore platform, the target must declare support for concurrent execution by setting the target configuration option `ConcurrentExecutionCompliant`. Otherwise, you will not be able to configure a multicore target model for concurrent execution.

If `ConcurrentExecutionCompliant` is not already configured for your custom target, you can set the option in the following ways:

- Include the following code directly in your system target file (*`mytarget`*`.tlc`):

  ```
  rtwgensettings.SelectCallback = 'slConfigUISetVal(hDlg,hSrc,...
       ''ConcurrentExecutionCompliant'',''on'');';
  rtwgensettings.ActivateCallback = 'slConfigUISetVal(hDlg,hSrc,...
       ''ConcurrentExecutionCompliant'',''on'');';
  ```

- Implement a callback function that sets the `ConcurrentExecutionCompliant` option, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your system target file. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following system target file code installs a `SelectCallback` function named `custom_select_callback_handler`:

  ```
  rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration.

  Your callback function should set the `ConcurrentExecutionCompliant` option as follows:

  ```
  slConfigUISetVal(hDlg,hSrc,'ConcurrentExecutionCompliant','on');
  slConfigUISetEnabled(hDlg,hSrc,'ConcurrentExecutionCompliant',false);
  hSrc.refreshDialog;
  ```

  For more information about the system target file callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ConcurrentExecutionCompliant` target configuration option is set to `'on'`, you can select the custom target and configure your multicore target model for concurrent execution.

For an example that shows how to configure custom target optional features, see “Customize System Target Files”.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”

Device Drivers

Device drivers that communicate with target hardware are essential to many real-time development projects.

You can integrate existing C (or C++) device driver functions into Simulink models by using the Legacy Code Tool. When you use the code generator to generate code from a model, the Legacy Code Tool can insert a call to your C function into the generated code. For details, see “Import Calls to External Code into Generated Code with Legacy Code Tool” and “Integrate C Functions into Simulink Models with Legacy Code Tool”.

See Also

More About

- “Custom Targets”
- “Sample Custom Targets”
- “Customize System Target Files”

Build Configurations for Embedded Targets in Embedded Coder

Model Setup

In this section...
“Block Selection” on page 95-2 “Configure Target Hardware Resources” on page 95-2

Block Selection

You can create models for targeting the same way you create other Simulink models—by combining standard blocks and C-MEX S-functions.

You can use blocks from the following sources:

- The Embedded Coder Support Packages.
- The Embedded Targets library (`embeddedtargetslib`) in the Embedded Coder product.
- Blocks from the System Toolboxes products
- Custom blocks

Avoid using blocks that do not generate code, including the following blocks.

Block Name/Category	Library	Description
Scope	Simulink, DSP System Toolbox software	Provides oscilloscope view of your output. Do not use the **Save data to workspace** option on the **Data history** pane in the Scope Parameters dialog.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	DSP System Toolbox	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	DSP System Toolbox	Send data to your MATLAB workspace.
Signal To Workspace	DSP System Toolbox	Send a signal to your MATLAB workspace.
Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
To Wave device	DSP System Toolbox	Send data to a `.wav` device.
From Wave device	DSP System Toolbox	Get data from a `.wav` device.

Configure Target Hardware Resources

To configure your Simulink model for your target hardware and toolchain:

1. In the Configuration Parameters dialog box, select **Hardware Implementation**, and then specify your **Hardware board**.

   Specifying a **Hardware board** automatically sets many **Code Generation** parameters to values appropriate for the hardware.
2. On the **Target Hardware Resources** tab, review the other parameters.
3. (Optional) In the Configuration Parameters dialog box, select **Code Generation**, and then review and modify other code generation features for your model.
4. Click **Apply**, and save the changes to your model.

See Also

"Embedded Coder Supported Hardware"

Processor-Specific Optimizations for Embedded Targets in Embedded Coder

Replace Code for Embedded Targets

In this section...

"Using a Processor-Specific Code Replacement Library to Optimize Code" on page 96-2
"Process of Determining Optimization Effects Using Real-Time Profiling Capability" on page 96-2

Using a Processor-Specific Code Replacement Library to Optimize Code

You can optimize the code the code generator produces for a specific processor by configuring the code generator to use a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see "What Is Code Replacement?" on page 57-2 and "Replace Code Generated from Simulink Models" on page 57-6. For information about developing code replacement libraries, see "What Is Code Replacement Customization?" on page 72-3 and "Develop a Code Replacement Library" on page 72-20.

Process of Determining Optimization Effects Using Real-Time Profiling Capability

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific code replacement library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific code replacement library when you generate code:

1. Enable real-time profiling in your model. Refer to "Code Execution Profiling".
2. Generate code for your project without specifying a code replacement library (the default **Code replacement library** setting is `None`).
3. Profile the code, and save the report.
4. Rebuild your project using a processor-specific code replacement library.
5. Profile the code, and save the second report.
6. Compare the profile report from running your application with the processor-specific library selected to the profile results in the first report with no code replacement library selected.

For an example of verifying the code optimization, search help for "Optimizing Embedded Code via Code Replacement Library" and select the example that matches your IDE.

Code Generation from MATLAB Code

Build Configuration for Code Generation from MATLAB Code

- “Specify Comment Style for C/C++ Code” on page 97-2
- “Specify Indent Style for C/C++ Code” on page 97-4
- “Generate Custom File and Function Banners for C/C++ Code” on page 97-6
- “Code Generation Template Files for MATLAB Code” on page 97-9
- “Customize Generated Identifiers” on page 97-18
- “Customize C/C++ File Names Generated from MATLAB Code” on page 97-20
- “Control Signed Left Shifts in Generated Code” on page 97-22
- “Control Data Type Casts in Generated Code” on page 97-23
- “Simplify Multiply Operations for Array Indexing in Loops” on page 97-26
- “Generate SIMD Code for MATLAB Functions” on page 97-27
- “Customize Data Type Replacement” on page 97-33
- “Import Custom Data Type Definitions from External Header Files” on page 97-35
- “MATLAB Code Patterns That Require a Nonempty Initialize Function in Generated Code” on page 97-37

Specify Comment Style for C/C++ Code

In this section...

"Specify Comment Style Using the MATLAB Coder App" on page 97-2
"Specify Comment Style Using the Command-Line Interface" on page 97-2

If you have an Embedded Coder, you can specify the comment style for C/C++ code generated from MATLAB code. Specify single-line style to generate single-line comments preceded by `//`. Specify multiline style to generate single-line or multiline comments delimited by `/*` and `*/`. Single-line style is the default for C++ code generation. Multiline style is the default for C code generation. For C code generation, select single-line comment style only if your compiler supports it.

Specify Comment Style Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Code Appearance** tab, select the **Include comments** check box if it is not already selected. By default, the **Include comments** check box is selected.
5. Set **Comment Style** to one of the following options.

Value	Description
`Auto(Use standard comment style of the target language)`	For C, generate multiline comments. For C++, generate single-line comments. (default)
`Single-line (Use C++-style comments)`	Generate single-line comments preceded by `//`.
`Multi-line (Use C-style comments)`	Generate single-line or multiline comments delimited by `/*` and `*/`.

Specify Comment Style Using the Command-Line Interface

1. Create a code configuration object for C/C++ code generation. For example, create a configuration object for C/C++ static library generation:

   ```
   cfg = coder.config('lib','ecoder',true);
   ```
2. Set the `CommentStyle` property to one of the following values:

Value	Description
`'Auto'`	For C, generate multiline comments. For C++, generate single-line comments. (default)
`'Single-line'`	Generate single-line comments preceded by `//`.

Value	Description
`'Multi-line'`	Generate single-line or multiline comments delimited by `/*` and `*/`.

For example, this code sets the comment style to single-line style:

```
cfg.CommentStyle='Single-line';
```

See Also

`coder.EmbeddedCodeConfig`

More About

- “Configure Build Settings”

Specify Indent Style for C/C++ Code

In this section...

"Specify Indent Style Using the MATLAB Coder App" on page 97-4
"Specify Indent Style Using the Command-Line Interface" on page 97-5

If you have an Embedded Coder license, you can control the indent style and indent size in C/C++ code generated from MATLAB code. Indent style controls the placement of braces. Indent size controls the number of characters per indentation level.

You can specify the K&R indent style or the Allman indent style. Both styles:

- Place the function opening and closing braces on their own lines at the same indentation level as the function header.
- Indent code within the function according to the indent size.
- For blocks within a function, place closing braces on a new line at the same indentation level as the control statement.
- Indent code within a block according to the indent size.

The K&R style and the Allman style differ in their placement of the opening brace for a control statement. If you want the opening brace on the same line as its control statement, select the K&R style. Here is code that has the K&R indent style:

```
void addone(const double x[6], double z[6])
{
  int i;
  for (i = 0; i < 6; i++) {
    z[i] = x[i] + 1.0;
  }
}
```

If you want the opening brace on its own line, select the Allman style. Here is code that has the Allman indent style:

```
void addone(const double x[6], double z[6])
{
  int i;
  for (i = 0; i < 6; i++)
  {
    z[i] = x[i] + 1.0;
  }
}
```

Specify Indent Style Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`

- `Executable (.exe)`

3. Click **More Settings**.
4. On the **All Settings** tab, under **Advanced**, set **Indent style** to K&R or `Allman`.
5. On the **All Settings** tab, under **Advanced**, set **Indent size** to an integer from 2 to 8.

Specify Indent Style Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

2. Set the `IndentStyle` property to `'K&R'` or `'Allman'`. For example:

```
cfg.IndentStyle = 'Allman';
```

3. Set the `IndentSize` property to an integer from 2 to 8. For example:

```
cfg.IndentSize = 4;
```

See Also

`coder.EmbeddedCodeConfig`

More About

- “Configure Build Settings”

Generate Custom File and Function Banners for C/C++ Code

When you generate C and C++ code from MATLAB code, you can use a code generation template (CGT) file to specify custom:

- File banners
- Function Banners
- File trailers
- Comments before code sections

This example shows how you can create your own CGT file and customize it to generate your own file and function banners.

1. Create a local copy of the default CGT file for MATLAB Coder and rename it. The default CGT file is `matlabcoder_default_template.cgt` in the *`matlabroot`*`/toolbox/coder/matlabcoder/templates/` folder.
2. Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If necessary, add the folder to the MATLAB path. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder. If the file is not on the MATLAB path, specify a full path to the file when adding the file to your configuration.
3. View the default template and generated output. For example, here is the default File Banner section:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: %<FileName>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

   When you generate code using this default, the file banner looks similar to this file banner:

```
/*
 * File: coderand.c
 *
 * MATLAB Coder version            : 2.7
 * C/C++ source code generated on  : 06-Apr-2014 14:34:15
 */
```

4. Edit your local copy of the CGT file. You can change the default values and add your own custom tokens. For example, here is the File Banner section with the style changed to `box` and a custom token `myCustomToken`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
```

```
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="box">
File: %<FileName>

My custom token                : %<myCustomToken>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

For more information, see “Code Generation Template Files for MATLAB Code” on page 97-9.

5 Create a configuration object for generation of a C static library for an embedded target.

```
% Create configuration object for an embedded target
cfgObj = coder.config('lib','ecoder',true);
```

6 Create a `MATLABCodeTemplate` object from your CGT file and add it to the configuration object.

```
% Specify the custom CGT file
CGTFile = 'myCGTFile.cgt';
% Use custom template
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

7 Assign values for custom tokens that you added to the template. For example, assign the value `'myValue'` to the `myCustomToken` token that you added in a previous step.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken','myValue');
```

8 Make sure that generation of comments is enabled. Otherwise, the code generator does not produce file or function banners.

```
cfgObj.GenerateComments=true;
```

9 Generate code using the configuration object that you created.

```
codegen -config cfgObj coderand
```

10 View the changes to the generated file banner. For example, here is the file banner for `coderand.c` using the customized CGT file:

```
/*************************************************************************/
/* File: coderand.c                                                      */
/*                                                                       */
/* My custom token                : myValue                              */
/*                                                                       */
/* MATLAB Coder version           : 2.7                                  */
/* C/C++ source code generated on : 06-Apr-2014 14:42:55                 */
/*************************************************************************/
```

Changes to a CGT file do not affect the generated code unless you create a `MATLABCodeTemplate` object from the modified CGT file, and then add it to the configuration object. If you modify the CGT File, `myCGTFile.cgt`, used in the previous example, you must repeat these steps:

1 Create a `MATLABCodeTemplate` object from `myCGTFile.cgt` and add it to the configuration object.

```
CGTFile = 'myCGTFile.cgt';
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

2 Assign the value `'myValue'` to the `myCustomToken` token.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken','myValue');
```

3 Generate code.

```
codegen -config cfgObj coderand
```

See Also

`coder.MATLABCodeTemplate`

More About

- "Code Generation Template Files for MATLAB Code" on page 97-9

Code Generation Template Files for MATLAB Code

In this section...
“Default CGT File” on page 97-9 “CGT File Structure” on page 97-9 “Components of the CGT File Sections” on page 97-10

A code generation template (CGT) file defines the sections in generated code that you can customize using comments and tokens. Using a code generation template (CGT) file for the generation of C and C++ code from MATLAB code, you can specify custom file banners and function banners for generated code. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. You can also customize comments before code sections. Use these banners to:

- Add a company copyright statement.
- Specify a special version symbol for your configuration management system.
- Remove timestamps.
- Add other custom information to your generated files.

For information on creating, customizing, and using a CGT file, see “Generate Custom File and Function Banners for C/C++ Code” on page 97-6.

Default CGT File

You can base your custom template on the default CGT file, `matlabcoder_default_template.cgt`, in the *`matlabroot`*`/toolbox/coder/matlabcoder/templates/` folder.

Note If you choose not to customize banners for your generated code, the default template is used for code generation.

CGT File Structure

A CGT file consists of 13 optional sections.

File Banner Section

Contains comments and tokens for use in generating a custom file banner.

Function Banner Section

Contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner

Contains comments and tokens for use in generating custom banners for shared utility functions.

File Trailer Section

Contains comments for use in generating a custom trailer banner.

Include Files Banner

Contains comments for use in generating a custom banner for the include files section.

Type Definitions

Contains comments for use in generating a custom banner for the type definitions section.

Named Constants

Contains comments for use in generating a custom banner for the named constants section.

Variable Declarations

Contains comments for use in generating a custom banner for the variable declarations section.

Variable Definitions

Contains comments for use in generating a custom banner for the variable definitions section.

Function Declarations

Contains comments for use in generating a custom banner for the function declarations section.

Function Definitions

Contains comments for use in generating a custom banner for the function definitions section.

Custom Source Code

Contains comments for use in generating a custom banner for the custom source code section.

Custom Header Code

Contains comments for use in generating a custom banner for the custom header code section.

Components of the CGT File Sections

Each CGT file section is defined by open and close tags.

CGT File Section	Open Tag	Close Tag
“File Banner” on page 97-13	`<FileBanner>`	`</FileBanner>`
“Function Banner Section” on page 97-9	`<FunctionBanner>`	`</FunctionBanner>`
“Shared Utility Function Banner” on page 97-9	`<SharedUtilityBanner>`	`</SharedUtilityBanner>`
“File Trailer Section” on page 97-9	`<FileTrailer>`	`</FileTrailer>`
“Include Files Banner” on page 97-10	`<IncludeFilesBanner>`	`</IncludeFilesBanner>`

CGT File Section	Open Tag	Close Tag
“Type Definitions” on page 97-10	`<TypeDefinitionsBanner>`	`</TypeDefinitionsBanner>`
“Named Constants” on page 97-10	`<NamedConstantsBanner>`	`</NamedConstantsBanner>`
“Variable Declarations” on page 97-10	`<VariableDeclarationsBanner>`	`</VariableDeclarationsBanner>`
“Variable Definitions” on page 97-10	`<VariableDefinitionsBanner>`	`</VariableDefinitionsBanner>`
“Function Declarations” on page 97-10	`<FunctionDeclarationsBanner>`	`</FunctionDeclarationsBanner>`
“Function Definitions” on page 97-10	`<FunctionDefinitionsBanner>`	`</FunctionDefinitionsBanner>`
“Custom Source Code” on page 97-10	`<CustomSourceCodeBanner>`	`</CustomSourceCodeBanner>`
“Custom Header Code” on page 97-10	`<CustomHeaderCodeBanner>`	`</CustomHeaderCodeBanner>`

You can customize your banners by including tokens and comments between the open and close tags for each section. Tokens are replaced with values in the generated code. The following rules apply to tokens in your CGT file:

- You can have only one token per line.
- Token values must not contain a ‘\t’ for formatting.

Note In the contents of your banner, C comment indicators, '/*' or '*/', can introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- `width`: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- `style`: specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is:

```
<OpenTag style = “style_value” width = “num_width”>
```

There are five options for the banner style. The `CommentStyle` and `TargetLang` configuration object properties determine the use of C or C++ comment style. The built-in style options for the `style` attribute are:

- `classic`

  Using C style comments

  ```
  /* single line comments */
  ```

```
/*
 * multiple line comments
 * second line
 */
```

Using C++ style comments

```
// single line comments

//
// multiple line comments
// second line
//
```

- box

  Using C style comments

```
/*********************************************************/
/* banner contents                                       */
/*********************************************************/
```

  Using C++ style comments

```
///////////////////////////////////////////////////////////
// banner contents                                       //
///////////////////////////////////////////////////////////
```

- open_box

  Using C style comments

```
/*********************************************************
 * banner contents
 *********************************************************/
```

  Using C++ style comments

```
///////////////////////////////////////////////////////////
// banner contents
///////////////////////////////////////////////////////////
```

- doxygen

  Using C style comments

```
/** single line comments */

/**
 * multiple line comments
 * second line
 */
```

  Using C++ style comments

```
///single line comments

///
/// multiple line comments
///second line
///
```

- `doxygen_qt`

  Using C style comments

  ```
  /*! single line comments */

  /*!
   * multiple line comments
   * second line
   */
  ```

  Using C++ style comments

  ```
  //!single line comments

  //!
  //! multiple line comments
  //!second line
  //!
  ```

File Banner

This section contains comments and tokens for use in generating a custom file banner that precedes the generated C and C++ code. If you omit the file banner section from the CGT file, the code generator does not generate a file banner in the generated code. The file banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: %<FileName>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

Summary of Tokens for File Banner Generation

`FileName`	Name of the generated file (for example, `"kalman.c"`)
`SourceGeneratedOn`	Timestamp of generated file
`MATLABCoderVersion`	Version of MATLAB Coder
`EmbeddedCoderVersion`	Version of Embedded Coder
`HardwareSelection`	Selected target
`OutputType`	Type of output (for example, lib, exe, or dll)

Function Banner

This section contains comments and tokens for use in generating a custom function banner that precedes a generated C or C++ function. If you omit the function banner section from the CGT file, the code generator does not generate function banners. The function banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom Function Banner section (optional)
%% Customize function banners by using the following predefined tokens:
%% %<FunctionName>, %<FunctionDescription>, %<FunctionPrototype>
%% %<Arguments>, %<ReturnType>
%%
<FunctionBanner style="classic">
%<FunctionPrototype>
%<FunctionDescription>
Arguments    : %<Arguments>
Return Type  : %<ReturnType>
</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

`FunctionName`	Name of function
`FunctionDescription`	Short abstract about the function If the configuration parameter `MATLABFcnDesc` is `true`, the function description appears in the function banner. Otherwise, the code generator treats the function description as user comments.
`FunctionPrototype`	MATLAB function prototype The function prototype appears in the function banner only if the `MATLABSourceComments` configuration parameter is `true`.
`Arguments`	List of function arguments
`ReturnType`	Return type of function

Shared Utility Banner

This section contains comments and tokens for use in generating a custom shared utility function banner that precedes a generated C or C++ shared utility function. If you omit the shared utility function banner section from the CGT file, the code generator does not generate shared utility function banners. The shared utility function banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom Shared Utility Function Banner section (optional)
%% Customize shared utility function banners by using the following
%% predefined tokens:
%% %<FunctionName>, %<FunctionDescription>
%% %<Arguments>, %<ReturnType>
%%
<SharedUtilityBanner style="classic">
Arguments    : %<Arguments>
Return Type  : %<ReturnType>
</SharedUtilityBanner>
```

Summary of Tokens for Shared Utility Function Banner Generation

`FunctionName`	Name of function
`FunctionDescription`	Short abstract about the function
`Arguments`	List of function arguments
`ReturnType`	Return type of function

File Trailer

The file trailer section contains comments for generating a custom file trailer that follows the generated C or C++ code. If you omit the file trailer section from the CGT file, the code generator does not generate a file trailer. The file trailer section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
%% You can use any of the predefined tokens used for File Banner
%%
<FileTrailer style="classic">
File trailer for %<FileName>

[EOF]
</FileTrailer>
```

Tokens for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation.

Include Files Banner

The include files banner section contains comments for generating a custom banner that precedes the include files section in the generated code. If you omit the include files banner section from the CGT file, the code generator does not generate a banner for this section. The include files banner section provided in the default CGT file is:

```
<IncludeFilesBanner  style="classic">
Include Files
</IncludeFilesBanner>
```

Type Definitions Banner

The type definitions banner section contains comments for generating a custom banner that precedes the type definitions section in the generated code. If you omit the type definitions banner section from the CGT file, the code generator does not generate a banner for this section. The type definitions banner section provided in the default CGT file is:

```
<TypeDefinitionsBanner  style="classic">
Type Definitions
</TypeDefinitionsBanner>
```

Named Constants Banner

The named constants banner section contains comments for generating a custom banner that precedes the named constants section in the generated code. If you omit the named constants banner section from the CGT file, the code generator does not generate a banner for this section. The named constants banner section provided in the default CGT file is:

```
<NamedConstantsBanner  style="classic">
Named Constants
</NamedConstantsBanner>
```

Variable Declarations

The variable declarations banner section contains comments for generating a custom banner that precedes the variable declarations section in the generated code. If you omit the variable declarations banner section from the CGT file, the code generator does not generate a banner for this section. The variable declarations banner section provided in the default CGT file is:

```
<VariableDeclarationsBanner  style="classic">
Variable Declarations
</VariableDeclarationsBanner>
```

Variable Definitions

The variable definitions banner section contains comments for generating a custom banner that precedes the variable definitions section in the generated code. If you omit the variable definitions banner section from the CGT file, the code generator does not generate a banner for this section. The variable definitions banner section provided in the default CGT file is:

```
<VariableDefinitionsBanner  style="classic">
Variable Definitions
</VariableDefinitionsBanner>
```

Function Declarations

The function declarations banner section contains comments for generating a custom banner that precedes the function declarations section in the generated code. If you omit the function declarations banner section from the CGT file, the code generator does not generate a banner for this section. The function declarations banner section provided in the default CGT file is:

```
<functionDeclarationsBanner  style="classic">
Function Declarations
</FunctionDeclarationsBanner>
```

Function Definitions

The function definitions banner section contains comments for generating a custom banner that precedes the function definitions section in the generated code. If you omit the function definitions banner section from the CGT file, the code generator does not generate a banner for this section. The function definitions banner section provided in the default CGT file is:

```
<FunctionDefinitionsBanner  style="classic">
Function Definitions
</FunctionDefinitionsBanner>
```

Custom Source Code

The custom source code banner section contains comments for generating a custom banner that precedes the custom source code section in the generated code. If you omit the custom source code banner section from the CGT file, the code generator does not generate a banner for this section. The custom source code banner section provided in the default CGT file is:

```
<CustomSourceCodeBanner  style="classic">
Custom Source Code
</CustomSourceCodeBanner>
```

Custom Header Code

The custom header code banner section contains comments for generating a custom banner that precedes the custom header code section in the generated code. If you omit the custom header code banner section from the CGT file, the code generator does not generate a banner for this section. The custom header code banner section provided in the default CGT file is:

```
<CustomHeaderCodeBanner  style="classic">
Custom Header Code
</CustomHeaderCodeBanner>
```

See Also

`coder.MATLABCodeTemplate`

More About

- “Generate Custom File and Function Banners for C/C++ Code” on page 97-6

Customize Generated Identifiers

In this section...
“Customize Identifiers by Using the MATLAB Coder App” on page 97-18 “Customize Generated Identifiers by Using the Command-Line Interface” on page 97-19

If you have Embedded Coder, you can customize the identifiers in C/C++ code generated from MATLAB code. For each kind of identifier that you want to customize, set the appropriate identifier format parameter to a macro that specifies the format of the generated identifiers. The macro can include:

- Valid C or C++ language identifiers (a-z, A-Z, _, 0–9).
- The tokens listed in the following table. `$M` is required.

Token	Description
`$M`	Code generator inserts name mangling text to avoid naming collisions. Required.
`$N`	Code generator inserts the name of the object (global variable, global type, local function, local temporary variable, or constant macro) for which the identifier is generated. Improves readability of generated code. Required.
`$R`	Code generator inserts the root project name into identifier, replacing unsupported characters with the underscore (_) character.

Customize Identifiers by Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library`
   - `Dynamic Library`
   - `Executable`
3. Click **More Settings**.
4. On the **Code Appearance** tab, under **Identifier Format**, for each kind of identifier that you want to customize, enter the macro.

Parameter	Default Macro
Global variables	`$M$N`
Global types	`$M$N`
Field name of global types	`$M$N`
Local functions	`$M$N`
Local temporary variables	`$M$N`

Parameter	Default Macro
Constant macros	`$M$N`
EMX Array Types	`emxArray_$M$N`
EMX Array Utility Functions	`emx$M$N`

For example, suppose that **Global variables** has the value `glob_$M$N`. For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling text.

Customize Generated Identifiers by Using the Command-Line Interface

1. Create a code configuration object for a library or executable program. For example:

   ```
   cfg = coder.config('lib','ecoder',true);
   ```
2. For each kind of identifier that you want to customize, specify the macro as a character vector.

Parameter	Description	Default Macro
`CustomSymbolStrGlobalVar`	Global variables	`'$M$N'`
`CustomSymbolStrType`	Global types	`'$M$N'`
`CustomSymbolStrField`	Field name of global types	`'$M$N'`
`CustomSymbolStrFcn`	Local functions	`'$M$N'`
`CustomSymbolStrTmpVar`	Local temporary variables	`'$M$N'`
`CustomSymbolStrMacro`	Constant macros	`'$M$N'`
`CustomSymbolStrEMXArray`	EMX Array Types	`'emxArray_$M$N'`
`CustomSymbolStrEMXArrayFcn`	EMX Array Utility Functions	`'emx$M$N'`

   For example:

   ```
   cfg.CustomSymbolStrGlobalVar = 'glob_$M$N';
   ```

For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling text.

See Also

`coder.EmbeddedCodeConfig`

More About

- “Configure Build Settings”

Customize C/C++ File Names Generated from MATLAB Code

In this section...

"Customize File Names by Using the MATLAB Coder App" on page 97-20
"Customize Generated File Names by Using the Command-Line Interface" on page 97-20

If you have Embedded Coder, you can customize the file names of the C/C++ code generated from MATLAB code. By changing the file names, you can avoid name clashes when integrating different C/C++ code projects together. Customize the file names by setting the file name format parameter `CustomFileNameStr` to a macro. The macro can include:

- Valid file name characters.
- The tokens listed in the following table. `$M` and `$N` are required.

Token	Description
`$M`	Code generator inserts name mangling text to avoid naming collisions. Required.
`$N`	Code generator inserts the non-customized file name. Required.
`$R`	Code generator inserts the root project name into the file name, replacing unsupported characters with the underscore (_) character.

By default, the macro is set to `$N$M`.

Customize File Names by Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library`
   - `Dynamic Library`
   - `Executable`
3. Click **More Settings**.
4. Enter the macro on the **Code Appearance** tab, next to **Generated source and header files:**

For example, suppose you set the macro to the value `ec_$N$M`. For an entry-point function called `foo`, when name mangling is not required, the generated file name for `foo` becomes `ec_foo`. If name mangling is required, the generated file name includes the name mangling text.

Customize Generated File Names by Using the Command-Line Interface

1. Create a code generation configuration object for a library or executable program. For example:

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

2. Specify the file name macro as a character vector in the `CustomFileNameStr` object property. For example:

   ```
   cfg.CustomFileNameStr = 'ec_$N$M';
   ```

   For an entry-point function called `foo`, when name mangling is not required, the generated file name for `foo` becomes `ec_foo`. If name mangling is required, the generated file name includes the name mangling text.

See Also

`coder.EmbeddedCodeConfig`

More About

- "Configure Build Settings"
- "Customize Generated Identifiers" on page 97-18

Control Signed Left Shifts in Generated Code

In this section...

"Control Signed Left Shifts Using the MATLAB Coder App" on page 97-22
"Control Signed Left Shifts Using the Command-Line Interface" on page 97-22

If you have an Embedded Coder license, you can control whether MATLAB Coder replaces multiplications by powers of two with signed left bitwise shifts. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers.

By default, MATLAB Coder replaces multiplication by powers of two with signed left shifts. Here is an example of generated C code that uses a signed left shift for multiplication by eight.

```
i <<= 3;
```

To increase the likelihood of generating MISRA C:2012 compliant code, disable the replacement of multiplication by powers of two with signed left shifts. Here is an example of generated C code that does not use a signed left shift for multiplication by eight:

```
i = i * 8;
```

Control Signed Left Shifts Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Code Appearance** tab, select or clear the **Use signed shift left for fixed-point operations and multiplication by powers of 2** check box.

Control Signed Left Shifts Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```
2. Set the `EnableSignedLeftShifts` property to `true` or `false`. For example:

   ```
   cfg.EnableSignedLeftShifts = false;
   ```

See Also

`coder.EmbeddedCodeConfig`

More About

- "Configure Build Settings"

Control Data Type Casts in Generated Code

In this section...
“Specify Casting Mode Using the MATLAB Coder App” on page 97-24 “Specify Casting Mode Using the Command-Line Interface” on page 97-24

If you have an Embedded Coder license, you can control data type casts in C/C++ code generated from MATLAB code. You can specify one of the following casting modes.

Casting Mode	Description

Nominal	Nominal casting mode is the default casting mode. Generated C/C++ code uses the default C compiler data type casting. When you do not have special data type information requirements, choose this option. Here is an example of code generated using nominal casting mode:
	`short addone(short x)`
	`{`
	`  int i;`
	`  i = x + 1;`
	`  if (i > 32767) {`
	`  i = 32767;`
	`  }`
	
	`  return (short)i;`
	`}`

Standards Compliant	Generated C/C++ code has data type casts that conform to MISRA standards. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations. Here is an example of code generated using standards-compliant casting mode:
	`short addone(short x)`
	`{`
	`  int i;`
	`  i = (int)x + (int)1;`
	`  if (i > (int)32767) {`
	`  i = (int)32767;`
	`  }`
	
	`  return (short)i;`
	`}`

Casting Mode	Description

Explicit	Generated C/C++ code has explicit data type casts. Explicit data type casts provide information about the amount of memory that the variable uses and the level of precision for calculations using the variable. Here is an example of code generated using explicit casting mode:
	`short addone(short x)`
	`{`
	`  int i;`
	`  i = (int)x + 1;`
	`  if (i > 32767) {`
	`  i = 32767;`
	`  }`
	
	`  return (short)i;`
	`}`

Specify Casting Mode Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **All Settings** tab, under **Advanced**, set **Casting mode** to one of the following values:
   - `Nominal`
   - `Standards Compliant`
   - `Explicit`

Specify Casting Mode Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```
2. Set the `CastingMode` property to one of the following values:
   - `'Nominal'`
   - `'Standards'`
   - `'Explicit'`

   For example:

   ```
   cfg.CastingMode = 'Standards';
   ```

See Also

`coder.EmbeddedCodeConfig`

More About

- "Configure Build Settings"

Simplify Multiply Operations for Array Indexing in Loops

If you use Embedded Coder to generate C/C++ code from MATLAB code, you can enable an optimization that simplifies array indexing in loops in the generated code. When possible, for array indices in loops, this optimization replaces multiply operations with add operations. Multiply operations can be expensive. This optimization, referred to as strength reduction, is useful when the C/C++ compiler on the target platform does not optimize the array indexing.

Here is code generated without the optimization:

```
for (i = 0; i < 10; i++) {
    z[5 * (1 + i) - 1] = x[5 * (1 + i)];
  }
```

Here is code generated with the optimization:

```
for (b_i = 0; b_i < 10; b_i++) {
    z[i + 4] = x[i + 5];
    i += 5;
  }
```

By default, the strength reduction optimization is disabled. To enable it:

- At the command line, set the configuration object parameter `EnableStrengthReduction` to `true`.
- In the MATLAB Coder app, project build settings, on the **All Settings** tab, set **Simplify array indexing** to `Yes`.

Even when the optimization replaces the multiply operations in the generated code, it is possible that the C/C++ compiler can generate multiply instructions.

See Also

More About

- “Optimization Strategies”
- “Configure Build Settings”

Generate SIMD Code for MATLAB Functions

You can generate single instruction, multiple data (SIMD) code from certain MATLAB functions by using Intel SSE and, if you have Embedded Coder, Intel AVX technology. SIMD is a computing paradigm in which a single instruction processes multiple data. Many modern processors have SIMD instructions that, for example, perform several additions or multiplications at once. For computationally intensive operations among supported functions, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms.

MATLAB Functions That Support SIMD Code

When certain conditions are met, you can generate SIMD code by using Intel SSE or Intel AVX technology. The following table lists MATLAB functions that support SIMD code generation. The table also details the conditions under which the support is available.

MATLAB Function	Conditions
`plus`	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32` or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`. • For integer data types, the value of **Saturate on integer overflow** is set to `No`.
`minus`	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32` or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`. • For integer data types, the value of **Saturate on integer overflow** is set to `No`.
`times`	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int16`, or `int32`. • For AVX512F, the input signal has a data type of `single` or `double`. • For integer data types, the value of **Saturate on integer overflow** is set to `No`.
`rdivide`	The input signal has a data type of `single` or `double`.

MATLAB Function	Conditions
`sqrt`	The input signal has a data type of `single` or `double`.
`ceil`	• For AVX2, AVX, SSE4.1, SSE2, and SSE, the input signal has a data type of `single` or `double`. • AVX512F is not supported.
`floor`	• For AVX2, AVX, SSE4.1, SSE2, and SSE, the input signal has a data type of `single` or `double`. • AVX512F is not supported.
`max`	• The input signal has a data type of `single` or `double`. • The value of the **Support nonfinite numbers** parameter is set to `off`.
`min`	• The input signal has a data type of `single` or `double`. • The value of the **Support nonfinite numbers** parameter is set to `off`.
`sum`	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int8`, `int16`, `int32` or `int64`. • For AVX512F, the input signal has a data type of `single` or `double`. • The **Optimize reductions** configuration parameter is set to `on`. • For integer data types, the value of **Saturate on integer overflow** is set to `No`.
`prod`	• For AVX, SSE, and FMA, the input signal has a data type of `single` or `double`. • For AVX2, SSE4.1, and SSE2, the input signal has a data type of `single`, `double`, `int16`, or `int32`. • For AVX512F, the input signal has a data type of `single` or `double`. • The **Optimize reductions** configuration parameter is set to `on`. • For integer data types, the value of **Saturate on integer overflow** is set to `No`.

MATLAB Function	Conditions
`bitand`	• For SSE2, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`. • For AVX2, and AVX512F, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`.
`bitor`	• For SSE2, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`. • For AVX2, and AVX512F, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`.
`bitxor`	• For SSE2, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`. • For AVX2, and AVX512F, the input signal has a data type of `int8`, `int16`, `int32`, or `int64`.
`bitshift`	• The input signal has a data type of `int32`.

If you have a DSP System Toolbox, you can generate SIMD code from certain MATLAB System objects. For more information, see “System objects in DSP System Toolbox that Support SIMD Code Generation” (DSP System Toolbox).

If you have the Embedded Coder Support Package for ARM Cortex-A Processors, you can generate SIMD code optimized for ARM Cortex A processors by setting the **Code replacement library** parameter to `GCC ARM Cortex-A`. You will also need to set the **Device vendor** parameter to `ARM Compatible` and the **Device type** parameter to `ARM Cortex-A`.

Generate SIMD Code Versus Plain C Code

Consider the MATLAB function `dynamic`. This function consists of addition and multiplication operations between the variable-size arrays `A` and `B`. These arrays have a data type of `single` and an upper bound of `100 x 100`.

```
function C = dynamic(A, B)
   assert(all(size(A) <= [100 100]));
   assert(all(size(B) <= [100 100]));
   assert(isa(A, 'single'));
   assert(isa(B, 'single'));

   C = zeros(size(A), 'like', A);
   for i = 1:numel(A)
       C(i) = (A(i) .* B(i)) + (A(i) .* B(i));
   end
end
```

To generate plain C code at the command line:

1. For `C` library code generation, create a code generation configuration object.

   ```
   cfg = coder.config('lib');
   ```

2. To generate a static library in the default location, `codegen\lib\dynamic`, use the `codegen` function.

   ```
   codegen('-config', cfg, 'dynamic');
   ```

3. In the list of generated files, click `dynamic.c`. In the plain (non-SIMD) C code, each loop iteration produces one result.

```
void dynamic(const float A_data[], const int A_size[2], const float B_data[],
             const int B_size[2], float C_data[], int C_size[2])
{
  float C_data_tmp;
  int i;
  int loop_ub;
  (void)B_size;
  C_size[0] = (signed char)A_size[0];
  C_size[1] = (signed char)A_size[1];
  loop_ub = (signed char)A_size[0] * (signed char)A_size[1];
  if (0 <= loop_ub - 1) {
    memset(&C_data[0], 0, loop_ub * sizeof(float));
  }
  loop_ub = A_size[0] * A_size[1];
  for (i = 0; i < loop_ub; i++) {
    C_data_tmp = A_data[i] * B_data[i];
    C_data[i] = C_data_tmp + C_data_tmp;
  }
}
```

To generate SIMD C code at the command line:

1. For C library code generation, use the `coder.config` function to create a code generation configuration object.

   ```
   cfg = coder.config('lib');
   ```

2. Set the `coder.HardwareImplementation` object `TargetHWDeviceType` property to `'Intel->x86-64 (Linux 64)'` or `'Intel->x86-64 (Windows64)'`.

   ```
   cfg.HardwareImplementation.TargetHWDeviceType = 'Intel->x86-64 (Windows64)';
   ```

3. Set the `coder.HardwareImplementation` object `ProdHWDeviceType` property to `'Intel->x86-64 (Linux 64)'` or `'Intel->x86-64 (Windows64)'`

   ```
   cfg.HardwareImplementation.ProdHWDeviceType = 'Intel->x86-64 (Windows64)';
   ```

   If you are using the MATLAB Coder app to generate code:

   - Set the **Hardware Board** parameter to `None-Select device below`.
   - Set the **Device vendor** parameter to `Intel`, `AMD`, or `Generic`.
   - Set the **Device type** to `x86-64 (Linux 64)`, `x86-64 (Windows64)`, or `MATLAB Host Computer`.

4. Set the `InstructionSetExtensions` property to an instruction set extension that your processor supports. This example uses SSE2 for Windows.

   ```
   cfg.InstructionSetExtensions = 'SSE2';
   ```

   The library that you choose depends on which instruction set extension your processor supports. If you use Embedded Coder, you can also select from the instruction sets `SSE`, `SSE4.1`, `AVX`, `AVX2`, `FMA`, and `AVX512F`.

   For more information, see https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.

   If you are using the MATLAB Coder app to generate code, on the **Speed** tab, set the **Leverage target hardware instruction set extensions** parameter to an instruction set that your processor supports.

5. Optionally, select the `OptimizeReductions` parameter to generate SIMD code for reduction operations such as sum and product functions.

   ```
   cfg.OptimizeReductions = 'on';
   ```

   If you are using the MATLAB Coder app to generate code, on the **Speed** tab, select the **Optimize reductions** parameter.

6. Use the `codegen` function to generate a static library in the default location, `codegen\lib\dynamic`.

   ```
   codegen('-config', cfg, 'dynamic');
   ```

7. In the list of generated files, click `dynamic.c`.

   ```
   void dynamic(const float A_data[], const int A_size[2], const float B_data[],
                const int B_size[2], float C_data[], int C_size[2])
   {
     __m128 r;
     float C_data_tmp;
     int i;
     int loop_ub;
     int scalarLB;
     int vectorUB;
     (void)B_size;
     C_size[0] = (signed char)A_size[0];
     C_size[1] = (signed char)A_size[1];
     loop_ub = (signed char)A_size[0] * (signed char)A_size[1];
     if (0 <= loop_ub - 1) {
       memset(&C_data[0], 0, loop_ub * sizeof(float));
     }
     loop_ub = A_size[0] * A_size[1];
     scalarLB = (loop_ub / 4) << 2;
     vectorUB = scalarLB - 4;
     for (i = 0; i <= vectorUB; i += 4) {
       r = _mm_mul_ps(_mm_loadu_ps(&A_data[i]), _mm_loadu_ps(&B_data[i]));
       _mm_storeu_ps(&C_data[i], _mm_add_ps(r, r));
     }
     for (i = scalarLB; i < loop_ub; i++) {
       C_data_tmp = A_data[i] * B_data[i];
       C_data[i] = C_data_tmp + C_data_tmp;
     }
   }
   ```

   The SIMD instructions are the intrinsic functions that start with the identifier `_mm`. These functions process multiple data in a single iteration of the loop because the loop increments by four for `single` data types. For `double` data types, the loop increments by two. For MATLAB code that processes more data and is more computationally intensive, than the code in this example, the presence of SIMD instructions can significantly speed up the code execution time.

   The second `for` loop is in the generated code because the `for` loop that contains SIMD code must be divisible by four for single data types. The second loop processes the remainder of the data.

For a list of a Intel intrinsic functions for supported MATLAB functions, see https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Generate SIMD Instructions for MEX Code

To generate SIMD instructions in MEX code, set the MEX configuration parameter **Hardware SIMD acceleration** to one of these acceleration levels.

- Full — AVX2
- Portable — SSE2
- None — No instruction sets

MATLAB Coder checks your specified hardware and compiler and uses the compatible intrinsics up to the level that you specify according to this list.

To generate SIMD instructions for MEX code at the command line, use the `coder.MexCodeConfig` object and set the `'SIMDAcceleration'` property to `'Full'`, `'Portable'`, or `'None'`.

Limitations

The generated code does not contain SIMD code when the MATLAB code meets these conditions:

- Scalar operations outside a loop. For example, if `a`,`b`, and `c` are scalars, the generated code does not contain SIMD code for an operation such as `c=a+b`.
- Indirectly indexed arrays or matrices. For example, if `A`,`B`,`C`, and `D` are vectors, the generated code does not contain SIMD code for an operation such as `D(A)=C(A)+B(A)`.
- Parallel for-Loops (`parfor`). The `parfor` loop does not contain SIMD code, but loops within the body of the `parfor` loop might contain SIMD code.
- Polyspace does not support analysis of generated code that includes SIMD instructions. Disable SIMD code generation by setting the **Leverage target hardware instruction set extensions** parameter to `None`.
- Reduction operations that operate on floating-point values when support for nonfinite numbers is enabled. To generate SIMD code for reduction operations that operate on floating-point values, in the MATLAB Coder app, set **Support nonfinite numbers** to `No`. Or, in a code generation configuration object, set the `SupportNonFinite` property to `false`.

See Also

More About

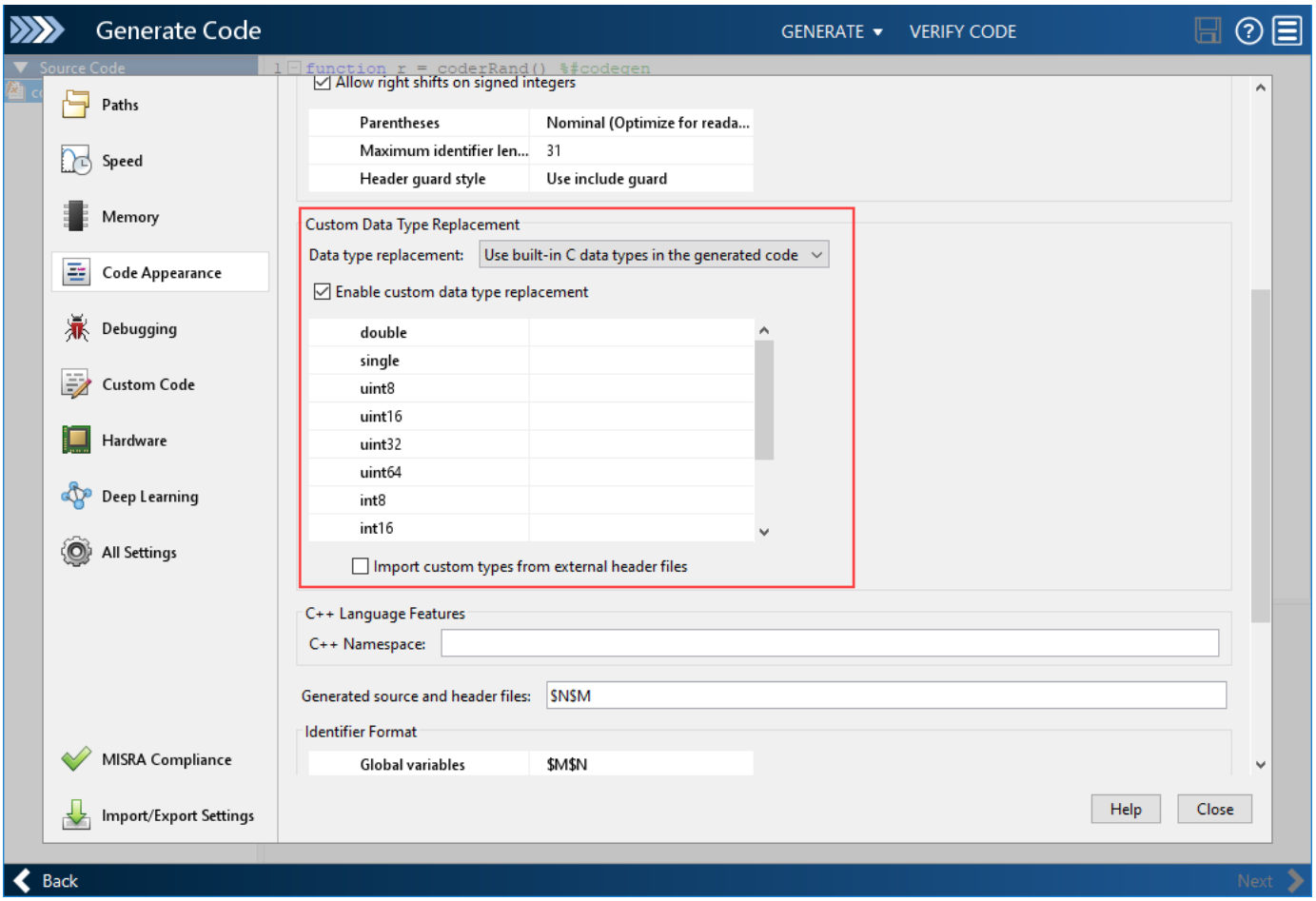
- “Generate C Code by Using the MATLAB Coder App”
- “Generate C Code at the Command Line”
- “Generate SIMD Code from Simulink Blocks” on page 76-105
- “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” (DSP System Toolbox)
- “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” (DSP System Toolbox)

Customize Data Type Replacement

While generating C/C++ code from MATLAB code, the **Data Type Replacement** option enables you to use built-in C data types or MathWorks typedefs in the generated code. Code generation supports these data types for custom renaming: `double`, `single`, `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `char`, and `logical`.

The **Data Type Replacement** option is available in the **Generate Code** window (**More Settings**), on the **Code Appearance** tab. You can select one of these options:

- `Use built-in C data types in the generated code`
- `Use MathWorks typedefs in the generated code`



Note The code generator does not replace variable-size character arrays with `char*` type strings in the generated code.

If you have Embedded Coder installed, you can specify custom names for MATLAB built-in data types in the generated C/C++ code. For example, you can customize `double` as `Custom_Double`.

Specify Custom Data Type Names by Using the MATLAB Coder App

To specify custom names for the MATLAB data types:

1. Open the MATLAB Coder app.
2. Navigate to the **Generate Code** step.
3. Click **More Settings**.
4. In the **Code Appearance** tab, select **Enable custom data type replacement** under **Custom Data Type Replacement**.
5. The table lists the name of the supported data types. Specify your custom names for these data types and press **Enter**. The specified custom names replace the built-in data type names in the generated code. If you do not specify a custom name, the code generator uses a default name.

Specify Custom Data Type Names by Using the Command-Line Interface

Use the `EnableCustomReplacementTypes` and `ReplacementTypes` properties in an Embedded Coder configuration object when you generate code by using `codegen`.

1. Write a MATLAB function from which you can generate code. This example uses the function `myAdd` that returns the sum of its inputs.

   ```
   function c = myAdd(a,b)
   c = a + b;
   end
   ```

2. Create a code configuration object for generation of a static library.

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

3. Set the `EnableCustomReplacementTypes` property to `true`.

   ```
   cfg.EnableCustomReplacementTypes = true;
   ```

4. Specify custom name for the data types. Here, the built-in data type name `uint8` is customized to `Custom_Uint8`.

   ```
   cfg.ReplacementTypes.uint8 = "Custom_Uint8";
   ```

5. Generate code by using `codegen` and `-config` option.

   ```
   codegen myAdd.m -args {1,uint8(1)} -config cfg -report
   ```

See Also

Functions
`codegen` | `coder.config`

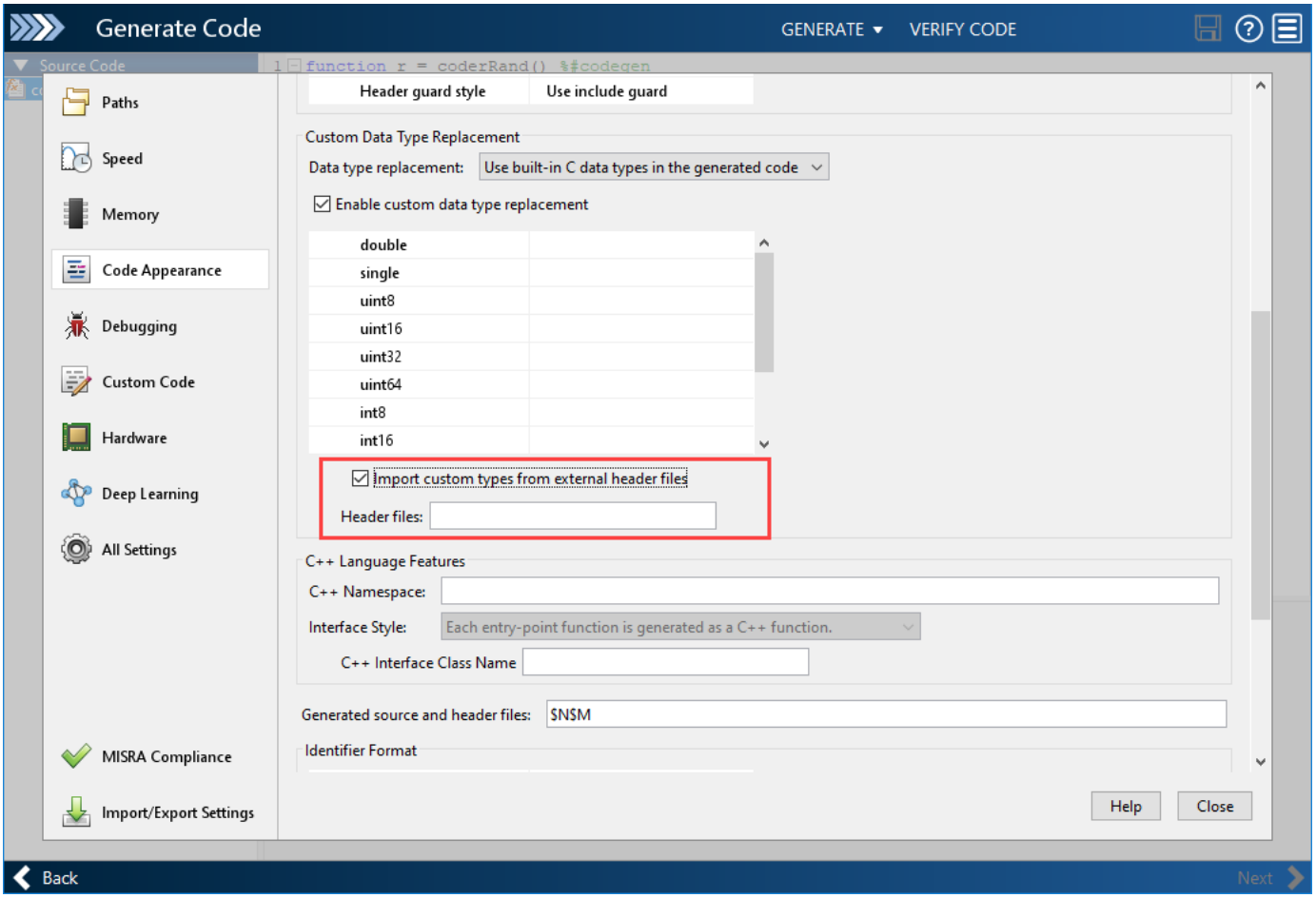
Objects
`coder.EmbeddedCodeConfig` | `coder.CodeConfig` | `coder.ReplacementTypes`

More About

- “Import Custom Data Type Definitions from External Header Files” on page 97-35

Import Custom Data Type Definitions from External Header Files

By default, MATLAB Coder uses the predefined data types in the generated code. You can use customized data type names by importing custom type definitions from external header files. You can then use your own type definitions in the generated C/C++ code. If you have existing legacy code, you can integrate that code, in the generated C/C++ code. You can import a header file by using either the MATLAB Coder app or the command-line interface.



Import Custom Types by Using the MATLAB Coder App

To import custom types from external header files:

1. Open the MATLAB Coder app and proceed to the **Generate Code** page.
2. Click **More Settings**.
3. In the **Code Appearance** tab, select **Enable custom data type replacement**.
4. Specify your custom names for the data types in **Enable custom data type replacement** table and press **Enter**.
5. Select the **Import custom types from external header files** check box.

6 In the **Header files** text field, enter a semicolon-separated list of external header file names and press **Enter**. For example, `myHeader1.h;myHeader2.h;myHeader3.h`.

Import Custom Types by Using the Command-Line Interface

Use the `ReplacementTypes.IsExtern` and `ReplacementTypes.HeaderFiles` properties in a `coder.EmbeddedCodeConfig` object when you generate code by using `codegen`.

1 Create a code configuration object for generation of a static library.

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

2 Specify custom names for the data types. For example, `double` is named as `Custom_Double` and `int8` is named as `Custom_Int8` in the code.

```
cfg.EnableCustomReplacementTypes = true;
cfg.ReplacementTypes.double = "Custom_Double";
cfg.ReplacementTypes.int8 = "Custom_Int8";
```

3 Specify configuration properties for importing the external header files.

```
% Include single header file

cfg.ReplacementTypes.IsExtern = true;
cfg.ReplacementTypes.HeaderFiles = "myHeader.h";
cfg.CustomInclude = 'C:\myFiles'; % Include path of the header file

% Include multiple header files

cfg.ReplacementTypes.IsExtern = true;
cfg.ReplacementTypes.HeaderFiles = "myHeader1.h;myHeader2.h;myHeader3.h";
cfg.CustomInclude = '"C:\Program Files\MATLAB\myFiles"'; % Include path of the header files
```

For more information on `CustomInclude`, see “Configure Build by Using the Configuration Object”.

4 Generate code by using `codegen` and the `-config` option.

```
codegen myAdd.m -args {1,int8(1)} -config cfg -report
```

See Also

Functions

`codegen` | `coder.config` | `coder.cinclude`

Objects

`coder.EmbeddedCodeConfig` | `coder.CodeConfig` | `coder.ReplacementTypes`

More About

- “Customize Data Type Replacement” on page 97-33

MATLAB Code Patterns That Require a Nonempty Initialize Function in Generated Code

Issue

When generating standalone code from MATLAB code, you choose not to generate the initialize function:

- In a `coder.EmbeddedCodeConfig` object, `IncludeInitializeFcn` is set to `false`.
- Alternatively, in the MATLAB Coder app, on the **All Settings** tab, **Initialize Function Required** is set to `No`.

Also, you ensure that the custom code to appear in the generated initialize function is empty:

- In a `coder.EmbeddedCodeConfig` object, `CustomInitializer` is set to `''`.
- Alternatively, in the MATLAB Coder app, on the **Custom Code** tab, **Custom C Code for Generated Files** for initialize function is empty.

You still get this error message:

```
The initialize function is not empty, but the IncludeInitializeFcn configuration setting requests
the initialize function be omitted.
```

Solution

In certain situations, the code generator determines that the initialize function must be nonempty and produces code for the initialize function. This code generation can occur even if the custom code for the initialize function that you specify is empty. Not including this generated code in the initialize function causes the generated entry-point functions to operate on an invalid state. In such situations, if you choose to not generate the initialize function, the code generator produces an error message.

To fix this issue, include the initialize function in your generated code by doing one of the following:

- In a `coder.EmbeddedCodeConfig` object, set `IncludeInitializeFcn` to `true` (this value is the default value).
- In the MATLAB Coder app, on the **All Settings** tab, set **Initialize Function Required** to `Yes` (this setting is the default setting).

Examples of MATLAB code patterns that require a nonempty initialize function in the generated code are:

- Your MATLAB code contains an operation that can generate nonfinite values (`Inf` or `NaN`). For example, define a MATLAB function `foo` that calls `factorial`. The `factorial` function grows quickly and returns `Inf` for inputs greater than a certain threshold. For an input of type `double`, the threshold is `170`. Executing `factorial(171)` in MATLAB returns `Inf`.

  ```
  function y = foo(a)
  y = factorial(a);
  end
  ```

Generate a static library for `foo`. In the `coder.EmbeddedCodeConfig` object `cfg`, set the parameter `CustomInitializer` to an empty character array. Set the parameter `IncludeInitializeFcn` to `true`. These values are the default values for the parameters.

```
cfg = coder.config('lib');

cfg.CustomInitializer = '';  %These are the default values
cfg.IncludeInitializeFcn = true;

codegen -config cfg foo -args {1}
```

The code generator automatically produces a nonempty initialize function `foo_initialize` that calls supporting code for nonfinite data. This function calls `rt_InitInfAndNaN`, which is defined in another generated file `rt_nonfinite.c`.

```
/* Include Files */
#include "foo_initialize.h"
#include "foo.h"
#include "foo_data.h"
#include "rt_nonfinite.h"

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void foo_initialize(void)
{
  rt_InitInfAndNaN();
  isInitialized_foo = true;
}

/*
 * File trailer for foo_initialize.c
 *
 * [EOF]
 */
```

- Your MATLAB code uses `global` or `persistent` variables. For example, define this MATLAB function:

  ```
  function y = bar
  global g
  y = g;
  end
  ```

  Generate a static library for `bar`. Specify the initial value of `g` as `1`. In the `coder.EmbeddedCodeConfig` object `cfg`, set the parameter `CustomInitializer` to an empty character array. Set the parameter `IncludeInitializeFcn` to `true`. These values are the default values for the parameters.

  ```
  cfg = coder.config('lib');

  cfg.CustomInitializer = '';  %These are the default values
  cfg.IncludeInitializeFcn = true;

  codegen -config cfg -globals {'g',1} bar
  ```

The code generator automatically produces a nonempty initialize function `bar_initialize` that initializes `g`.

```
/* Include Files */
#include "bar_initialize.h"
#include "bar.h"
#include "bar_data.h"

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void bar_initialize(void)
{
  g = 1.0;
  isInitialized_bar = true;
}

/*
 * File trailer for bar_initialize.c
 *
 * [EOF]
 */
```

See Also

`coder.EmbeddedCodeConfig`

More About

- “Use Generated Initialize and Terminate Functions”

MISRA and AUTOSAR Compliance of Code Generated from MATLAB

- “Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance” on page 98-2
- “C Coding Directives and Rules Supported for Code Generation” on page 98-6
- “C++ Coding Rules Supported for Code Generation” on page 98-20

Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance

The Motor Industry Software Reliability Association (MISRA)[11] has established “Guidelines for the Use of the C Language in Critical Systems” (MISRA C) and “Guidelines for the Use of the C++ Language in Critical Systems”. The AUTomotive Open System ARchitecture (AUTOSAR) development partnership has also established "Guidelines for the Use of the C++14 Language in Critical and Safety-Related Systems". When generating C or C++ code for your MATLAB code base, you can configure the code generator to produce code that is compliant with a large subset of these coding rules.

Scope of Code Generation Support

When you generate C or C++ code from MATLAB code, you can configure the code generator to maximize the compliance of the generated code with the MISRA C:2012, MISRA C++:2008, and AUTOSAR C++14 guidelines.

To maximize compliance of the generated code:

- Set code generation configuration properties to the recommended values. You can do this either at the command line or in the MATLAB Coder app. See “Configure Code Generation Configuration Object Properties at Command Line” on page 98-3 and “Configure MATLAB Coder App Settings” on page 98-4.
- Make sure any user-written C/C++ files or third-party C/C++ files that you integrate with the generated code by using `coder.ceval` are compliant with the MISRA and AUTOSAR guidelines.

The generated code might include one or more of these files that are not compliant with MISRA and AUTOSAR guidelines:

- `coder_posix_time.h`
- `coder_posix_time.c`
- `rt_defines.h`
- `rt_nonfinite.h`
- `rtGetInf.h`
- `rtGetNaN.h`
- `rtwhalf.h`
- `rtwhalf.c`
- `rtwhalf.cpp`
- `rtwtypes.h`

In certain situations, you might be able to configure the code generator to avoid using the custom type definitions contained in the `rtwtypes.h` file in the generated code. For example, see “Additional Settings for MISRA C++ Compliance” on page 98-5.

The code generator supports a large subset of the required and mandatory coding rules in the MISRA and AUTOSAR guidelines. To see a complete list, refer to these topics:

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

- “C Coding Directives and Rules Supported for Code Generation” on page 98-6
- “C++ Coding Rules Supported for Code Generation” on page 98-20

Code generation interprets these rules in the same way as the Polyspace Bug Finder product. To learn about how the code generator interprets a particular coding rule, see the *Polyspace Implementation* section in the corresponding reference page in the Polyspace Bug Finder documentation:

- “MISRA C:2012 Directives and Rules” (Polyspace Bug Finder)
- “MISRA C++:2008 Rules” (Polyspace Bug Finder)
- “AUTOSAR C++14 Rules” (Polyspace Bug Finder)

Configure Code Generation Configuration Object Properties at Command Line

1. Create an Embedded Coder code generation configuration object.

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

2. Set the properties in the following table to the values for improved MISRA compliance by using the `coder.setupMISRAConfig` function.

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

Property	Value for Improved MISRA Compliance
`CastingMode`	`'Standards'`
`CppGenerateEnumClass` (if target language is C++)	`true`
`CppInterfaceClassName` (if target language is C++)	`''`
`CppInterfaceStyle` (if target language is C++)	`'Functions'`
`CppNamespace` (if target language is C++)	Valid namespace name for C++
`DataTypeReplacement`	`'CoderTypedefs'`
`EnableDynamicMemoryAllocation`	`false`
`EnableOpenMP`	`false`
`EnableRuntimeRecursion`	`false`
`EnableSignedLeftShifts`	`false`
`EnableSignedRightShifts`	`false`
`GenerateDefaultInSwitch`	`true`
`HeaderGuardStyle`	`'UseIncludeGuard'`
`JustifyMISRAViolations`	`true`
`MaxIdLength` (if target language is C)	`31`
`ParenthesesLevel`	`'Maximum'`
`RuntimeChecks`	`false`

Property	Value for Improved MISRA Compliance
`TargetLangStandard`	`'C99 (ISO)'` for C, `'C++11 (ISO)'` for C++

Note `GenerateComments` must be set to `true` to add MISRA and AUTOSAR annotations using `JustifyMISRAViolations`.

`coder.setupMISRAConfig` sets the parameters in this table to the recommended values for improved MISRA compliance. At this point, if you want to override any of these settings, you can modify those property values manually.

If the `CppNamespace` property is unset, and the `TargetLang` property is `'C++'`, then `coder.setupMISRAConfig` sets the `CppNamespace` property to a default character vector, `'Codegen'`. Modify this value to a namespace name that is particular to your project.

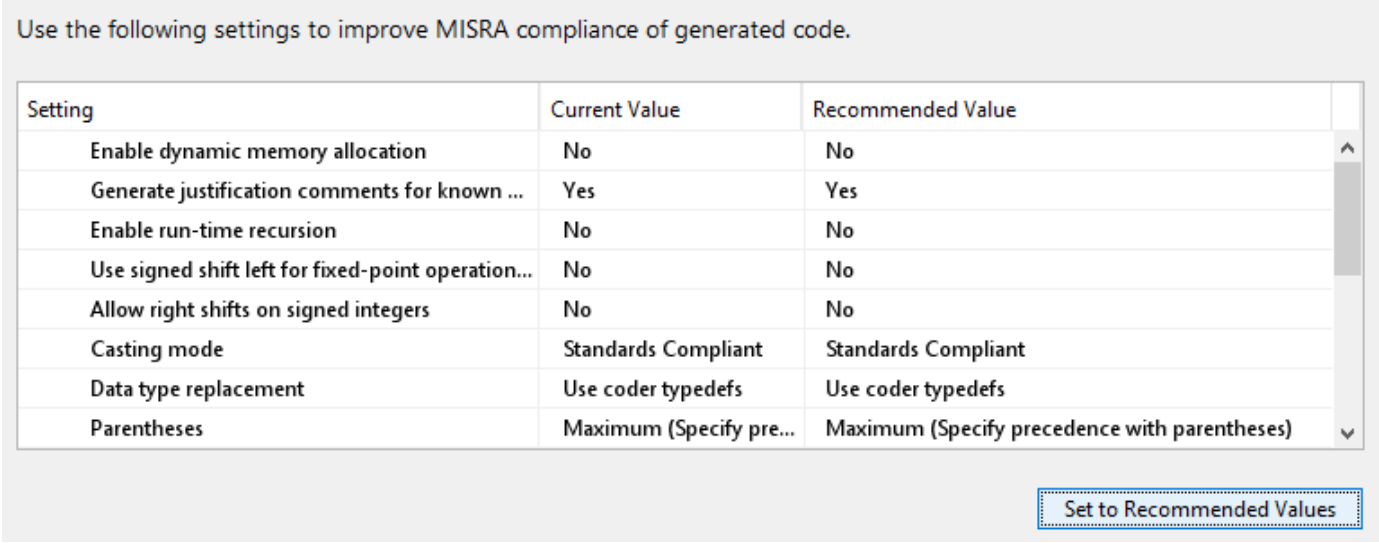
3 Pass the configuration object to `codegen` by using the `-config` option.

```
codegen myFunction -config cfg ...
```

Configure MATLAB Coder App Settings

1 On the **Generate Code** page, set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable`.
2 Click **More Settings**.
3 Click **MISRA Compliance**.

The app displays the settings that can impact MISRA compliance.



4 To set all of these settings to the recommended values, click **Set to Recommended Values**.

To modify an individual setting, click **All Settings**, and then search for the setting.

Additional Settings for MISRA C++ Compliance

By default, the C++ code that you generate uses a custom type definition for `bool` to maximize compatibility with certain precompiled libraries. This custom type definition can cause violations for MISRA C++:2008 Rule 5-0-13 and Rule 5-3-1.

To use the built-in `bool` type in generated C++ code and avoid these violations, change the **Hardware Board** setting in the configuration object from `MATLAB Host Computer`. To see how to change this setting in the App, see "Access Hardware Implementation Properties with a Dialog Box". For example, for code generation and build on a 64-bit Windows platform, change the **Hardware Board** to `None - Select device below`, and then set **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)`.

Check Compliance of Generated Code Using Static Analysis Tool

The best practice is to verify compliance of the generated code independently of the code generator. After you generate code, use a static analysis tool (such as Polyspace Bug Finder) to check the extent of compliance to the MISRA and AUTOSAR coding standards.

To learn how to use Polyspace Bug Finder to check C/C++ code for coding rule violations, see:

- "Polyspace Verification of C/C++ Code Generated by MATLAB Coder" on page 100-22
- "Run Polyspace Bug Finder on Desktop" (Polyspace Bug Finder)

See Also

`coder.EmbeddedCodeConfig` | `coder.setupMISRAConfig`

More About

- "C Coding Directives and Rules Supported for Code Generation" on page 98-6
- "C++ Coding Rules Supported for Code Generation" on page 98-20
- "Polyspace Verification of C/C++ Code Generated by MATLAB Coder" on page 100-22
- "Run Polyspace Bug Finder on Desktop" (Polyspace Bug Finder)

External Websites

- www.misra.org.uk
- www.autosar.org

C Coding Directives and Rules Supported for Code Generation

When you generate C code from MATLAB code, you can configure the code generator to maximize the compliance of the generated code with the required and mandatory coding rules in the MISRA C:2012 guidelines.

The following tables list all the required and mandatory coding directives and rules in MISRA C:2012 Main Document, MISRA C:2012 Amendment 1, and MISRA C:2012 Amendment 2. For each directive or rule, the *Compliance* column has one of these entries:

- Compliant: Generated code is compliant with this directive/rule.
- Not Compliant: In some situations, the generated code might not be compliant with this directive/rule.
- Not Applicable: This directive/rule does not apply to the code generation process. For some of these directives/rules, the *Additional Notes* section suggests actions that you can take to improve the compliance of the generated code.

Note These tables assume that you have already followed these steps to maximize MISRA compliance of the generated code:

- Set the code generation configuration properties to the recommended values. You can do this either at the command line or in the MATLAB Coder app. See “Configure Code Generation Configuration Object Properties at Command Line” on page 98-3 and “Configure MATLAB Coder App Settings” on page 98-4.
- Make sure any user-written C/C++ files or third-party C/C++ files that you integrate with the generated code by using `coder.ceval` are compliant with the MISRA guidelines.

The generated code might include one or more files that are not compliant with MISRA guidelines. For a complete list of these files, see “Scope of Code Generation Support” on page 98-2.

In certain situations, you might be able to configure the code generator to avoid using the custom type definitions contained in the `rtwtypes.h` file in the generated code. For example, see “Additional Settings for MISRA C++ Compliance” on page 98-5.

The code generator interprets these rules in the same way as the Polyspace Bug Finder product. To learn about how the code generator interprets a particular coding rule, see the *Polyspace Implementation* section in the corresponding reference page in the Polyspace Bug Finder documentation: “MISRA C:2012 Directives and Rules” (Polyspace Bug Finder).

Compilation and Code Design Directives

Directive	Description	Compliance	Additional Notes
D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.	Not Compliant	
D2.1	All source files shall compile without any compilation errors.	Compliant	
D3.1	All code shall be traceable to documented requirements.	Not Applicable	To make the generated code compliant, use Requirements Toolbox to include comments in the generated code that contain information about the requirements. See “Requirements Traceability for Code Generated from MATLAB Code” (Requirements Toolbox).
D4.1	Run-time failures shall be minimized.	Not Compliant	
D4.3	Assembly language shall be encapsulated and isolated.	Compliant	
D4.7	If a function returns error information, then that error information shall be tested.	Not Compliant	To reduce this violation, in your MATLAB code, include tests for the output of `coder.ceval` calls.
D4.10	Precautions shall be taken in order to prevent the contents of a header file being included more than once.	Compliant	
D4.11	The validity of values passed to library functions shall be checked.	Not Compliant	To reduce this violation, in your MATLAB code, check the validity of values that you pass to `coder.ceval`.
D4.12	Dynamic memory allocation shall not be used.	Compliant	
D4.14	The validity of values received from external sources shall be checked.	Not Compliant	To reduce this violation, in your MATLAB code, check the validity of values that `coder.ceval` returns.

1. Standard C Environment

Rule	Description	Compliance	Additional Notes
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.	Not Compliant	
1.3	There shall be no occurrence of undefined or critical unspecified behaviour.	Compliant	
1.4	Emergent language features shall not be used.	Compliant	

2. Unused Code

Rule	Description	Compliance	Additional Notes
2.1	A project shall not contain unreachable code.	Not Compliant	
2.2	There shall be no dead code.	Not Compliant	

3. Comments

Rule	Description	Compliance	Additional Notes
3.1	The character sequences /* and // shall not be used within a comment.	Not Compliant	To reduce this violation, make sure the comments that you include in your MATLAB code or external C/C+ code satisfies this rule. Alternatively, consider disabling comment generation by setting the configuration property `GenerateComments` to `false`.
3.2	Line-splicing shall not be used in // comments.	Compliant	

4. Character Sets and Lexical Conventions

Rule	Description	Compliance	Additional Notes
4.1	Octal and hexadecimal escape sequences shall be terminated.	Compliant	

5. Identifiers

Rule	Description	Compliance	Additional Notes
5.1	External identifiers shall be distinct.	Not Compliant	To reduce this violation, make sure your MATLAB entry-point function names contain less than 5 characters.
5.2	Identifiers declared in the same scope and name space shall be distinct.	Compliant	
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	Compliant	
5.4	Macro identifiers shall be distinct.	Compliant	
5.5	Identifiers shall be distinct from macro names.	Compliant	
5.6	A typedef name shall be a unique identifier.	Compliant	
5.7	A tag name shall be a unique identifier.	Compliant	
5.8	Identifiers that define objects or functions with external linkage shall be unique.	Compliant	

6. Types

Rule	Description	Compliance	Additional Notes
6.1	Bit-fields shall only be declared with an appropriate type.	Compliant	
6.2	Single-bit named bit fields shall not be of a signed type.	Compliant	

7. Literals and Constants

Rule	Description	Compliance	Additional Notes
7.1	Octal constants shall not be used.	Compliant	
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.	Compliant	
7.3	The lowercase character "l" shall not be used in a literal suffix.	Compliant	
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".	Compliant	

8. Declarations and Definitions

Rule	Description	Compliance	Additional Notes
8.1	Types shall be explicitly specified.	Compliant	
8.2	Function types shall be in prototype form with named parameters.	Compliant	
8.3	All declarations of an object or function shall use the same names and type qualifiers.	Compliant	
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.	Compliant	
8.5	An external object or function shall be declared once in one and only one file.	Compliant	
8.6	An identifier with external linkage shall have exactly one external definition.	Compliant	
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.	Compliant	
8.10	An inline function shall be declared with the static storage class.	Compliant	
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.	Compliant	
8.14	The restrict type qualifier shall not be used.	Compliant	

9. Initialization

Rule	Description	Compliance	Additional Notes
9.1	The value of an object with automatic storage duration shall not be read before it has been set.	Not Compliant	
9.2	The initializer for an aggregate or union shall be enclosed in braces.	Compliant	
9.3	Arrays shall not be partially initialized.	Compliant	
9.4	An element of an object shall not be initialized more than once.	Compliant	
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.	Compliant	

10. The Essential Type Model

Rule	Description	Compliance	Additional Notes
10.1	Operands shall not be of an inappropriate essential type.	Compliant	
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.	Compliant	
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.	Compliant	
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	Compliant	
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.	Compliant	
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.	Compliant	
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.	Compliant	

11. Pointer Type Conversion

Rule	Description	Compliance	Additional Notes
11.1	Conversions shall not be performed between a pointer to a function and any other type.	Compliant	
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.	Compliant	
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.	Not Compliant	
11.6	A cast shall not be performed between pointer to void and an arithmetic type.	Compliant	
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.	Compliant	
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.	Not Compliant	
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.	Compliant	

12. Expressions

Rule	Description	Compliance	Additional Notes
12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.	Not Compliant	
12.5	The sizeof operator shall not have an operand which is a function parameter declared as "array of type".	Compliant	

13. Side Effects

Rule	Description	Compliance	Additional Notes
13.1	Initializer lists shall not contain persistent side effects.	Compliant	
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.	Not Compliant	
13.5	The right hand operand of a logical && or || operator shall not contain persistent side effects.	Compliant	
13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.	Compliant	

14. Control Statement Expressions

Rule	Description	Compliance	Additional Notes
14.1	A loop counter shall not have essentially floating type.	Compliant	
14.2	A for loop shall be well-formed.	Compliant	
14.3	Controlling expressions shall not be invariant.	Not Compliant	
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.	Compliant	

15. Control Flow

Rule	Description	Compliance	Additional Notes
15.2	The goto statement shall jump to a label declared later in the same function.	Compliant	
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.	Compliant	
15.6	The body of an iteration-statement or a selection-statement shall be a compound-statement.	Compliant	
15.7	All if ... else if constructs shall be terminated with an else statement.	Compliant	

16. Switch Statements

Rule	Description	Compliance	Additional Notes
16.1	All switch statements shall be well-formed.	Compliant	
16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Compliant	
16.3	An unconditional break statement shall terminate every switch-clause.	Compliant	
16.4	Every switch statement shall have a default label.	Compliant	
16.5	A default label shall appear as either the first or the last switch label of a switch statement.	Compliant	
16.6	Every switch statement shall have at least two switch-clauses.	Compliant	
16.7	A switch-expression shall not have essentially Boolean type.	Compliant	

17. Functions

Rule	Description	Compliance	Additional Notes
17.1	The features of <stdarg.h> shall not be used.	Compliant	
17.2	Functions shall not call themselves, either directly or indirectly.	Compliant	
17.3	A function shall not be declared implicitly.	Compliant	
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Compliant	
17.6	The declaration of an array parameter shall not contain the static keyword between the [ ].	Compliant	
17.7	The value returned by a function having non-void return type shall be used.	Compliant	

18. Pointers and Arrays

Rule	Description	Compliance	Additional Notes
18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.	Not Compliant	
18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	Compliant	
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.	Compliant	
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.	Not Compliant	
18.7	Flexible array members shall not be declared.	Compliant	
18.8	Variable-length array types shall not be used.	Compliant	

19. Overlapping Storage

Rule	Description	Compliance	Additional Notes
19.1	An object shall not be assigned or copied to an overlapping object.	Compliant	

20. Preprocessing Directives

Rule	Description	Compliance	Additional Notes
20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name.	Compliant	
20.3	The #include directive shall be followed by either a <filename> or "filename"sequence.	Compliant	
20.4	A macro shall not be defined with the same name as a keyword.	Compliant	
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.	Compliant	
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.	Compliant	
20.8	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.	Compliant	
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.	Compliant	
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.	Compliant	
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.	Compliant	
20.13	A line whose first token is # shall be a valid preprocessing directive.	Compliant	
20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.	Compliant	

21. Standard Libraries

Rule	Description	Compliance	Additional Notes
21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name.	Compliant	
21.2	A reserved identifier or reserved macro name shall not be declared.	Not Compliant	
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.	Compliant	
21.4	The standard header file <setjmp.h> shall not be used.	Compliant	
21.5	The standard header file <signal.h> shall not be used.	Compliant	
21.6	The Standard Library input/output functions shall not be used.	Not Compliant	To reduce this violation, in your MATLAB code, avoid using functions such as `fopen`, `fclose`, `fprintf`, `fscanf`, `fread`, `fwrite`, and `coder.read`.
21.7	The Standard Library functions atof, atoi, atol and atoll of <stdlib.h> shall not be used.	Compliant	
21.8	The Standard Library termination functions of <stdlib.h> shall not be used.	Compliant	
21.9	The Standard Library functions bsearch and qsort of <stdlib.h> shall not be used.	Compliant	
21.10	The Standard Library time and date functions shall not be used.	Not Compliant	To reduce this violation, in your MATLAB code, avoid using functions such as `tic`, `toc`, and `pause`.
21.11	The standard header file <tgmath.h> shall not be used.	Compliant	
21.13	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.	Compliant	
21.14	The Standard Library function memcmp shall not be used to compare null terminated strings.	Compliant	
21.15	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types.	Not Compliant	

Rule	Description	Compliance	Additional Notes
21.16	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.	Not Compliant	
21.17	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.	Compliant	
21.18	The size_t argument passed to any function in <string.h> shall have an appropriate value.	Not Compliant	
21.19	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type.	Compliant	
21.20	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function.	Compliant	
21.21	The Standard Library function system of <stdlib.h> shall not be used.	Compliant	

22. Resources

Rule	Description	Compliance	Additional Notes
22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	Compliant	
22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function.	Compliant	
22.3	The same file shall not be open for read and write access at the same time on different streams.	Compliant	
22.4	There shall be no attempt to write to a stream which has been opened as read-only.	Compliant	
22.5	A pointer to a FILE object shall not be dereferenced.	Compliant	
22.6	The value of a pointer to a FILE shall not be used after the associated stream has been closed.	Compliant	
22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.	Compliant	
22.8	The value of errno shall be set to zero prior to a call to an errno-setting-function.	Compliant	
22.9	The value of errno shall be tested against zero after calling an errno-setting-function.	Compliant	
22.10	The value of errno shall only be tested when the last function to be called was an errno-setting-function.	Compliant	

See Also

`coder.EmbeddedCodeConfig` | `coder.setupMISRAConfig`

Related Examples

- “Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance” on page 98-2

External Websites

- www.misra.org.uk

C++ Coding Rules Supported for Code Generation

When you generate C++ code from MATLAB code, you can configure the code generator to maximize the compliance of the generated code with the required coding rules in the MISRA C++:2008 and AUTOSAR C++14 guidelines.

The following tables list all the required coding rules in the MISRA C++:2008 and AUTOSAR C++14 guidelines. For each directive or rule, the *Compliance* column has one of these entries:

- Compliant: Generated code is compliant with this directive/rule.
- Not Compliant: In some situations, the generated code might not be compliant with this directive/rule.
- Not Applicable: This directive/rule does not apply to the code generation process. For some of these directives/rules, the *Additional Notes* section suggests actions that you can take to improve the compliance of the generated code.

Note These tables assume that you have already followed these steps to maximize MISRA and AUTOSAR compliance of the generated code:

- Set the code generation configuration properties to the recommended values. You can do this either at the command line or in the MATLAB Coder app. See “Configure Code Generation Configuration Object Properties at Command Line” on page 98-3 and “Configure MATLAB Coder App Settings” on page 98-4.
- Make sure any user-written C/C++ files or third-party C/C++ files that you integrate with the generated code by using `coder.ceval` are compliant with the MISRA and AUTOSAR guidelines.

The generated code might include one or more files that are not compliant with MISRA and AUTOSAR guidelines. For a complete list of these files, see “Scope of Code Generation Support” on page 98-2.

In certain situations, you might be able to configure the code generator to avoid using the custom type definitions contained in the `rtwtypes.h` file in the generated code. For example, see “Additional Settings for MISRA C++ Compliance” on page 98-5.

The code generator interprets these rules in the same way as the Polyspace Bug Finder product. To learn about how the code generator interprets a particular coding rule, see the *Polyspace Implementation* section in the corresponding reference page in the Polyspace Bug Finder documentation:

- “MISRA C++:2008 Rules” (Polyspace Bug Finder)
- “AUTOSAR C++14 Rules” (Polyspace Bug Finder)

0. Language Independent Issues

Rule	Description	Compliance	Additional Notes
M0-1-1	A project shall not contain unreachable code.	Compliant	
M0-1-2	A project shall not contain infeasible paths.	Compliant	
M0-1-3	A project shall not contain unused variables.	Not Compliant	
M0-1-4	A project shall not contain non-volatile POD variables having only one use.	Not Compliant	
A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used.	Not Compliant	
A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used.	Compliant	
M0-1-8	All functions with void return type shall have external side effect(s).	Not Compliant	• To add annotations to the generated code for known violations of this rule, set the `JustifyMISRAViolations` and `GenerateComments` configuration properties to `true`. These annotations enable static analysis tools, such as Polyspace, to automatically recognize these comments and report the annotated violations as **Justified**. • To comply with this rule, suppress generation of initialize and terminate functions by setting the configuration properties `IncludeInitializeFcn` and `IncludeTerminateFcn` to `false`.
M0-1-9	There shall be no dead code.	Not Compliant	
A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.	Compliant	
A0-1-4	There shall be no unused named parameters in non-virtual functions.	Compliant	

Rule	Description	Compliance	Additional Notes
A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.	Compliant	
M0-2-1	An object shall not be assigned to an overlapping object.	Compliant	
M0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	Not Applicable	To comply with this rule, you must take the appropriate actions.
M0-3-2	If a function generates error information, then that error information shall be tested.	Not Compliant	To reduce this violation, in your MATLAB code, include tests for the output of `coder.ceval` calls.
M0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	Not Applicable	To make the generated code compliant, use Requirements Toolbox to include comments in the generated code that contain the appropriate documentation. See “Requirements Traceability for Code Generated from MATLAB Code” (Requirements Toolbox).
M0-4-2	Use of floating-point arithmetic shall be documented.	Not Compliant	To make the generated code compliant, use Requirements Toolbox to include comments in the generated code that contain the appropriate documentation. See “Requirements Traceability for Code Generated from MATLAB Code” (Requirements Toolbox).
A0-4-1	Floating-point implementation shall comply with IEEE 754 standard.	Not Applicable	The code generator also assumes that this is the case.
A0-4-2	Type long double shall not be used.	Compliant	
A0-4-3	The implementations in the chosen compiler shall strictly comply with the C++14 Language Standard.	Not Applicable	The code generator also assumes that this is the case.
A0-4-4	Range, domain and pole errors shall be checked when using math functions.	Compliant	

1. General

Rule	Description	Compliance	Additional Notes
A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.	Compliant	
M1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	Not Applicable	
A1-1-2	A warning level of the compilation process shall be set in compliance with project policies.	Not Applicable	You must use the compiler options to specify the compiler warning level.
A1-1-3	An optimization option that disregards strict standard compliance shall not be turned on in the chosen compiler.	Not Applicable	You must use the compiler options to appropriately configure the compiler.
A1-2-1	When using a compiler toolchain (including preprocessor, compiler itself, linker, C++ standard libraries) in safety-related software, the tool confidence level (TCL) shall be determined. In case of TCL2 or TCL3, the compiler shall undergo a "Qualification of a software tool", as per ISO 26262-8.11.4.6 [5].	Not Applicable	
A1-4-1	Code metrics and their valid boundaries shall be defined and code shall comply with defined boundaries of code metrics.	Not Compliant	

2. Lexical Conventions

Rule	Description	Compliance	Additional Notes
A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.	Compliant	
A2-5-1	Trigraphs shall not be used.	Compliant	
A2-5-2	Digraphs shall not be used.	Compliant	
M2-7-1	The character sequence /* shall not be used within a C-style comment.	Compliant	
A2-7-1	The character \ shall not occur as a last character of a C++ comment.	Compliant	
A2-7-2	Sections of code shall not be "commented out".	Not Compliant	To reduce this violation, avoid writing comments that look like code. Consider disabling comment generation by setting the configuration property `GenerateComments` to `false`.
A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	Not Compliant	
A2-7-5	Comments shall not document any actions or sources (e.g. tables, figures, paragraphs, etc.) that are outside of the file.	Not Applicable	
A2-8-1	A header file name should reflect the logical entity for which it provides declarations.	Not Compliant	
M2-10-1	Different identifiers shall be typographically unambiguous.	Compliant	
A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	Compliant	
A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.	Compliant	
A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	Compliant	
A2-11-1	Volatile keyword shall not be used.	Compliant	

Rule	Description	Compliance	Additional Notes
A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.	Compliant	
A2-13-6	Universal character names shall be used only inside character or string literals.	Compliant	
M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	Compliant	
M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	Compliant	
M2-13-4	Literal suffixes shall be upper case.	Compliant	
A2-13-2	String literals with different encoding prefixes shall not be concatenated.	Compliant	
A2-13-3	Type wchar_t shall not be used.	Compliant	
A2-13-4	String literals shall not be assigned to non-constant pointers.	Compliant	

3. Basic Concept

Rule	Description	Compliance	Additional Notes
A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	Compliant	
A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: ".h", ".hpp" or ".hxx".	Compliant	
M3-1-2	Functions shall not be declared at block scope.	Compliant	
A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly.	Compliant	
A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.	Not Compliant	
M3-2-1	All declarations of an object or function shall have compatible types.	Compliant	
M3-2-2	The One Definition Rule shall not be violated.	Compliant	
M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	Compliant	
M3-2-4	An identifier with external linkage shall have exactly one definition.	Compliant	
A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.	Compliant	
A3-3-2	Static and thread-local objects shall be constant-initialized.	Not Compliant	
M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	Compliant	
M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	Not Compliant	
A3-8-1	An object shall not be accessed outside of its lifetime.	Not Compliant	

Rule	Description	Compliance	Additional Notes
M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Compliant	
A3-9-1	Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.	Not Compliant	
M3-9-3	The underlying bit representations of floating-point values shall not be used.	Not Compliant	

4. Standard Conversions

Rule	Description	Compliance	Additional Notes
M4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator.	Not Compliant	
A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [ ], the assignment operator =, the equality operators == and ! =, the unary & operator, and the relational operators <, <=, >, >=.	Compliant	
M4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator.	Not Compliant	
A4-7-1	An integer expression shall not lead to data loss.	Not Compliant	
M4-10-1	NULL shall not be used as an integer value.	Compliant	
A4-10-1	Only nullptr literal shall be used as the null-pointer-constant.	Compliant	
M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant.	Compliant	

5. Expressions

Rule	Description	Compliance	Additional Notes
A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Not Compliant	
M5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Not Compliant	
M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Not Compliant	
M5-0-5	There shall be no implicit floating-integral conversions.	Compliant	
M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	Compliant	
M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	Compliant	
M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	Compliant	
M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	Not Compliant	
M5-0-10	If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Not Compliant	
M5-0-11	The plain char type shall only be used for the storage and use of character values.	Compliant	
M5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	Compliant	
A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool.	Compliant	
M5-0-14	The first operand of a conditional-operator shall have type bool.	Compliant	

Rule	Description	Compliance	Additional Notes
M5-0-15	Array indexing shall be the only form of pointer arithmetic.	Not Compliant	
M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	Not Compliant	
M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	Compliant	
A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes.	Compliant	
M5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Compliant	
A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection.	Compliant	
M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	Not Compliant	
M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	Not Compliant	
A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	Not Compliant	
A5-1-2	Variables shall not be implicitly captured in a lambda expression.	Compliant	
A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression.	Compliant	
A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects.	Compliant	
A5-1-7	A lambda shall not be an operand to decltype or typeid.	Compliant	
M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	Compliant	
A5-2-2	Traditional C-style casts shall not be used.	Not Compliant	

Rule	Description	Compliance	Additional Notes
A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	Not Compliant	
M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	Compliant	
A5-2-4	reinterpret_cast shall not be used.	Compliant	
A5-2-6	The operands of a logical && or || shall be parenthesized if the operands contain binary operators.	Compliant	
M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Compliant	
M5-2-9	A cast shall not convert a pointer type to an integral type.	Compliant	
M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression.	Compliant	
M5-2-11	The comma operator, && operator and the || operator shall not be overloaded.	Compliant	
A5-2-5	An array or container shall not be accessed beyond its range.	Not Compliant	
M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	Not Compliant	
M5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	Compliant	
M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	Not Compliant	
M5-3-3	The unary & operator shall not be overloaded.	Compliant	
M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	Compliant	
A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects.	Compliant	
A5-3-2	Null pointers shall not be dereferenced.	Compliant	

Rule	Description	Compliance	Additional Notes
A5-3-3	Pointers to incomplete class types shall not be deleted.	Compliant	
A5-5-1	A pointer to member shall not access non-existent class members.	Compliant	
A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero.	Not Compliant	
M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	Compliant	
A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant.	Compliant	
M5-14-1	The right hand operand of a logical &&, || operators shall not contain side effects.	Compliant	
A5-16-1	The ternary conditional operator shall not be used as a sub-expression.	Compliant	
M5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	Not Applicable	
M5-18-1	The comma operator shall not be used.	Compliant	
M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around.	Compliant	

6. Statements

Rule	Description	Compliance	Additional Notes
M6-2-1	Assignment operators shall not be used in sub-expressions.	Compliant	
A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.	Compliant	
M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	Not Compliant	
A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only.	Compliant	
M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.	Compliant	
M6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	Compliant	
M6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	Compliant	
M6-4-2	All if ... else if constructs shall be terminated with an else clause.	Compliant	
M6-4-3	A switch statement shall be a well-formed switch statement.	Compliant	
M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Compliant	
M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause.	Compliant	
M6-4-6	The final clause of a switch statement shall be the default-clause.	Compliant	
M6-4-7	The condition of a switch statement shall not have bool type.	Compliant	

Rule	Description	Compliance	Additional Notes
A6-4-1	A switch statement shall have at least two case-clauses, distinct from the default label.	Compliant	
A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.	Not Compliant	
A6-5-2	A for loop shall contain a single loop-counter which shall not have floating-point type.	Compliant	
M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	Compliant	
M6-5-3	The loop-counter shall not be modified within condition or statement.	Compliant	
M6-5-4	The loop-counter shall be modified by one of: --, ++, - = n, or + = n; where n remains constant for the duration of the loop.	Compliant	
M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	Compliant	
M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	Compliant	
A6-6-1	The goto statement shall not be used.	Compliant	
M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	Compliant	
M6-6-2	The goto statement shall jump to a label declared later in the same function body.	Compliant	
M6-6-3	The continue statement shall only be used within a well-formed for loop.	Compliant	

7. Declaration

Rule	Description	Compliance	Additional Notes
A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration.	Not Compliant	
A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time.	Not Compliant	
M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	Not Compliant	
A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.	Not Compliant	
A7-1-4	The register keyword shall not be used.	Compliant	
A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.	Compliant	
A7-1-6	The typedef specifier shall not be used.	Compliant	
A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line.	Compliant	
A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration.	Compliant	
A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type.	Compliant	
A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	Compliant	
A7-2-2	Enumeration underlying base type shall be explicitly defined.	Compliant	

Rule	Description	Compliance	Additional Notes
A7-2-3	Enumerations shall be declared as scoped enum classes.	Compliant	
A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.	Compliant	
M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	Not Compliant	
M7-3-2	The identifier main shall not be used for a function other than the global function main.	Compliant	
M7-3-3	There shall be no unnamed namespaces in header files.	Compliant	
M7-3-4	Using-directives shall not be used.	Compliant	
A7-3-1	All overloads of a function shall be visible from where it is called.	Compliant	
M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	Compliant	
A7-4-1	The asm declaration shall not be used.	Compliant	
M7-4-1	All usage of assembler shall be documented.	Compliant	The code generator does not produce inline assembly language code.
M7-4-2	Assembler instructions shall only be introduced using the asm declaration.	Compliant	
M7-4-3	Assembly language shall be encapsulated and isolated.	Compliant	
M7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	Compliant	
M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	Compliant	
A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const.	Compliant	

Rule	Description	Compliance	Additional Notes
A7-5-2	Functions shall not call themselves, either directly or indirectly.	Compliant	
A7-6-1	Functions declared with the [[noreturn]] attribute shall not return.	Compliant	

8. Declarators

Rule	Description	Compliance	Additional Notes
M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	Compliant	
A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.	Compliant	
M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	Compliant	
A8-4-1	Functions shall not be defined using the ellipsis notation.	Compliant	
M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	Compliant	
A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Compliant	
M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	Compliant	
A8-4-5	consume parameters declared as X && shall always be moved from.	Compliant	
A8-4-6	forward parameters declared as T && shall always be forwarded.	Compliant	
A8-4-7	in parameters for "cheap to copy" types shall be passed by value.	Not Compliant	
A8-4-8	Output parameters shall not be used.	Not Compliant	
A8-4-9	in-out parameters declared as T & shall be modified.	Not Compliant	
A8-4-10	A parameter shall be passed by reference if it can't be NULL	Not Compliant	
A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	Compliant	

Rule	Description	Compliance	Additional Notes
A8-4-12	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	Compliant	
A8-4-13	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a const lvalue reference to express that the function retains a reference count.	Compliant	
A8-4-14	Interfaces shall be precisely and strongly typed.	Not Compliant	
A8-5-0	All memory shall be initialized before it is read.	Not Compliant	
A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.	Compliant	
M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Compliant	
A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization.	Compliant	
A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization.	Compliant	

9. Classes

Rule	Description	Compliance	Additional Notes
M9-3-1	Const member functions shall not return non-const pointers or references to class-data.	Compliant	
A9-3-1	Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.	Compliant	
M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	Not Compliant	
A9-5-1	Unions shall not be used.	Compliant	
M9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	Compliant	Generated code does not rely on the memory layout of bit-fields.
A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes.	Compliant	
A9-6-2	Bit-fields shall be used only when interfacing to hardware or conforming to communication protocols.	Not Applicable	
M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	Compliant	

10. Derived Classes

Rule	Description	Compliance	Additional Notes
A10-0-1	Public inheritance shall be used to implement "is-a" relationship.	Not Applicable	To generate compliant code, design the class hierarchy in your MATLAB code to be compliant with this rule.
A10-0-2	Membership or non-public inheritance shall be used to implement "has-a" relationship.	Not Applicable	To generate compliant code, design the class hierarchy in your MATLAB code to be compliant with this rule.
A10-1-1	Class shall not be derived from more than one base class which is not an interface class.	Compliant	
M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Compliant	
M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	Compliant	
A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes.	Compliant	
A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.	Compliant	
A10-3-2	Each overriding virtual function shall be declared with the override or final specifier.	Compliant	
A10-3-3	Virtual functions shall not be introduced in a final class.	Compliant	
A10-3-5	A user-defined assignment operator shall not be virtual.	Compliant	
M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	Compliant	

11. Member Access Control

Rule	Description	Compliance	Additional Notes
M11-0-1	Member data in non-POD class types shall be private.	Not Compliant	
A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.	Compliant	
A11-3-1	Friend declarations shall not be used.	Compliant	

12. Special Member Functions

Rule	Description	Compliance	Additional Notes
A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.	Compliant	
A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects.	Compliant	
A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.	Not Compliant	
M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	Compliant	
A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	Compliant	
A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.	Not Compliant	
A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	Compliant	
A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor.	Compliant	
A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	Compliant	
A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.	Compliant	
A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers.	Not Compliant	

Rule	Description	Compliance	Additional Notes
A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.	Compliant	
A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.	Compliant	
A12-8-3	Moved-from object shall not be read-accessed.	Compliant	
A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics.	Compliant	
A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment.	Compliant	
A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.	Compliant	

13. Overloading

Rule	Description	Compliance	Additional Notes
A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	Compliant	
A13-1-3	User defined literals operators shall only perform conversion of passed parameters.	Compliant	
A13-2-1	An assignment operator shall return a reference to "this".	Compliant	
A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue".	Compliant	
A13-2-3	A relational operator shall return a boolean value.	Compliant	
A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded.	Compliant	
A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.	Compliant	
A13-5-2	All user-defined conversion operators shall be defined explicit.	Compliant	
A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other.	Compliant	
A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept.	Compliant	
A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.	Compliant	

14. Templates

Rule	Description	Compliance	Additional Notes
A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.	Compliant	
M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	Compliant	
M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.	Compliant	
A14-7-1	A type used as a template argument shall provide all members that are used by the template.	Compliant	
A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.	Compliant	
A14-8-2	Explicit specializations of function templates shall not be used.	Compliant	

15. Exception Handling

Rule	Description	Compliance	Additional Notes
A15-0-1	A function shall not exit with an exception if it is able to complete its task.	Not Compliant	
A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	Compliant	
A15-0-3	Exception safety guarantee of a called function shall be considered.	Compliant	
A15-0-4	Unchecked exceptions shall be used to represent errors from which the caller cannot reasonably be expected to recover.	Not Compliant	
A15-0-5	Checked exceptions shall be used to represent errors from which the caller can reasonably be expected to recover.	Not Compliant	
A15-0-6	An analysis shall be performed to analyze the failure modes of exception handling. In particular, the following failure modes shall be analyzed: (a) worst time execution time not existing or cannot be determined, (b) stack not correctly unwound, (c) exception not thrown, other exception thrown, wrong catch activated, (d) memory not available while exception handling.	Not Compliant	
A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time.	Compliant	
A15-0-8	A worst-case execution time (WCET) analysis shall be performed to determine maximum execution time constraints of the software, covering in particular the exceptions processing.	Not Applicable	To generate compliant code, you must perform the WECT analysis.
A15-1-2	An exception object shall not be a pointer.	Compliant	
M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	Compliant	

Rule	Description	Compliance	Additional Notes
M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	Compliant	
M15-1-2	NULL shall not be thrown explicitly.	Compliant	
M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler.	Compliant	
A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	Compliant	
A15-1-5	Exceptions shall not be thrown across execution boundaries.	Compliant	
A15-2-1	Constructors that are not noexcept shall not be invoked before program startup.	Not Compliant	
A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.	Compliant	
M15-3-1	Exceptions shall be raised only after start-up and before termination.	Compliant	
A15-3-2	If a function throws an exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.	Not Compliant	
A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.	Compliant	
A15-3-4	Catch-all (ellipsis and std::exception) handlers shall be used only in(a) main,(b) task main functions,(c) in functions that are supposed to isolate independent components and(d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.	Compliant	

Rule	Description	Compliance	Additional Notes
M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	Compliant	
M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	Compliant	
A15-3-5	A class type exception shall be caught by reference or const reference.	Compliant	
M15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	Compliant	
M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	Compliant	
A15-4-1	Dynamic exception-specification shall not be used.	Compliant	
A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.	Compliant	
A15-4-3	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider.	Compliant	
A15-4-4	A declaration of non-throwing function shall contain noexcept specification.	Not Compliant	
A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.	Compliant	

Rule	Description	Compliance	Additional Notes
A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.	Not Compliant	
A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.	Not Compliant	
A15-5-3	The std::terminate() function shall not be called implicitly.	Compliant	

16. Preprocessing Directives

Rule	Description	Compliance	Additional Notes
A16-0-1	The pre-processor shall only be used for unconditional and conditional file inclusion and include guards, and using the following directives: (1) #ifndef, (2) #ifdef, (3) #if, (4) #if defined, (5) #elif, (6) #else, (7) #define, (8) #endif, (9) #include.	Not Compliant	
M16-0-1	#include directives in a file shall only be preceded by other pre-processor directives or comments.	Compliant	
M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace.	Compliant	
M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.	Compliant	
M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	Compliant	
M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.	Compliant	
M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.	Compliant	
M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms.	Compliant	
M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Compliant	
M16-2-3	Include guards shall be provided.	Compliant	
A16-2-1	The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.	Compliant	
A16-2-2	There shall be no unused include directives.	Not Compliant	

Rule	Description	Compliance	Additional Notes
A16-2-3	An include directive shall be added explicitly for every symbol used in a file.	Not Compliant	
M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	Compliant	
A16-6-1	#error directive shall not be used.	Compliant	
A16-7-1	The #pragma directive shall not be used.	Compliant	

17. Library Introduction

Rule	Description	Compliance	Additional Notes
A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.	Compliant	
M17-0-2	The names of standard library macros and objects shall not be reused.	Compliant	
M17-0-3	The names of standard library functions shall not be overridden.	Compliant	
A17-0-2	All project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++14 Coding Guidelines.	Not Compliant	
M17-0-5	The setjmp macro and the longjmp function shall not be used.	Compliant	
A17-1-1	Use of the C Standard Library shall be encapsulated and isolated.	Compliant	
A17-6-1	Non-standard entities shall not be added to standard namespaces.	Compliant	

18. Language Support Library

Rule	Description	Compliance	Additional Notes
A18-0-1	The C library facilities shall only be accessed through C++ library headers.	Compliant	
A18-0-2	The error state of a conversion from string to a numeric value shall be checked.	Compliant	
M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	Compliant	
M18-0-4	The time handling functions of library <ctime> shall not be used.	Not Compliant	To reduce this violation, in your MATLAB code, avoid using functions such as `tic`, `toc`, and `pause`.
M18-0-5	The unbounded functions of library <cstring> shall not be used.	Compliant	
A18-0-3	The library <clocale> (locale.h) and the setlocale function shall not be used.	Compliant	
A18-1-1	C-style arrays shall not be used.	Not Compliant	
A18-1-2	The std::vector<bool> specialization shall not be used.	Compliant	
A18-1-3	The std::auto_ptr type shall not be used.	Compliant	
A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.	Compliant	
A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator.	Compliant	
M18-2-1	The macro offsetof shall not be used.	Compliant	
A18-5-1	Functions malloc, calloc, realloc and free shall not be used.	Compliant	
A18-5-2	Non-placement new or delete expressions shall not be used.	Compliant	
A18-5-3	The form of the delete expression shall match the form of the new expression used to allocate the memory.	Compliant	

Rule	Description	Compliance	Additional Notes
A18-5-4	If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.	Compliant	
A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.	Compliant	
A18-5-6	An analysis shall be performed to analyze the failure modes of dynamic memory management. In particular, the following failure modes shall be analyzed: (a) non-deterministic behavior resulting with nonexistence of worst-case execution time, (b) memory fragmentation, (c) running out of memory, (d) mismatched allocations and deallocations, (e) dependence on non-deterministic calls to kernel.	Not Applicable	To generate compliant code, you must perform the failure mode analysis.
A18-5-7	If non-realtime implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-realtime program phases.	Compliant	
A18-5-8	Objects that do not outlive a function shall have automatic storage duration.	Not Compliant	
A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.	Compliant	
A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity.	Compliant	
A18-5-11	operator new and "operator delete" shall be defined together.	Compliant	

Rule	Description	Compliance	Additional Notes
M18-7-1	The signal handling facilities of <csignal> shall not be used.	Compliant	
A18-9-1	The std::bind shall not be used.	Compliant	
A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.	Compliant	
A18-9-3	The std::move shall not be used on objects declared const or const&.	Compliant	
A18-9-4	An argument to std::forward shall not be subsequently used.	Compliant	

19. Diagnostics Library

Rule	Description	Compliance	Additional Notes
M19-3-1	The error indicator errno shall not be used.	Compliant	

20. General Utilities Library

Rule	Description	Compliance	Additional Notes
A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer.	Compliant	
A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership.	Not Compliant	
A20-8-3	A std::shared_ptr shall be used to represent shared ownership.	Not Compliant	
A20-8-4	A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required.	Compliant	
A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr.	Compliant	
A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr.	Compliant	
A20-8-7	A std::weak_ptr shall be used to represent temporary shared ownership.	Compliant	

21. Strings Library

Rule	Description	Compliance	Additional Notes
A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char.	Compliant	

23. Containers Library

Rule	Description	Compliance	Additional Notes
A23-0-1	An iterator shall not be implicitly converted to const_iterator.	Compliant	
A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers.	Compliant	

25. Algorithms Library

Rule	Description	Compliance	Additional Notes
A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied.	Compliant	
A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.	Compliant	
A26-5-1	Pseudorandom numbers shall not be generated using std::rand().	Compliant	
A26-5-2	Random number engines shall not be default-initialized.	Compliant	

27. Input Output Library

Rule	Description	Compliance	Additional Notes
M27-0-1	The stream input/output library <cstdio> shall not be used.	Not Compliant	To reduce this violation, in your MATLAB code, avoid using functions such as `fopen`, `fclose`, `fprintf`, `fscanf`, `fread`, `fwrite`, and `coder.read`.
A27-0-1	Inputs from independent components shall be validated.	Not Compliant	
A27-0-4	C-style strings shall not be used.	Not Compliant	
A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.	Compliant	

See Also

`coder.EmbeddedCodeConfig` | `coder.setupMISRAConfig`

Related Examples

- “Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance” on page 98-2

External Websites

- www.misra.org.uk
- www.autosar.org

Storage Classes for Code Generation from MATLAB Code

- “Storage Classes for Code Generation from MATLAB Code” on page 99-2
- “Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code” on page 99-4

Storage Classes for Code Generation from MATLAB Code

If you have an Embedded Coder license, you can use storage classes to control the declaration and definition of a global variable in the generated C/C++ code.

In the context of code generation, a storage class is a specification that determines the declaration and definition of a variable in the generated code. For code generation, the term storage class is not the same as the C language term storage class specifier.

Storage classes help you to integrate generated code with external code. You can make a generated variable visible to external code. You can also make variables declared in the external code visible to the generated code. For code generation from MATLAB code, you can use storage classes with global variables only. The storage class determines:

- The file placement of a global variable declaration and definition.
- Whether the global variable is imported from external code or exported for use by external code.

To assign a storage class to a global variable, in your MATLAB code, use the `coder.storageClass` function. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generator recognize `coder.storageClass` calls.

The syntax for `coder.storageClass` is:

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

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

`storage_class` can be one of the following values.

Storage Class	Description
`'ExportedGlobal'`	• Defines the variable in the `Variable Definitions` section of the C file *`entry_point_name`*`.c`. • Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`.h` • Initializes the variable in the function *`entry_point_name`*`_initialize.h`.
`'ExportedDefine'`	Declares the variable with a `#define` directive in the `Exported data define` section of the header file *`entry_point_name`*`.h`.
`'ImportedExtern'`	Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`_data.h`. The external code must supply the variable definition.
`'ImportedExternPointer'`	Declares the variable as an `extern` pointer in the `Variable Declarations` section of the header file *`entry_point_name`*`_data.h`. The external code must define a valid pointer variable.

Storage classes have these requirements and limitations:

- Assign the storage class to a global variable in a function that declares the global variable. You do not have to assign the storage class in more than one function.
- After you assign a storage class to a global variable, you cannot assign a different storage class to that global variable.
- You cannot assign a storage class to a constant global variable.
- A global variable with an `ExportedDefine` storage class must be a scalar but not a complex or multi-word scalar. The global variable must only be read and not written to in the code.

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

See Also

`coder.storageClass`

Related Examples

- “Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code” on page 99-4
- “Generate Code for Global Data”

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

This example uses storage classes to control the declarations and definitions of global variables in C/C++ code generated from MATLAB code. Using storage classes helps you to interface generated code with external code.

This example requires an Embedded Coder license.

Write a function `addglobals` that adds four global variables. Declare the global variables in the function.

```
function y = addglobals %#codegen
% Define the global variables.
global u;
global v;
global x;
global z;

% Assign the storage classes.
coder.storageClass('u','ExportedGlobal');
coder.storageClass('v','ImportedExtern');
coder.storageClass('x','ImportedExternPointer');
coder.storageClass('z','ExportedDefine');
y = u + v + x + z;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported global variables `u` and `v`.

```
#include <stdio.h>

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

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

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

Generate the code. This example uses the `-globals` argument to specify the types and initial values of the global variables `u`, `v`, `x`, and `z`. Alternatively, you can define global variables in the MATLAB global workspace. For the imported global variables `v` and `x`, the code generator uses the initial values only to determine the type.

```
codegen -config cfg -globals {'u', 1, 'v', 2, 'x', 3, 'z', 4} addglobals -report
```

From the initial values `1`, `2`, `3`, and `4` `codegen` determines that `u`, `v`, `x` and `z` have type `double`. `codegen` defines and declares the exported global variables `u` and `z`. It generates code that initializes `u` to `1.0` and `z` to `4.0`. `codegen` declares the imported global variables `v` and `x`. It does not define these variables or generate code that initializes them. `myfile.c` provides the code that defines and initializes `v` and `x`.

To view the code generated for the global variables, open the report. Click the **View report** link.

View the definition for the exported global `z` in the `Exported data define` section in `addglobals.h`.

```
/* Definition for custom storage class: ExportedDefine */
#define z                               4.0
```

View the definition and declaration for the exported global `u`.

- `u` is defined in the `Variable Definitions` section in `addglobals.c`.

  ```
  /* Variable Definitions */
  /* Definition for custom storage class: ExportedGlobal */
  double u;
  ```

- `u` is declared as `extern` in the `Variable Declarations` section in `addglobals.h`.

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

- `u` is initialized in `addglobals_initialize.c`.

  ```
  /* Include Files */
  #include "addglobals_initialize.h"
  #include "addglobals.h"
  #include "addglobals_data.h"

  /* Function Definitions */

  /*
   * Arguments    : void
   * Return Type  : void
   */
  void addglobals_initialize(void)
  {
    u = 1.0;
    isInitialized_addglobals = true;
  }
  ```

View the definition and declaration for the imported external global `v` and the imported external global pointer `x`.

`v` and `x` are declared as `extern` in the `Variable Declarations` section in `addglobals_data.h`.

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

/* Declaration for custom storage class: ImportedExternPointer */
extern double *x;
```

See Also

`coder.storageClass`

More About

- “Storage Classes for Code Generation from MATLAB Code” on page 99-2
- “Generate Code for Global Data”
- “Configure Build for External C/C++ Code”

Verification of Code Generated from MATLAB Code

- “Highlight Potential Data Type Issues in a Report” on page 100-2
- “Find Potential Data Type Issues in Generated Code” on page 100-4
- “PIL Execution with ARM Cortex-A at the Command Line” on page 100-9
- “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App” on page 100-11
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” on page 100-12
- “Polyspace Verification of C/C++ Code Generated by MATLAB Coder” on page 100-22

Highlight Potential Data Type Issues in a Report

In this section...

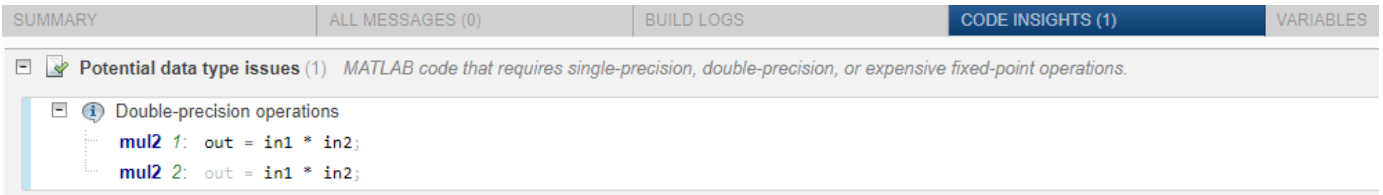
"Enable Highlight Option Using the MATLAB Coder App" on page 100-2
"Enable Highlight Option Using the Command Line Interface" on page 100-2

If you have Embedded Coder, you have the option to highlight potential data types issues in the code generation report for standalone code generated from MATLAB code. If you enable this option, on the **Code Insights** tab, the report identifies single-precision and double-precision operations in the generated C/C++ code. If you have Fixed-Point Designer, the report also identifies expensive fixed-point operations.

To view potential data type issues in a code generation report:

1. Click the **Code Insights** tab.
2. Expand the **Potential data type issues** section.
3. Expand the data type issues section that you want to see.

   For example, this report identifies MATLAB code that results in double-precision operations in the generated code.



The option to highlight potential data type issues is disabled by default.

Enable Highlight Option Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Debugging** tab, select the **Always create a report** and **Highlight potential data type issues** check boxes.

Enable Highlight Option Using the Command Line Interface

1. Create an embedded code configuration object for `'lib'`, `'dll'`, or `'exe'`:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```

2. Set the `GenerateReport` and `HighlightPotentialDataTypeIssues` configuration object properties to `true`:

```
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
```

See Also

Related Examples

- "Find Potential Data Type Issues in Generated Code" on page 100-4

Find Potential Data Type Issues in Generated Code

In this section...

"Data Type Issues Overview" on page 100-4
"Enable Highlighting of Potential Data Type Issues" on page 100-4
"Find and Address Cumbersome Operations" on page 100-5
"Find and Address Expensive Rounding" on page 100-5
"Find and Address Expensive Comparison Operations" on page 100-6
"Find and Address Multiword Operations" on page 100-7

Data Type Issues Overview

When you generate C code from MATLAB code, you can highlight potential data type issues in the C code generation report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations. The expensive fixed-point operations checks require a Fixed-Point Designer.

- The double-precision check highlights expressions that result in a double-precision operation. When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone.

  For a strict-single precision design, specify a language standard that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Language standard** to `C99 (ISO)`.
- The single-precision check highlights expressions that result in a single operation.
- The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see "Tips for Making Generated Code More Efficient" (Fixed-Point Designer).

Enable Highlighting of Potential Data Type Issues

Enable the highlight option using the MATLAB Coder app

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Debugging** tab, select the **Always create a report** and **Highlight potential data type issues** check boxes.

Enable the highlight option using the command-line interface

1. Create an embedded code configuration object for `'lib'`, `'dll'`, or `'exe'`:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```

2. Set the `GenerateReport` and `HighlightPotentialDataTypeIssues` configuration object properties to `true`:

   ```
   cfg.GenerateReport = true;
   cfg.HighlightPotentialDataTypeIssues = true;
   ```

Find and Address Cumbersome Operations

Cumbersome operations usually occur due to an insufficient range of output. Avoid inputs to a multiply or divide operation that have word lengths larger than the base integer type of your processor. Software can process operations with larger word lengths, but this approach requires more code and runs slower.

This example requires Embedded Coder and Fixed-Point Designer. The target word length for the processor in this example is 64.

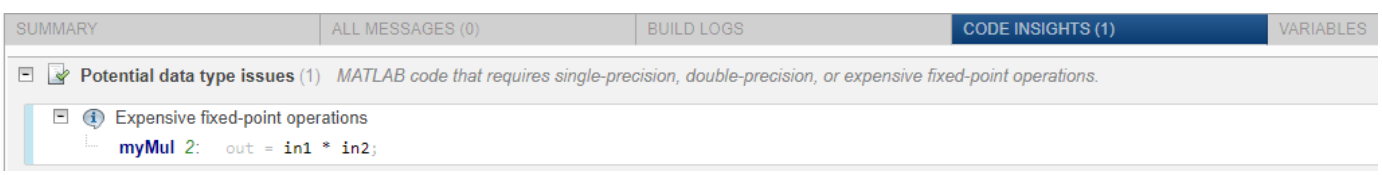
1. Create the function `myMul`.

   ```
   function out = myMul(in1, in2)
       out = fi(in1*in2, 1, 64, 0);
   end
   ```

2. Generate code for `myMul`.

   ```
   cfg = coder.config('lib');
   cfg.GenerateReport = true;
   cfg.HighlightPotentialDataTypeIssues = true;
   fm = fimath('ProductMode', 'SpecifyPrecision', 'ProductWordLength', 64);
   codegen -config cfg myMul -args {fi(1, 1, 64, 4, fm), fi(1, 1, 64, 4, fm)}
   ```

3. Click **View report**.
4. In the code generation report, click the **Code Insights** tab.
5. Expand the **Potential data type issues** section. Then, expand the **Expensive fixed-point operations** section.



   The report flags the expression `in1 * in2`. To resolve the issue, modify the data types of `in1` and `in2` so that the word length of the product does not exceed the target word length of 64.

Find and Address Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method.

This example requires Embedded Coder and Fixed-Point Designer.

1. Create the function `myRounding`.

```
function [quot] = myRounding(in1, in2)
   quot = in1 / in2;
end
```

2. Generate code for `myRounding`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRounding -args {fi(1, 1, 16, 2), fi(1, 1, 16, 4)}
```

3. Click **View report**.
4. In the code generation report, click the **Code Insights** tab.
5. Expand the **Potential data type issues** section. Then, expand the **Expensive fixed-point operations** section.

The division operation `in1/in2` uses the default rounding method, `nearest`. Changing the rounding method to `Floor` provides a more efficient implementation.

Find and Address Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, before comparing an unsigned integer to a signed integer, one of the inputs must be cast to the signedness of the other. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

This example requires Embedded Coder and Fixed-Point Designer.

1. Create the function `myRelop`.

```
function out = myRelop(in1, in2)
    out = in1 > in2;
end
```

2. Generate code for `myRelop`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRelop -args {fi(1, 1, 14, 3, 1), fi(1, 0, 14, 3, 1)}
```

3. Click **View report**.
4. In the code generation report, click the **Code Insights** tab.
5. Expand the **Potential data type issues** section. Then, expand the **Expensive fixed-point operations** section.

The first input argument, `in1`, is signed, while `in2` is unsigned. Extra code is generated because a cast must occur before the two inputs can be compared.

Change the signedness and scaling of one of the inputs to generate more efficient code.

Find and Address Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

This example requires Embedded Coder and Fixed-Point Designer. In this example, the target word length is 64.

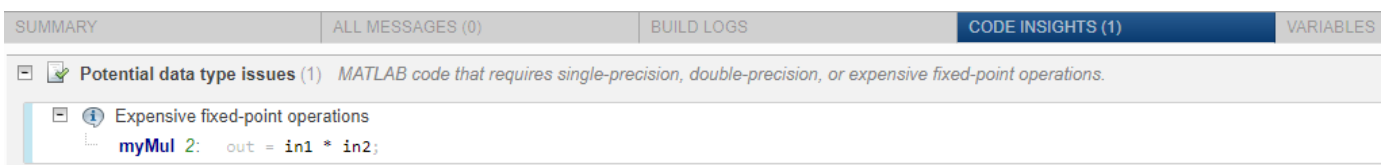
1. Create the function `myMul`.

```
function out = myMul(in1, in2)
    out = in1 * in2;
end
```

2. Generate code for `myMul`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myMul -args {fi(1, 1, 33, 4), fi(1, 1, 32, 4)}
```

3. Click **View report**.
4. In the code generation report, click the **Code Insights** tab.
5. Expand the **Potential data type issues** section. Then, expand the **Expensive fixed-point operations** section.



   The report flags the `in1 * in2` operation in line 2 of `myMul`.

6. In the code pane, pause over `in1`, `in2`, and the expression `in1 * in2`. You see that:
   - The word length of `in1` is 33 bits and the word length of `in2` is 32 bits.
   - The word length of the expression `in1 * in2` is 65 bits.

   The software detects a multiword operation because the word length 65 is larger than the target word length of 64.

7. To resolve this issue, modify the data types of `in1` and `in2` so that the word length of the product does not exceed the target word length. Alternatively, specify the `ProductMode` property of the local `fimath` object.

See Also

More About

- “Highlight Potential Data Type Issues in a Report” on page 100-2
- “Code Generation Reports”

PIL Execution with ARM Cortex-A at the Command Line

This example shows how to set up a PIL execution to verify generated code at the command line.

You can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware by using a MATLAB Coder procedure. You can profile algorithm performance and speed for your generated code. To verify generated code with the MATLAB Coder app, you must have an Embedded Coder license.

This PIL execution is available with these hardware support packages. To use the PIL execution, install one of these support packages.

- Embedded Coder Support Package for BeagleBone® Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors

In the Command Window, select the hardware for PIL execution.

```
hw = coder.hardware('ARM Cortex-A9 (QEMU)')
```

```
hw = 

  Hardware with properties:

            Name: 'ARM Cortex-A9 (QEMU)'
    CPUClockRate: 1000
```

When using the BeagleBone hardware, more hardware properties are supported (`Username`, `Password`, and `DeviceAddress`). Set these properties based on your specific hardware or application.

```
hw = coder.hardware('BeagleBone Black')
```

```
hw = 

  Hardware with properties:

             Name: 'BeagleBone Black'
     CPUClockRate: 1000
         Password: 'root'
         Username: 'admin'
    DeviceAddress: '192.168.1.10'
```

Add the hardware to the MATLAB Coder configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
cfg.Hardware = hw;
```

Generate PIL code for a function, `averaging_filter`.

```
codegen -config cfg averaging_filter -args {zeros(1,16)}
```

For more information on the `averaging_filter` function, see “Generate Code for an Averaging Filter”.

For another example of PIL verification, see the “Processor-in-the-Loop Verification of MATLAB Functions” (Embedded Coder Support Package for ARM Cortex-A Processors) page in the

documentation of the Embedded Coder Support Package for ARM Cortex-A Processors. To install the Embedded Coder Support Package for ARM Cortex-A Processors, see “Embedded Coder Supported Hardware”.

PIL Execution with ARM Cortex-A by Using the MATLAB Coder App

You can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware by using a MATLAB Coder procedure. You can profile algorithm performance and speed for your generated code. To verify generated code with the MATLAB Coder app, you must have an Embedded Coder license.

This PIL execution is available with these hardware support packages. To use the PIL execution, install one of these support packages.

- Embedded Coder Support Package for BeagleBone Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors

You can set up PIL execution with the MATLAB Coder app.

To configure the build type and hardware board:

1. On the **Generate Code** page, in the **Generate** dialog box:
   - Set the **Build type** to `Static Library`.
   - Clear the **Generate code only** check box.
   - Set the **Hardware Board** to `BeagleBone Black` or `ARM Cortex-A9 (QEMU)`.
2. If necessary, modify the settings for your board. To modify the settings, click **More Settings**, and then click **Hardware**.
3. To generate the library, click **Generate**.
4. Set up your PIL execution. Click **Verify Code** to open the **Verify Code** dialog box.

   Because the hardware board is not `MATLAB Host Computer`, the **Verify Code** dialog box is configured for PIL execution.

   In the **Verify Code** dialog box:
   - Enter the name of the test file to use for PIL execution.
   - Select **Generated code**.
5. To start the PIL execution, click **Run Generated Code**.
6. To stop the PIL execution, click **Stop**.

For another example of PIL verification, see the "Processor-in-the-Loop Verification of MATLAB Functions" page in the documentation of the Embedded Coder Support Package for ARM Cortex-A Processors. Install the support package to view the documentation.

Interactively Trace Between MATLAB Code and Generated C/C++ Code

This example shows how to trace between MATLAB source code and the generated C/C++ code. Tracing between source code and generated code helps you to:

- Understand how the code generator implemented your algorithm.
- Debug issues in the generated code.
- Evaluate the quality of the generated code.

If you have Embedded Coder and enable production of a code generation report with traceability, you can view MATLAB source code and generated C/C++ code next to each other. As you move your pointer over code, you can follow highlighted traces to the corresponding generated code or source MATLAB code.

Create the MATLAB Source Code

To illustrate interactive traceability, this example produces a report for generation of a C static library for a MATLAB function `lpsolve` that solves the linear program:

- maximize y = c*x
- subject to A*x <= b
- where x >= 0 and b >= 0

lpsolve

```
%Attempts to solve the linear program:
%
%   maximize y = c*x
%   subject to: A*x <= b,
%   where x >= 0 and b >= 0
%
%   using the dictionary form of the simplex algorithm.
%
%   c must be a double, real row vector.
%   b must be a double, real column vector.
%   A must be a double, real, non-empty matrix with
%     size(A) = [length(b),length(c)].
%
%   Compare to
%   x = linprog(-c(:),A,b,[],[],zeros(length(c),1))
%   y = c*x
%
%   Copyright 2011-2018 The MathWorks, Inc.

function [x,y] = lpsolve(c,A,b)
%#codegen
narginchk(3,3);
m = int32(size(A,1));
n = int32(size(A,2));
if ~validInputs(c,A,b)
    % Something is wrong with the inputs. Return NaNs.
    x = nan(n,1);
```

```
    y = nan;
    return
end
cn = c;
nIdx = 1:n; % indices of non-basic variables
bIdx = n + (1:m); % indices of basic variables
% Perform iterations.
j = findPivotColumn(cn,nIdx);
unbounded = false;
while j > 0 && ~unbounded
    [i,r] = findPivotRow(j,A,b);
    if i == 0 || isinf(r)
        unbounded = true;
    else
        b(i) = r;
        % Take account of the variables entering and leaving the basis.
        entering = nIdx(j);
        leaving = bIdx(i);
        bIdx(i) = entering;
        nIdx(j) = leaving;
        [cn,A,b] = pivot(cn,A,b,i,j);
        j = findPivotColumn(cn,nIdx);
    end
end
% Construct the solution.
if unbounded
    x = nan(n,1);
    y = inf;
else
    x = zeros(n,1);
    y = 0;
    for k = 1:m
        j = bIdx(k);
        if j <= n
            x(j) = b(k);
            y = y + c(j)*x(j);
        end
    end
end

%-------------------------------------------------------------------------------

function p = validInputs(c,A,b)
coder.inline('never');
% All inputs must be double.
p = isa(c,'double') && isa(A,'double') && isa(b,'double');
% All inputs must be real.
p = p && isreal(c) && isreal(A) && isreal(b);
% A must be a non-empty matrix.
p = p && ismatrix(A) && ~isempty(A);
% c must be a row-vector with length = size(A,2).
p = p && isrow(c) && size(c,2) == size(A,2);
% b must be a column-vector with length = size(A,1).
p = p && iscolumn(b) && size(b,1) == size(A,1);
% The elements of b must be non-negative.
p = p && all(b(:) >= 0);

%-------------------------------------------------------------------------------
```

```
function [c,A,b] = pivot(c,A,b,i,j)
% Pivot on element i,j.
coder.inline('never');
m = int32(size(A,1));
aij = A(i,j);
A(i,j) = 1;
A(i,:) = A(i,:)/aij;
for k = 1:m
    if k ~= i
        akj = A(k,j);
        A(k,j) = 0;
        A(k,:) = A(k,:) - akj*A(i,:);
        b(k) = b(k) - akj*b(i);
    end
end
cj = c(j);
c(j) = 0;
c = c - cj*A(i,:);

%-------------------------------------------------------------------------------

function j = findPivotColumn(c,idx)
% Find a suitable variable to enter the basis using the smallest
% subscript rule to prevent cycling.
coder.inline('never');
j = int32(0);
nc = int32(numel(c));
lastidx = int32(0);
for k = 1:nc
    if c(k) > 0 && (lastidx < 1 || idx(k) < lastidx)
        j = k;
        lastidx = idx(k);
    end
end

%-------------------------------------------------------------------------------

function [i,r] = findPivotRow(j,A,b)
% Find the pivot row in column j of A and also return r = b(i)/A(i,j).
coder.inline('never');
r = inf;
i = int32(0);
nb = int32(numel(b));
for k = 1:nb
    if A(k,j) > 0
        rtmp = b(k)/A(k,j);
        if rtmp < r
            r = rtmp;
            i = k;
        end
    end
end

%-------------------------------------------------------------------------------
```

The example also uses a test function `lpsolve_test` that calls `lpsolve` with representative input values.

lpsolve_test

```
% Test function for lpsolve
%
%   Copyright 2011-2018 The MathWorks, Inc.

function lpsolve_test()
c = [2 3 1 1];
A = [2 3 1 -1;1 0 2 1;0 2 1 1];
b = [27;9;18];
[x,y] = lpsolve(c,A,b);
x % display x
y % display y
```

In a writable folder, create the files `lpsolve.m` and `lpsolve_test.m` by copying the code from the expanders in this section.

Prepare for Code Generation

Before you generate C/C++ code, it is a best practice to screen your MATLAB code for code generation readiness.

```
coder.screener('lpsolve')
```

The code generation readiness report indicates that `lpsolve` is suitable for code generation.

It is also a best practice to check for run-time issues by generating and testing a MEX function.

- To specify the types of the inputs arguments, pass representative input values to the `codegen -args` option. Alternatively, because you have a test function, you can use `coder.getArgTypes` to determine the types.
- To generate and test a MEX function, use the `-test` option.

```
c = [2 3 1 1];
A = [2 3 1 -1;1 0 2 1;0 2 1 1];
b = [27;9;18];
codegen lpsolve -args {c A b} -test lpsolve_test
```

```
Running test file: 'lpsolve_test' with MEX function 'lpsolve_mex'.

x =

     5
     7
     0
     4


y =

    35
```

`codegen` successfully generates and runs the MEX function.

Produce a Code Generation Report with Traceability

To generate a report that has interactive traceability:

- Create a `coder.EmbeddedCodeConfig` object. The `EnableTraceability` property controls traceability. By default, the `EnableTraceability` property is `true`.
- Specify the types of the input arguments by passing representative input values to the `-args` option.
- Enable production of a code generation report by using the `-report` option.

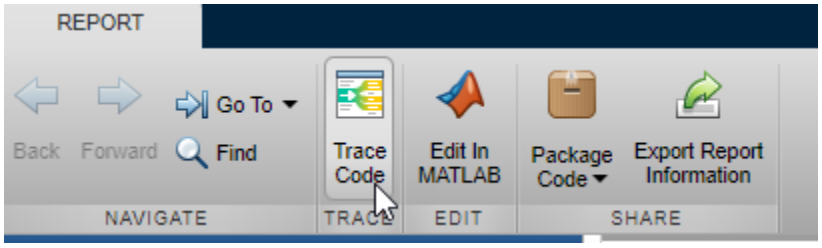
```
cfg = coder.config('lib', 'ecoder', true);
codegen -config cfg lpsolve -args {c A b} -report
```

Access Trace Mode in the Report

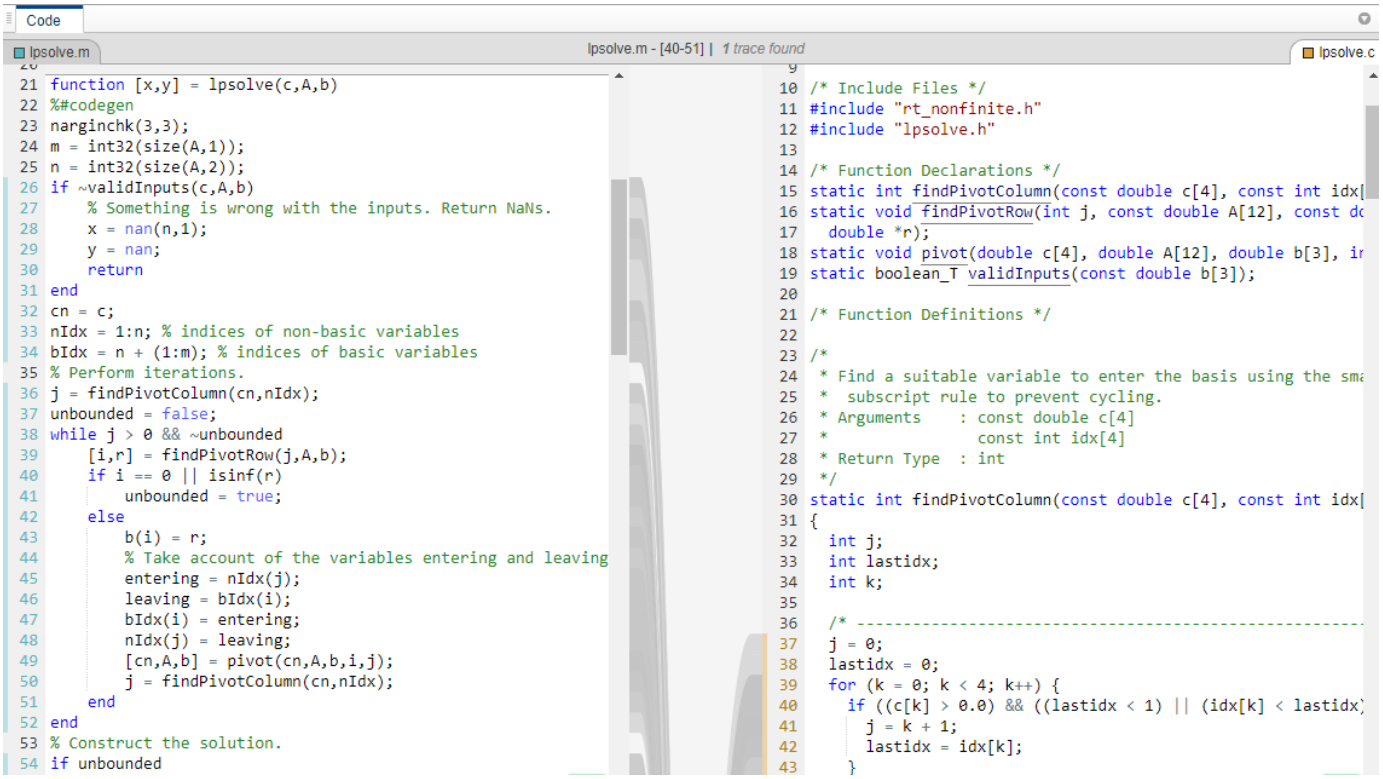
To open the code generation report, click **View report**.

In the code pane, you see `lpsolve.m`.

To enable tracing, on the **Report** tab, click **Trace Code**.



You see the MATLAB source code and the generated C code next to each other.

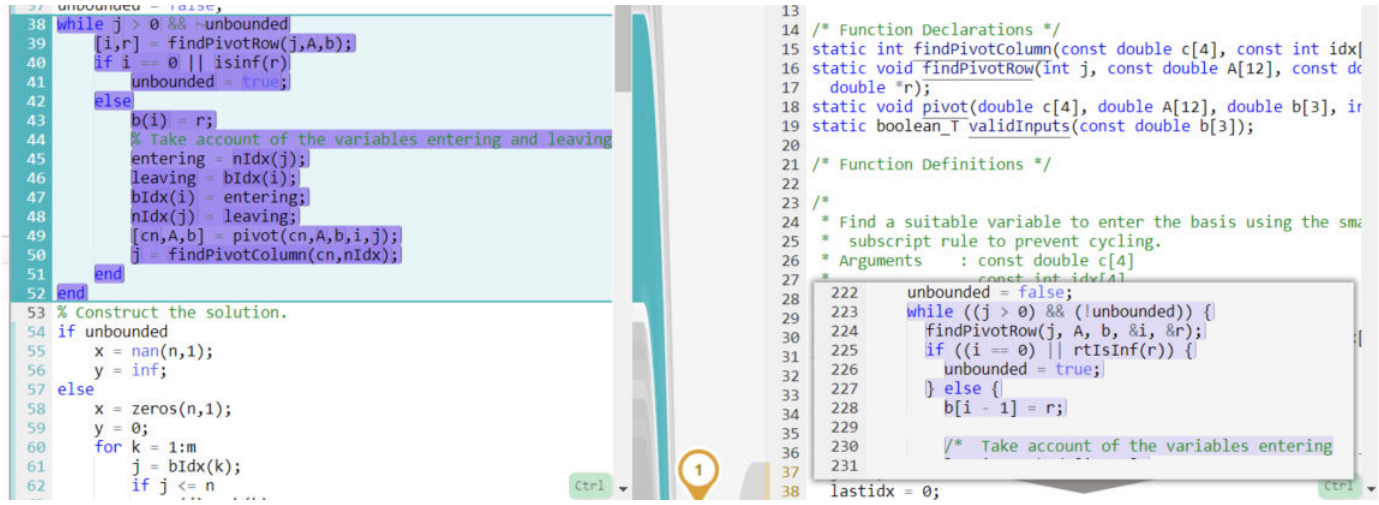


Trace Code

You can trace from the MATLAB code to the C code or from the C code to the MATLAB code. Traceable code is marked with blue on the side that you are tracing from and with orange on the side that you are tracing to. As you move your pointer over traceable code, the code is highlighted in purple and you see the traces to the corresponding code on the other side. When you select highlighted code by clicking it, the code becomes yellow and you can see the traces even when you move your pointer away from the selection. The code remains selected until you press **Esc** or select different code. To change the side that you are tracing from, select code on the other side.

Explore tracing in the example report.

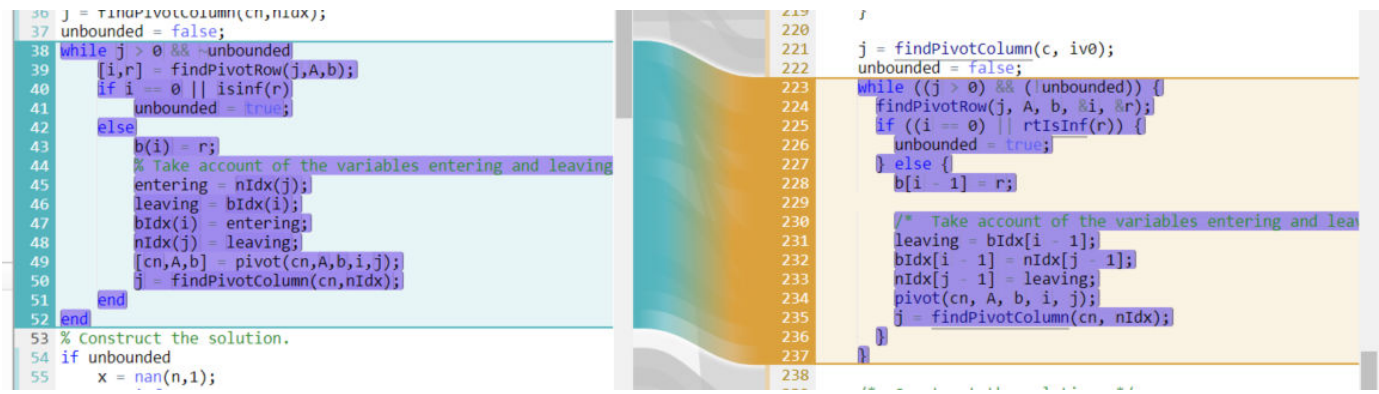
1. In the MATLAB code, point to the `while`-loop that starts at line 38 and scroll down until the entire `while`-loop is in view.



You see this symbol that tells you that the highlighted MATLAB code has one trace that is not in view.

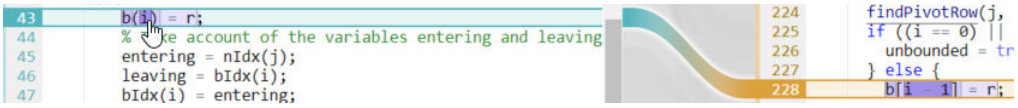
You see the corresponding C code in a separate window in the C code pane.

2 In the C code pane, scroll down until you see the C code that corresponds to the `while` loop.

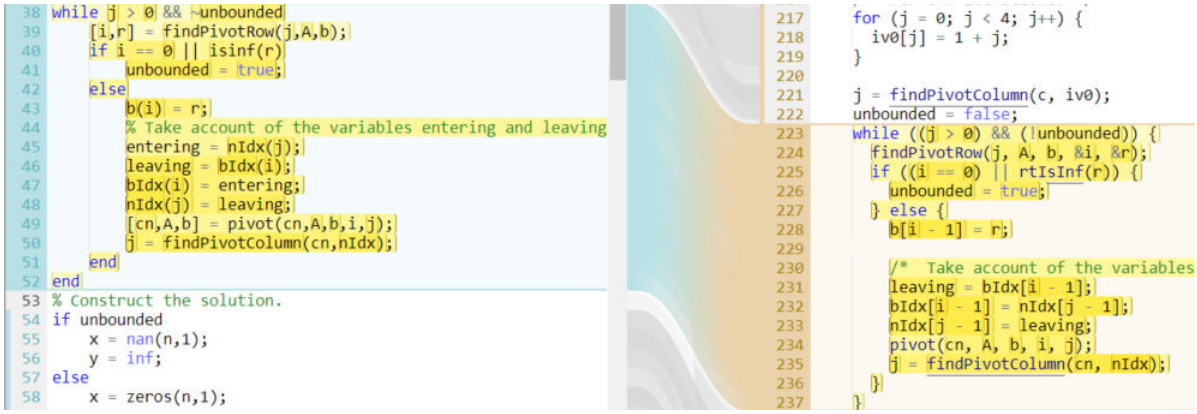


3 In the MATLAB code, move your pointer over different syntax elements in the `while` loop. Highlight variables, expressions, and code blocks.

When you move your pointer over expressions that are part of larger expressions, different shades of purple help you to find the relevant expression in the corresponding C code. For example, at line 43, pause over `i`.



4 In the MATLAB code, move your pointer back to the `while`-loop that starts at line 38. When the entire loop is highlighted in purple, select it by clicking. When you move your pointer outside of the trace, the yellow highlighting identifies the selected trace.



To clear the selection, press **Esc** or select different code.

View Multiple Traces

When code traces to more than one place in the corresponding source or generated code:

- If you pause over the code that you are tracing, at the top of the code pane, you see the number of traces.
- If some traces are not in view, you see a symbol that tells you how many traces are out of view.

- In the code pane, if you select the code that you want to trace, at the top of the code pane, you can select the trace that you want to see.

In the report for `lpsolve`, view multiple traces.

1. Pause over line 36.

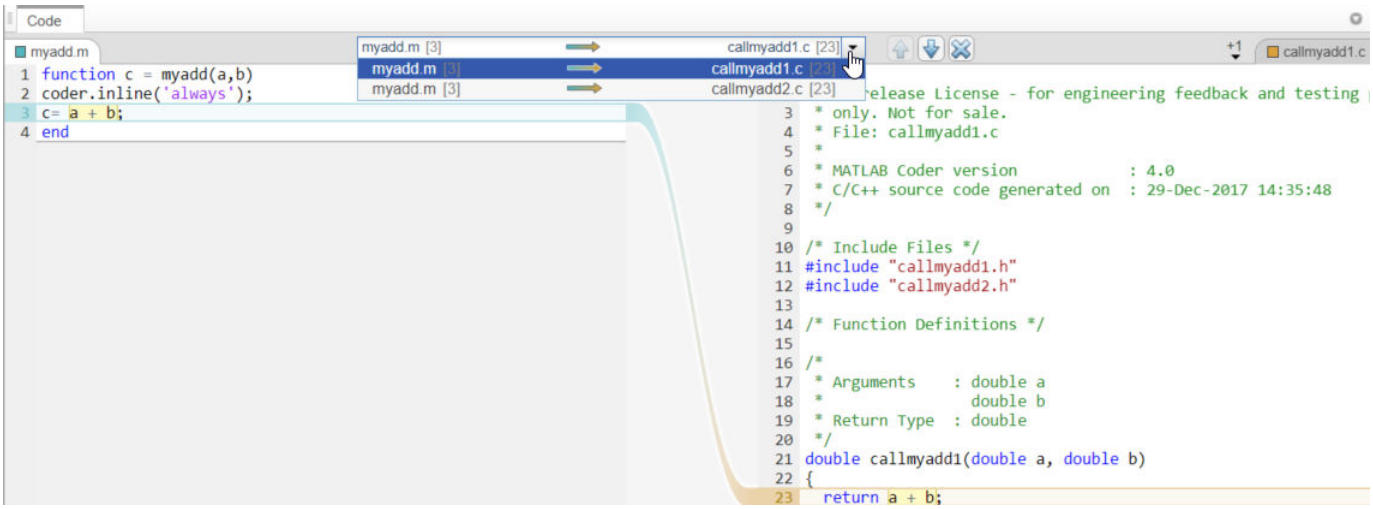
   At the top of the code pane, you see that line 36 has two traces.
2. Select line 36.

   At the top of the code pane, you see the location of the first trace.
3. To list all traces, click the arrow to the right of the traces.

View Traces to Different Files

In the code generation report for `lpsolve`, all traces from the MATLAB code go to one C file `lpsolve.c`. If MATLAB code traces to multiple C files, above the C code, you see a symbol such as

that provides the number of additional files in which you can find a trace. If you click the symbol, you can select the file that you want to see. If you select the MATLAB code, then you can select the trace that you want to see.



Likewise, above the MATLAB source code, a symbol such as +1 indicates that the highlighted C code traces to multiple MATLAB files.

Switch the Locations of the Source and Generated Code

1. To view `lpsolve.c` on the left side of the code pane, in the list of generated files, click `lpsolve.c`.
2. To view the MATLAB code on the left side of the code pane, click a MATLAB function, for example, `lpsolve`.

Enable Code Tooltips and Links

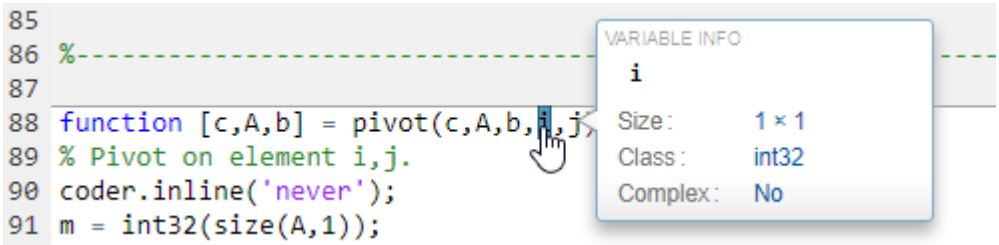
When you are not in trace mode:

- In the MATLAB code, if you point to a variable or expression, a tooltip provides information such as the type.
- In the C code, links go to other parts of the code such as type or function definitions.

When you are in trace mode, to enable MATLAB code tooltips and C code links, hold down **Ctrl**. MATLAB code tooltips are available only for a selected MATLAB function.

In the report for `lpsolve`, view type information for a variable.

1. In the **MATLAB Source** pane, select `pivot`.
2. In the code pane, hold down **Ctrl** and pause over the input argument `i`.



Note On a Macintosh platform, use the **Command** key instead of **Ctrl**.

Disable Traceability

To produce a code generation report that does not include traceability:

- In a `coder.EmbeddedCodeConfig` object, set the `EnableTraceability` property to `false`.
- In the MATLAB Coder app, set **Enable code traceability** to `No`.

See Also

`coder.EmbeddedCodeConfig`

More About

- "Tracing Generated C/C++ Code to MATLAB Source Code"
- "Code Generation Reports"
- "Configure Build Settings"

Polyspace Verification of C/C++ Code Generated by MATLAB Coder

To check for run-time errors in C/C++ code generated from MATLAB code, you can use Polyspace Code Prover. To check for defects, you can use Polyspace Bug Finder. If you have Polyspace and Embedded Coder, Polyspace is integrated into the MATLAB Coder workflow.

- In the MATLAB Coder app, you can run Polyspace analysis without additional setup.
- At the command line, after code generation with `codegen`, you can run Polyspace analysis on the generated code by providing the code generation output folder to `pslinkrun`.

Run Polyspace Analysis in the MATLAB Coder App

1. Generate standalone C/C++ code (a static library, dynamically linked library, or executable program).
2. On the **Generate Code** page, click **Polyspace**.
3. Select the options. See "Configure Advanced Polyspace Options in MATLAB Coder App" (Polyspace Code Prover).
4. Click **Run**.

   The app logs the analysis output on the **Polyspace Log** tab and opens the Polyspace user interface.

Run Polyspace Analysis on Code Generated by codegen

1. Create a `pslinkoptions` object for verification of code generated by MATLAB Coder.
2. As needed, change object properties:
   - In the `ResultDir` property, specify the name of the folder for the Polyspace results.
   - In the `VerificationMode` property, specify the Polyspace verification product.
3. Run the verification by using `pslinkrun`. Provide the `pslinkoptions` object and the folder that contains the generated code.
4. To view verification results, open the Polyspace user interface.

For example, suppose that you generated a static library for the MATLAB function `myFunction` and that the code generation output folder is `codegen/lib/myFunction`. To run Polyspace Code Prover on the generated code, use this code:

```
opts = pslinkoptions('codegen');
opts.ResultDir = 'polyspace';
opts.VerificationMode = 'CodeProver';
pslinkrun('-codegenfolder', 'codegen/lib/myFunction', opts);
polyspaceCodeProver('polyspace/myFunction.psprj');
```

You can also set the `VerificationMode` property to `'BugFinder'` and view verification results by using `polyspaceBugFinder`.

Review Analysis Results

In the **Results List** pane of the Polyspace user interface, review the run-time checks. See whether you can trace the issues back to the original MATLAB code. See "Interactively Trace Between MATLAB Code and Generated C/C++ Code" on page 100-12.

For example, an operation in the C code might overflow because Polyspace assumes an unbounded range for a function input. Consider specifying a constraint on the input and reanalyzing the code with Polyspace. See “Run Polyspace on C/C++ Code Generated from MATLAB Code” (Polyspace Code Prover).

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” (Polyspace Code Prover)
- “Configure Advanced Polyspace Options in MATLAB Coder App” (Polyspace Code Prover)
- “Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance” on page 98-2